Przemysław Kiciak

Wydanie II poprawione, poszerzone i pogłębione

OpenGL i GLSL

(nie taki krótki kurs)

Część II

WSA

OpenGL i GLSL

Przemysław Kiciak

Wydanie II
poprawione,
poszerzone
i pogłębione

OpenGL i GLSL

(nie taki krótki kurs)

Część II

WSA

Projekt okładki Anna Ludwicka
Projekt stron tytułowych Przemysław Kiciak
Redaktor Maria Kasperska
Skład systemem TEX Przemysław Kiciak

ISBN 978-83-971793-2-5 część II
ISBN 978-83-971793-0-1 części I–III

Wydanie II
Warszawa 2024

Własny Sumpt Autora
e-mail: przemek@mimuw.edu.pl
www.mimuw.edu.pl/~przemek

PDF: 19 września 2024 , 6378824 B.

Spis treści części II

15

Druga aplikacja

Druga aplikacja rysuje powierzchnie zakrzywione zbudowane z płatów Béziera. Dla odmiany i poszerzenia horyzontów jest to aplikacja biblioteki GLFW. Wykorzystamy w niej wiele procedur z pierwszej aplikacji. Zamiana biblioteki okienkowej wymaga badzo niewielu zmian tych procedur — większość zmian została wymuszona rosnącym stopniem skomplikowania kolejnych wersji aplikacji i moim dążeniem do utrzymania porządku w plikach źródłowych.

15.1. Krzywe i płaty powierzchni Béziera

O binómio de Newton é tão belo como a Vénus de Milo.
O que há é pouca gente para dar por isso.[1]

Fernando Pessoa

Opis reprezentacji Béziera wielomianowych płatów parametrycznych i jej własności można znaleźć w książkach na ten temat, na przykład w mojej [41]. Matematyka będąca podstawą tej reprezentacji może wystraszyć wiele osób[2], więc nie przedstawiam tu jej zbyt szczegółowo, jednak bez tej matematyki nie ma mowy o eleganckich i efektywnych algorytmach umożliwiających wykonywanie obrazów takich płatów i w będących w moim posiadaniu książkach o OpenGL-u nie takie algorytmy są opisane. Poniższy (zgniły) kompromis wystarczy do przedstawienia algorytmów, których użyjemy, a Czytelników zachęcam do dowiedzenia się więcej z innych źródeł.

Podstawą reprezentacji Béziera krzywych i płatów parametrycznych są **wielomiany bazowe Bernsteina**, określone wzorem

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \ldots, n. \tag{15.1}$$

Krzywą Béziera stopnia n określa się wzorem

$$\boldsymbol{p}(t) = \sum_{i=0}^{n} \boldsymbol{p}_i B_i^n(t), \quad t \in [a, b]. \tag{15.2}$$

[1] Dwumian Newtona jest równie piękny jak Wenus z Milo. Jak niewielu ludzi zdaje sobie z tego sprawę.
[2] nad czym głęboko ubolewam

Odcinek (przedział) $[a,b]$ osi liczbowej jest **dziedziną parametryzacji** $\boldsymbol{p}$. Każdej liczbie t z tego przedziału odpowiada pewien punkt $\boldsymbol{p}(t)$ krzywej. Krzywa ta znajduje się w przestrzeni, do której należą tzw. **punkty kontrolne** $\boldsymbol{p}_0,\dots,\boldsymbol{p}_n$. Dla każdego $t\in\mathbb{R}$ jest spełniona równość $\sum_{i=0}^{n} B_i^n(t)=1$, dzięki czemu każdy punkt $\boldsymbol{p}(t)$ jest kombinacją afiniczną punktów kontrolnych krzywej.

Przedział $[a,b]$ może być wybrany dowolnie, ale zazwyczaj przyjmuje się, że jest to przedział $[0,1]$. Wszystkie wielomiany Bernsteina przyjmują w nim wartości nieujemne, co w powiązaniu z faktem, że ich suma dla każdego t jest równa 1, sprawia, że jeśli $[a,b]=[0,1]$, to krzywa jest położona w **otoczce wypukłej** zbioru swoich punktów kontrolnych.

Łamana kontrolna składa się z n odcinków, a jej kolejnymi wierzchołkami są punkty $\boldsymbol{p}_0,\dots,\boldsymbol{p}_n$. Jest ona przybliżeniem krzywej Béziera; ponieważ $\boldsymbol{p}(0)=\boldsymbol{p}_0$ oraz $\boldsymbol{p}(1)=\boldsymbol{p}_n$, punktami końcowymi krzywej są pierwszy i ostatni wierzchołek łamanej kontrolnej. Jeśli f oznacza dowolne przekształcenie afiniczne, to obraz $f(\boldsymbol{p})$ krzywej $\boldsymbol{p}$ jest reprezentowany przez punkty kontrolne $f(\boldsymbol{p}_0),\dots,f(\boldsymbol{p}_n)$. Zatem, aby poddać krzywą Béziera dowolnemu przekształceniu afinicznemu (np. obrócić ją lub przesunąć), wystarczy zastosować to przekształcenie do jej punktów kontrolnych.

Tensorowy płat Béziera stopnia (n,m) jest określony wzorem

$$\boldsymbol{p}(u,v)=\sum_{i=0}^{n}\sum_{j=0}^{m}\boldsymbol{p}_{ij}B_i^n(u)B_j^m(v),\quad u\in[a,b],\ v\in[c,d]. \tag{15.3}$$

Dziedziną parametryzacji płata jest zatem prostokąt $[a,b]\times[c,d]$, przy czym zazwyczaj przyjmuje się, że $a=c=0$, $b=d=1$, a wtedy dziedzina ta jest kwadratem jednostkowym, $[0,1]^2$. Krzywą można widzieć jako powyginany i porozciągany odcinek i podobnie płat tensorowy jest powyginanym i porozciąganym kwadratem.

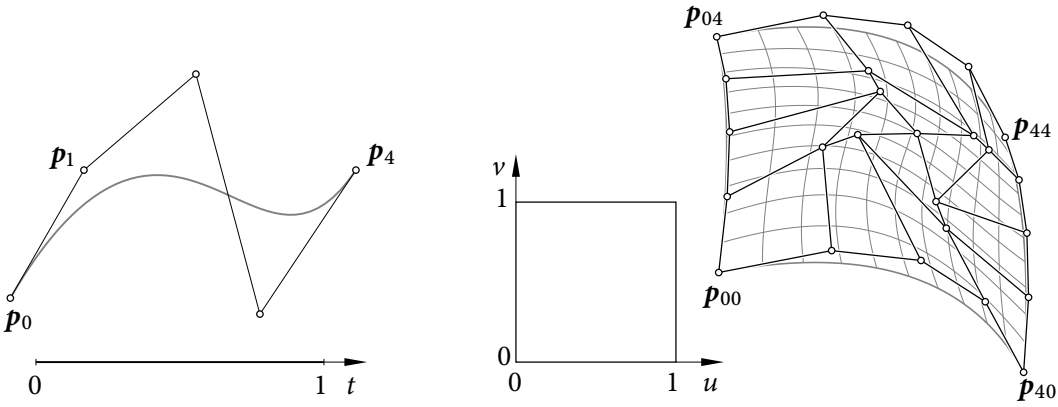
W zbiorze **punktów kontrolnych płata**, $\boldsymbol{p}_{ij}$, wyróżniamy $n+1$ **kolumn** (ciągów $\boldsymbol{p}_{i0},\dots,\boldsymbol{p}_{im}$) oraz $m+1$ **wierszy** (ciągów $\boldsymbol{p}_{0j},\dots,\boldsymbol{p}_{nj}$), które wygodnie jest przedstawiać jako łamane. W ten sposób powstaje **siatka kontrolna** — odpowiednik łamanej kontrolnej krzywej. Jej kształt określa kształt płata. Podobnie jak dla krzywej, aby otrzymać obraz płata Béziera w dowolnym przekształceniu afinicznym, wystarczy przekształcić jego siatkę kontrolną.

Wzór definiujący płat możemy przepisać w postaci

$$\boldsymbol{p}(u,v)=\sum_{i=0}^{n}\underbrace{\left(\sum_{j=0}^{m}\boldsymbol{p}_{ij}B_j^m(v)\right)}_{\boldsymbol{q}_i}B_i^n(u)=\sum_{i=0}^{n}\boldsymbol{q}_iB_i^n(u). \tag{15.4}$$

Wynika z niej, że mając dane liczby u,v, możemy obliczyć $n+1$ punktów, $\boldsymbol{q}_0,\dots,\boldsymbol{q}_n$, z których każdy jest punktem krzywej Béziera stopnia m reprezentowanej przez odpowiednią kolumnę siatki kontrolnej. Otrzymamy w ten sposób **krzywą stałego parametru** v płata; punkt $\boldsymbol{p}(u,v)$ płata jest punktem tej krzywej, odpowiadającym danemu u.[3]

[3] Alternatywnie, zamiast kolumn możemy potraktować *wiersze* siatki kontrolnej jak łamane kontrolne krzywych Béziera stopnia n; otrzymamy w ten sposób reprezentację Béziera krzywej stałego parametru u płata i możemy obliczać punkty płata jako punkty tej krzywej.



Rysunek 15.1. Krzywa Béziera i tensorowy płat Béziera

Zanim zajmiemy się algorytmami znajdowania punktów krzywych Béziera, zbadajmy wynikające z powyższych spostrzeżeń (niektóre) własności płatów i (niektóre) ich konsekwencje praktyczne. Zauważamy, że

$$\boldsymbol{p}(u,0) = \sum_{i=0}^{n} \boldsymbol{p}_{i0} B_i^n(u), \quad \boldsymbol{p}(u,1) = \sum_{i=0}^{n} \boldsymbol{p}_{im} B_i^n(u),$$
$$\boldsymbol{p}(0,v) = \sum_{j=0}^{m} \boldsymbol{p}_{0j} B_j^m(v), \quad \boldsymbol{p}(1,v) = \sum_{j=0}^{m} \boldsymbol{p}_{nj} B_j^m(v),$$

a zatem skrajne wiersze i kolumny siatki kontrolnej wyznaczają cztery krzywe brzegowe płata o dziedzinie $[0,1]^2$, co więcej, narożniki siatki kontrolnej są narożnikami płata: $\boldsymbol{p}(0,0) = \boldsymbol{p}_{00}$, $\boldsymbol{p}(1,0) = \boldsymbol{p}_{n0}$, $\boldsymbol{p}(0,1) = \boldsymbol{p}_{0m}$, $\boldsymbol{p}(1,1) = \boldsymbol{p}_{nm}$.

Do wykonania obrazu potrzebny jest algorytm obliczania punktu $\boldsymbol{p}(u,v)$ dla danych liczb u, v oraz wektora $\boldsymbol{n}(u,v) = \boldsymbol{p}_u(u,v) \wedge \boldsymbol{p}_v(u,v)$, tj. iloczynu wektorowego pochodnych cząstkowych parametryzacji $\boldsymbol{p}$, który jest wektorem normalnym płata w punkcie $\boldsymbol{p}(u,v)$; wektor ten natychmiast po obliczeniu unormujemy, aby otrzymać wektor jednostkowy. Potrzebujemy zatem algorytmu obliczania pochodnych cząstkowych płata Béziera. Wyprowadzanie go zaczniemy od spojrzenia na krzywe.

Aby obliczać punkty krzywej Béziera, oznaczmy $s = 1 - t$ i rozpiszmy wzór (15.2)[4]:

$$\begin{aligned}\boldsymbol{p}(t) &= \boldsymbol{p}_0\tbinom{n}{0}s^n + \boldsymbol{p}_1\tbinom{n}{1}ts^{n-1} + \cdots + \boldsymbol{p}_{n-1}\tbinom{n}{n-1}t^{n-1}s + \boldsymbol{p}\tbinom{n}{n}t^n \\ &= \Big(\ldots\big(\boldsymbol{p}_0\tbinom{n}{0}s + \boldsymbol{p}_1\tbinom{n}{1}t\big)s + \cdots + \boldsymbol{p}_{n-1}\tbinom{n}{n-1}t^{n-1}\Big)s + \boldsymbol{p}\tbinom{n}{n}t^n.\end{aligned}$$

Na podstawie tego wzoru możemy obliczyć punkt $\boldsymbol{p}(t)$ jako wartość (wektorowego) wielomianu zmiennej s, korzystając ze schematu Hornera. W tym celu trzeba obliczyć (wektorowe) współczynniki $\boldsymbol{p}_i\binom{n}{i}t^i$ tego wielomianu w bazie potęgowej. Punkty kontrolne $\boldsymbol{p}_i$

[4] Na początku otwieramy $n-1$ nawiasów, z których każdy zamkniemy po dodaniu kolejnego składnika postaci $\boldsymbol{p}_i\binom{n}{i}t^i$, a po zamknięciu wyrażenie w nawiasie pomnożymy przez s.

mamy dane, kolejne potęgi parametru t obliczymy w pętli, a do obliczenia współczynników dwumianowych Newtona możemy użyć wzorów

$$\binom{n}{0} = 1, \quad \binom{n}{1} = n, \quad \binom{n}{i+1} = \binom{n}{i}(n-i)/(i+1), \ i = 1, \ldots, n-1.$$

Współczyniki dwumianowe są liczbami całkowitymi; w szczególności iloczyn $\binom{n}{i}(n-i)$ jest zawsze podzielny przez $i+1$ i tu w działaniach stałopozycyjnych nie ma błędów zaokrągleń. Ale może być nadmiar — dla 32-bitowych liczb całkowitych ze znakiem wystąpi on, gdy $n \geqslant 30$. W praktycznych zastosowaniach mamy do czynienia z krzywymi i płatami znacznie niższych stopni.

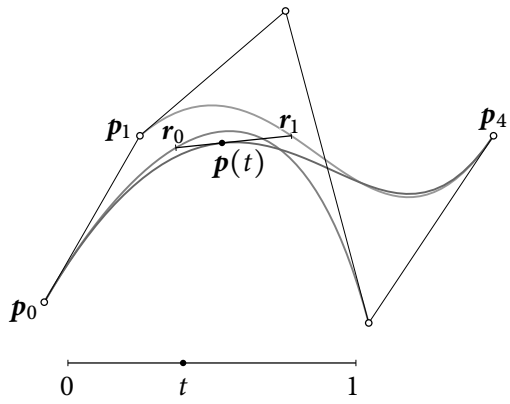
Aby znaleźć punkt krzywej i wektor pochodnej parametryzacji krzywej Béziera, możemy skorzystać z faktu, że parametryzacja określona wzorem (15.2) dla $n > 0$ ma równoważne przedstawienie w postaci

$$\boldsymbol{p}(t) = (1-t)\underbrace{\sum_{i=0}^{n-1} \boldsymbol{p}_i B_i^{n-1}(t)}_{\boldsymbol{r}_0} + t\underbrace{\sum_{i=0}^{n-1} \boldsymbol{p}_{i+1} B_i^{n-1}(t)}_{\boldsymbol{r}_1} = (1-t)\boldsymbol{r}_0 + t\boldsymbol{r}_1, \tag{15.5}$$

a pochodna parametryzacji w punkcie t wyraża się wzorem

$$\boldsymbol{p}'(t) = \sum_{i=0}^{n-1} n(\boldsymbol{p}_{i+1} - \boldsymbol{p}_i) B_i^{n-1}(t) = n(\boldsymbol{r}_1 - \boldsymbol{r}_0). \tag{15.6}$$

Obliczenie punktu $\boldsymbol{p}(t)$ i wektora $\boldsymbol{p}'(t)$ dla danego t można zatem zacząć od znalezienia punktów $\boldsymbol{r}_0$ i $\boldsymbol{r}_1$ położonych na dwóch krzywych Béziera stopnia $n-1$: pierwsza z nich jest reprezentowana przez punkty kontrolne $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_{n-1}$, a druga przez punkty $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$ (rys. 15.2).



Rysunek 15.2. Obliczanie punktu i wektora pochodnej krzywej Béziera

Teraz zobaczmy, jak można obliczyć punkt tensorowego płata Béziera i pochodne cząstkowe jego parametryzacji. Pochodna cząstkowa względem u jest pochodną krzywej stałego parametru v, będącej krzywą Béziera reprezentowaną przez punkty $\boldsymbol{q}_0, \ldots, \boldsymbol{q}_n$ (zobacz

wzór (15.4)). Natomiast pochodną cząstkową względem v możemy otrzymać, jeśli oprócz punktów $\boldsymbol{q}_i$ (będących punktami krzywych Béziera stopnia m reprezentowanych przez kolumny siatki kontrolnej płata) dla ustalonego parametru v obliczymy wektory $\boldsymbol{q}'_0, \ldots, \boldsymbol{q}'_n$ — pochodne tych krzywych w punkcie v. Mamy zatem

$$\boldsymbol{p}_u(u,v) = n\sum_{i=0}^{n-1}\sum_{j=0}^{m}(\boldsymbol{p}_{i+1,j} - \boldsymbol{p}_{ij})B_i^{n-1}(u)B_j^m(v) = n(\boldsymbol{r}_1 - \boldsymbol{r}_0), \tag{15.7}$$

$$\boldsymbol{p}_v(u,v) = m\sum_{i=0}^{n}\sum_{j=0}^{m-1}(\boldsymbol{p}_{i,j+1} - \boldsymbol{p}_{ij})B_i^n(u)B_j^{m-1}(v) = \sum_{i=0}^{n}\boldsymbol{q}'_i B_i^n(u), \tag{15.8}$$

przy czym punkty $\boldsymbol{r}_0$ i $\boldsymbol{r}_1$ znajdziemy podczas obliczania punktu $\boldsymbol{p}(u,v)$ na krzywej stałego parametru v.

Obliczenia pochodnych cząstkowych płata Béziera możemy nie dokończyć, pomijając mnożenie przez czynniki n i m; pochodne są potrzebne do obliczenia jednostkowego wektora normalnego płata, a zatem ich długości są nieistotne — istotne są tylko kierunki i zwroty tych wektorów, bo natychmiast po obliczeniu ich iloczynu wektorowego (przy użyciu dostępnej w GLSL-u funkcji `cross`) iloczyn ten unormujemy. Implementacje w GLSL-u algorytmów tu opisanych są pokazane na listingu 15.4.

15.2. Wymierne płaty Béziera

Wspomnijmy w tym miejscu o **wymiernych płatach Béziera**. Otrzymujemy je, konstruując płaty wielomianowe (takie jak opisane wyżej) w przestrzeni $\mathbb{R}^4$; ich punkty kontrolne są wektorami współrzędnych jednorodnych punktów w $\mathbb{R}^3$. Punkt $\boldsymbol{P}(u,v)$ płata wielomianowego w $\mathbb{R}^4$ (tzw. **płata jednorodnego**) traktujemy jak wektor współrzędnych jednorodnych punktu $\boldsymbol{p}(u,v)$ płata wymiernego w przestrzeni trójwymiarowej. Współrzędne kartezjańskie tego punktu jak zwykle otrzymamy przez podzielenie pierwszych trzech współrzędnych jednorodnych przez czwartą współrzędną (wagową). Jeśli wszystkie punkty kontrolne mają taką samą współrzędną wagową (która nie może być zerem), to mamy zwykły wielomianowy tensorowy płat Béziera. Ale jeśli dopuszczamy niejednakowe współrzędne wagowe punktów kontrolnych, to mamy istotnie szerszą klasę płatów powierzchni[5].

Siatkę kontrolną płata wymiernego możemy narysować jako obiekt trójwymiarowy; w tym celu obliczamy współrzędne kartezjańskie jej wierzchołków, dzieląc pierwsze trzy współrzędne jednorodne przez współrzędną wagową. Modelując płaty wymierne, postępujemy odwrotnie: rozmieszczamy punkty kontrolne w przestrzeni trójwymiarowej i dobieramy ich wagi; na podstawie punktów w $\mathbb{R}^3$ i wag możemy łatwo obliczyć punkty kontrolne płata jednorodnego w $\mathbb{R}^4$.

Dla płata wymiernego również potrzebujemy obliczać punkty i wektory normalne. Podam przepis (bez dowodu), jak to czynić. Jeśli symbole $\boldsymbol{P}(u,v)$, $\boldsymbol{P}_u(u,v)$ i $\boldsymbol{P}_v(u,v)$ oznaczają

[5] Zawierającą m.in. wszystkie kwadryki, tj. powierzchnie drugiego stopnia, a także powierzchnie obrotowe, których tworzące mają parametryzacje wymierne.

odpowiednio punkt płata jednorodnego i jego pochodne cząstkowe w punkcie (u,v), to iloczyn wektorowy *trzech* wektorów w $\mathbb{R}^4$

$$\boldsymbol{N}(u,v) = \boldsymbol{P}(u,v) \wedge \boldsymbol{P}_v(u,v) \wedge \boldsymbol{P}_u(u,v)$$

reprezentuje (jednoznacznie) płaszczyznę styczną do płata wymiernego w $\mathbb{R}^3$. W szczególności, biorąc pierwsze trzy współrzędne tego iloczynu, otrzymamy wektor normalny $\boldsymbol{n}(u,v)$ tej płaszczyzny[6].

Uwaga: Przestawienie dowolnych dwóch argumentów iloczynu wektorowego powoduje zmianę zwrotu wyniku tego działania. Zatem, wektor w $\mathbb{R}^3$ utworzony z pierwszych trzech współrzędnych wektora $\boldsymbol{P}(u,v) \wedge \boldsymbol{P}_u(u,v) \wedge \boldsymbol{P}_v(u,v)$ jest zorientowany przeciwnie do wektora $\boldsymbol{n}(u,v)$ określonego wyżej. Wybór tego wektora zapewnia zgodność zwrotu z wektorem $\boldsymbol{p}_u(u,v) \wedge \boldsymbol{p}_v(u,v)$, tj. iloczynem wektorowym (podanych w tej kolejności) pochodnych cząstkowych parametryzacji płata. Jest to istotne, bo jeśli w pewnym wierzchołku trójkąta otrzymanego w wyniku rozdrabniania płata pochodne cząstkowe są liniowo zależne, szader geometrii podstawi w miejsce ich iloczynu wektorowego (który jest wektorem zerowym) wektor normalny płaszczyzny trójkąta, aby „załatać" tę osobliwość. Zwrot tego wektora musi być zgodny ze zwrotami obliczonych przez szader rozdrabniania wektorów normalnych płata w pozostałych wierzchołkach trójkąta, tak dla płatów wielomianowych, jak i wymiernych.

Ponieważ biblioteka GLSL-a nie zawiera standardowej funkcji obliczającej iloczyn wektorowy w $\mathbb{R}^4$, trzeba taką funkcję napisać samemu. Dla wektorów $\boldsymbol{a} = (x_{\boldsymbol{a}}, y_{\boldsymbol{a}}, z_{\boldsymbol{a}}, w_{\boldsymbol{a}})$, $\boldsymbol{b} = (x_{\boldsymbol{b}}, y_{\boldsymbol{b}}, z_{\boldsymbol{b}}, w_{\boldsymbol{b}})$, $\boldsymbol{c} = (x_{\boldsymbol{c}}, y_{\boldsymbol{c}}, z_{\boldsymbol{c}}, w_{\boldsymbol{c}})$ jest taki wzór:

$$\boldsymbol{a} \wedge \boldsymbol{b} \wedge \boldsymbol{c} = \begin{bmatrix} -d_{34}y_{\boldsymbol{c}} + d_{24}z_{\boldsymbol{c}} - d_{23}w_{\boldsymbol{c}} \\ d_{34}x_{\boldsymbol{c}} - d_{14}z_{\boldsymbol{c}} + d_{13}w_{\boldsymbol{c}} \\ -d_{24}x_{\boldsymbol{c}} + d_{14}y_{\boldsymbol{c}} - d_{12}w_{\boldsymbol{c}} \\ d_{23}x_{\boldsymbol{c}} - d_{13}y_{\boldsymbol{c}} + d_{12}z_{\boldsymbol{c}} \end{bmatrix},$$

a w nim występują liczby

$$d_{12} = x_{\boldsymbol{a}}y_{\boldsymbol{b}} - y_{\boldsymbol{a}}x_{\boldsymbol{b}}, \quad d_{13} = x_{\boldsymbol{a}}z_{\boldsymbol{b}} - z_{\boldsymbol{a}}x_{\boldsymbol{b}}, \quad d_{14} = x_{\boldsymbol{a}}w_{\boldsymbol{b}} - w_{\boldsymbol{a}}x_{\boldsymbol{b}},$$
$$d_{23} = y_{\boldsymbol{a}}z_{\boldsymbol{b}} - z_{\boldsymbol{a}}y_{\boldsymbol{b}}, \quad d_{24} = y_{\boldsymbol{a}}w_{\boldsymbol{b}} - w_{\boldsymbol{a}}y_{\boldsymbol{b}}, \quad d_{34} = z_{\boldsymbol{a}}w_{\boldsymbol{b}} - w_{\boldsymbol{a}}z_{\boldsymbol{b}}.$$

Iloczyn wektorowy z uwagi na często stosowany symbol „×" jest nazywany po angielsku *cross product*, dlatego wbudowana procedura obliczająca iloczyn wektorowy w $\mathbb{R}^3$ ma nazwę `cross`. Możemy użyć tej samej nazwy dla podprogramu obliczającego iloczyn wektorowy w $\mathbb{R}^4$, ale lepiej jest użyć innej nazwy, np. `cross4`, bo nazwa podprogramu napisanego w treści szadera zasłoniłaby nazwę funkcji wbudowanej[7]. Podprogram obliczający pierwsze trzy współrzędne iloczynu wektorowego jest pokazany na listingu 15.1.

[6] Jeśli $\boldsymbol{N}(u,v) = \left[\begin{smallmatrix} \boldsymbol{n} \\ w \end{smallmatrix}\right]$, to punkt $\boldsymbol{q} = \boldsymbol{o} - \frac{w}{\boldsymbol{n}^T\boldsymbol{n}}\boldsymbol{n}$ jest punktem tej płaszczyzny położonym najbliżej początku $\boldsymbol{o}$ układu współrzędnych w $\mathbb{R}^3$ — oczywiście nie musi on leżeć na płacie $\boldsymbol{p}$.

[7] Można użyć tej samej nazwy dla podprogramów różniących się listami parametrów, ale identyfikatory podprogramów zadeklarowanych w treści szadera zasłaniają identyfikatory podprogramów wbudowanych (zobacz podrozdz. 9.13). W treści szadera można zatem napisać instrukcję wywołującą podprogram wbudowany, na przykład `cross`, a potem podprogramy lub prototypy podprogramów o tej samej nazwie. Wtedy w zapisanych dalej instrukcjach te podprogramy zasłonią swojego wbudowanego imiennika.

Listing 15.1. Podprogram obliczania iloczynu wektorowego w $\mathbb{R}^4$

GLSL

```
vec3 cross4 ( vec4 a, vec4 b, vec4 c )
{
  float d12, d13, d14, d23, d24, d34;

  d12 = a.x*b.y-a.y*b.x;  d13 = a.x*b.z-a.z*b.x;  d14 = a.x*b.w-a.w*b.x;
  d23 = a.y*b.z-a.z*b.y;  d24 = a.y*b.w-a.w*b.y;  d34 = a.z*b.w-a.w*b.z;
  return vec3 ( -d34*c.y+d24*c.z-d23*c.w, d34*c.x-d14*c.z+d13*c.w,
                -d24*c.x+d14*c.y-d12*c.w );
} /*cross4*/
```

15.3. Szadery

Program szaderów do rysowania płatów Béziera korzysta z mechanizmu rozdrabniania; w pamięci GPU umieścimy tablicę z punktami kontrolnymi i będziemy rysować płat czworokątny. Jego dziedziną jest kwadrat jednostkowy $[0,1]^2$, dokładnie taki, jaki standardowo przyjmuje się za dziedzinę tensorowych płatów Béziera. Etap rozdrabniania podzieli tę dziedzinę na trójkąty, które przekaże do dalszych etapów potoku przetwarzania grafiki. Zadaniem szadera rozdrabniania jest obliczenie, dla podanego wierzchołka trójkątów w rozdrobnionej dziedzinie, odpowiedniego punktu płata Béziera i wektora normalnego płata w tym punkcie.

Opisane w znanych mi książkach o OpenGL-u procedury rysowania płatów Béziera zakładają, że punkty kontrolne płatów są dane w tablicy wierzchołków, z której poszczególne wierzchołki z odpowiednimi atrybutami są wybierane przez etap pobierania wierzchołków. Ten mechanizm jest wygodny i elastyczny, ale ma podstawowe ograniczenie: małą maksymalną liczbę wierzchołków płata. Limit liczby wierzchołków płata można poznać, wykonując instrukcję

```
glGetIntegerv ( GL_MAX_PATCH_VERTICES, &n );
```

W implementacji, której używam, płat może mieć co najwyżej 32 wierzchołki, co wystarczy do reprezentowania płatów Béziera stopnia $(5,4)$, ale już nie $(5,5)$.[8] Dlatego zastosujemy inne rozwiązanie: tablicę punktów kontrolnych umieścimy w bloku magazynowym, a etap pobierania wierzchołków będzie pobierał dane z pustego VAO. Szader rozdrabniania będzie brał punkty kontrolne z tego bloku, a nie ze zmiennej wbudowanej (tablicy) `gl_in`, której zawartość zignoruje. Tracimy przy tym nieco elastyczności, bo pobieranie wierzchołków musimy oprogramować samemu i raczej nie zrobimy tego dla *wszystkich* reprezentacji

[8] Płat stopnia (n,m) ma $(n+1)(m+1)$ punktów kontrolnych. Ograniczenie liczby wierzchołków płata w OpenGL-u można by ominąć, korzystając z tego, że wierzchołek może mieć wiele atrybutów, wystarczy zatem „przemycać" po kilka punktów kontrolnych płata Béziera jako kolejne atrybuty jednego wierzchołka. W punkcie 15.8.1 podałem informacje, które mogą w tym pomóc. Ale nadmiar trików programistycznych może prowadzić do niespodzianek.

liczb (dopuścimy tylko liczby zmiennopozycyjne pojedynczej precyzji — chyba że ktoś sobie doprogramuje inne możliwości). Jednakże etap pobierania wierzchołków narzuca używanie współrzędnych jednorodnych (wektorów w $\mathbb{R}^4$), a tak możemy osobno oprogramować przetwarzanie płatów w $\mathbb{R}^2$, $\mathbb{R}^3$ i $\mathbb{R}^4$, otrzymując znacznie prostsze i trochę szybsze procedury dla pierwszych dwóch przypadków.

Napiszemy program składający się z pięciu szaderów; podobnie jak w aplikacji 1D wypełnią one wszystkie programowalne etapy w potoku przetwarzania grafiki. Program ten ma umożliwiać rysowanie wielu płatów jednocześnie, przy czym mogą to być wielomianowe płaty płaskie oraz wielomianowe i wymierne płaty w przestrzeni trójwymiarowej. Punkty kontrolne umieścimy w tablicy w bloku magazynowym; ponadto umożliwimy użycie dodatkowej tablicy indeksów do tablicy punktów kontrolnych. Dzięki niej można zmniejszyć długość tablicy punktów kontrolnych — przez wyeliminowanie kopii punktów wspólnych dla dwóch lub większej liczby płatów (np. wchodzących w skład wspólnego wiersza lub kolumny siatek kontrolnych płatów mających wspólną krzywą brzegową).

Listing 15.2. Szader wierzchołków

GLSL

```
1: #version 430 core
2:
3: layout(location=0) out int instance;
4:
5: void main ( void )
6: {
7:   instance = gl_InstanceID;
8: } /*main*/
```

Szader wierzchołków tego programu jest pokazany na listingu 15.2. Nie nadaje on wartości wbudowanej zmiennej wyjściowej `gl_Position`, bo dalej wartość tej zmiennej zostanie zignorowana. Istotną informacją wyprowadzaną przez ten szader jest **numer instancji** płata. Aby spowodować rysowanie, aplikacja wywoła procedurę `glDrawArraysInstanced`, której ostatni parametr określa *liczbę instancji* rysowanego prymitywu: jeśli jest ona równa n, to poszczególne instancje są ponumerowane od 0 do $n-1$. Numer instancji jest podawany szaderowi wierzchołków w zmiennej wbudowanej `gl_InstanceID` (typu `int`), a ten kopiuje ją do zmiennej wyjściowej `instance`.

Adaptacyjne dobieranie parametrów podziału dziedziny płata dostosowanych do wielkości obrazu jest zadaniem dość skomplikowanym i dlatego na początek zadowolimy się ich „ręcznym" dobieraniem[9]. Na listingu 15.3 jest pokazany szader sterowania rozdrabnianiem, który przypisuje wszystkim czterem elementom tablicy `gl_TessLevelOuter` i obu elementom tablicy `gl_TessLevelInner` tę samą wartość, wziętą z pola `TessLevel` w bloku magazynowym `BezPatch`, zawierającym opis zbioru płatów Béziera do narysowania; blok ten,

[9] W podrozdziale 23.5 przedstawię pewien algorytm adaptacyjnego dobierania parametrów rozdrabniania dla poszczególnych płatów, zaimplementowany przy użyciu szaderów obliczeniowych.

w którym większość informacji jest używana w obliczeniach wykonywanych przez szader rozdrabniania, jest opisany dalej[10].

Listing 15.3. Szader sterowania rozdrabnianiem

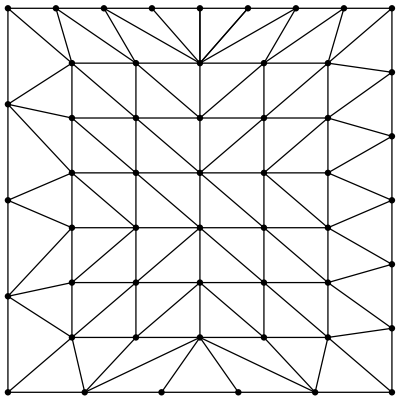
GLSL

```
#version  430 core

layout(vertices=1) out;

layout(location=0) in int instance[];

layout(location=0) out int inst[];

layout(std430,binding=2) buffer BezPatch {
    int  npatches, dim, udeg, vdeg;
    int  stride_u, stride_v, stride_p, stride_q, nq;
    bool use_ind;
    vec3 Colour;
    int  TessLevel;
    bool BezNormals;
  } bezp;

void main ( void )
{
  if ( gl_InvocationID == 0 ) {
    gl_TessLevelOuter[0] = gl_TessLevelOuter[1] =
    gl_TessLevelOuter[2] = gl_TessLevelOuter[3] =  bezp.TessLevel;
    gl_TessLevelInner[0] = gl_TessLevelInner[1] =  bezp.TessLevel;
    inst[gl_InvocationID] = instance[gl_InvocationID];
  }
} /*main*/
```

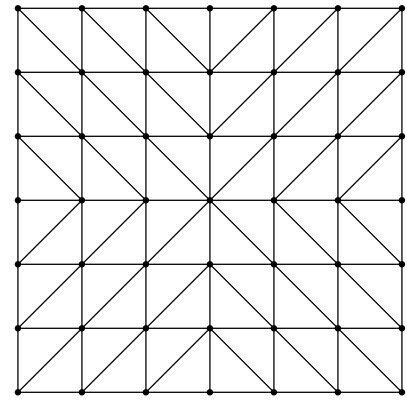
W tej aplikacji przyjmiemy, że każdy płat ma tylko jeden wierzchołek; w konsekwencji wartość zmiennej wbudowanej `gl_InvocationID`, która jest numerem wierzchołka prymitywu (płata) wprowadzonego do potoku przetwarzania grafiki, jest zawsze równa 0, ale obliczenia, które mają być wykonane tylko dla jednego wierzchołka płata, są wykonywane w instrukcji warunkowej — dla bezpieczeństwa w razie późniejszych zmian aplikacji. Zmienna wyjściowa `inst` jest tablicą o długości 1. Szader przekazuje w niej otrzymany od szadera wierzchołków numer instancji rysowanego płata, potrzebny dalej szaderowi rozdrabniania.

Przykłady podziału dziedziny na trójkąty po podaniu różnych parametrów są pokazane na rysunku 15.3. W ogólności etap rozdrabniania dziedziny nie wytwarza triangulacji Delaunaya (co byłoby świetne, ale chyba zbyt kosztowne, zobacz [56]), ale to, co wytwarza, może być.

[10] Zamiast szadera sterowania rozdrabnianiem możemy użyć procedury `glPatchParameterfv`, zgodnie z opisem w rozdziale 12.



`gl_TessLevelOuter == {4,5,6,8}`
`gl_TessLevelInner == {6,7}`



`gl_TessLevelOuter == {6,6,6,6}`
`gl_TessLevelInner == {6,6}`

Rysunek 15.3. Podziały kwadratowej dziedziny płata

Szader rozdrabniania jest taki długi, że jego listing, 15.4, trzeba było rozdrobnić na pięć części. Pierwsza z nich pokazuje deklaracje wejścia/wyjścia oraz bloków magazynowych i bloków zmiennych jednolitych używanych przez szader. W linii 5 jest podana informacja (dla etapu rozdrabniania dziedziny), że dziedzina, którą należy podzielić (na trójkąty) jest czworokątem, przy czym boki dziedziny mają być podzielone równomiernie, a orientacja trójkątów przekazywanych dalej ma być przeciwna do ruchu wskazówek zegara (`ccw` — *counterclockwise*). Tablica `instance` (linia 7) zawiera *w pierwszym elemencie* numer instancji, który jest interpretowany jako numer płata do narysowania.

Blok wyjściowy `GVertex`, który będzie wejściem dla szadera geometrii, zawiera trzy znajome atrybuty wierzchołka: kolor (który tu pobierzemy z bloku zmiennych jednolitych `BezPatch`) oraz położenie wierzchołka w przestrzeni i wektor normalny płata, podane w układzie współrzędnych świata.

W bloku `CPoints` (linie 15–17) jest tablica `cp` ze współrzędnymi punktów kontrolnych. Tablica nie ma podanej długości; za utworzenie bufora o odpowiedniej długości i dopilnowanie, aby program nie wychodził poza zakres jej indeksów, jest odpowiedzialna aplikacja.

Tablica `cpi` w bloku `CPIndices` (linie 19–21) zawiera indeksy do tablicy punktów kontrolnych. Jej faktyczna długość też jest ustalana przez aplikację.

Blok `BezPatch`, pokazany ze szczegółami na listingu 15.3, zawiera opis zbioru płatów Béziera do narysowania. Pole `npatches` zawiera liczbę płatów[11]. Wartość zmiennej `dim`, 2, 3 lub 4, jest liczbą współrzędnych punktów kontrolnych. Zmienne `udeg` i `vdeg` przechowują stopnie wszystkich płatów w tym zbiorze ze względu na parametry u i v. Zmienne `stride_u`, `stride_v`, `stride_p`, `stride_q` i `nq` służą do uzyskania dostępu do właściwych punktów kontrolnych, co będzie opisane dalej. Zmienna `use_ind` określa, czy po punkty kontrolne (do tablicy `CPoints.cp`) należy sięgać bezpośrednio, czy też posługując się indeksami z tablicy `CPIndices.cpi`. W zmiennej `Colour` jest podany kolor płatów.

[11] Szader nie odwołuje się do niego, ale później (w p. 15.8.2) go zmodyfikujemy i będzie to robił.

Informacja z pola `TessLevel` jest używana przez opisany wcześniej szader sterowania rozdrabnianiem. Pole `BezNormals` służy do wybierania wektorów normalnych w wierzchołkach trójkątów przesyłanych do etapu obcinania. Jeśli to pole ma wartość `true`, to wektory te są obliczonymi przez szader rozdrabniania wektorami normalnymi płata Béziera, dzięki czemu powstaje obraz gładkiej powierzchni zakrzywionej. Wartość `false` powoduje przyjęcie wektora normalnego płaszczyzny trójkąta, czego skutkiem będzie obraz pokazujący kawałkami płaskie przybliżenie powierzchni.

Listing 15.4. Szader rozdrabniania — używane zmienne

GLSL

```
1: #version  430 core
2:
3: #define MAX_DEG 10
4:
5: layout(quads,equal_spacing,ccw) in;
6:
7: layout(location=0) in int instance[];
8:
9: out GVertex {
10:     vec3 Colour;
11:     vec3 Position;
12:     vec3 Normal;
13:   } Out;
14:
15: layout(std430,binding=0) buffer CPoints {
16:     float cp[];
17:   } cp;
18:
19: layout(std430,binding=1) buffer CPIndices {
20:     int cpi[];
21:   } cpi;
22:
23: layout(std430,binding=2) buffer BezPatch { .... } bezp; /* listing 15.3 */
24:
25: uniform TransBlock {
26:     mat4 mm, mmti, vm, pm, vpm;
27:     vec4 eyepos;
28:   } trb;
```

Blok `TransBlock` w liniach 25–28, dobrze znany z pierwszej aplikacji, zawiera macierze przekształceń potrzebnych do rzutowania punktu i położenie obserwatora w układzie świata.

W liniach 29–75 są podane dwie procedury realizujące algorytm obliczania punktu i wektora normalnego płata płaskiego, zawartego w płaszczyźnie xy. Procedura `BCHorner2f` oblicza punkt położony na krzywej stopnia n. Jej punkty kontrolne są podane w tablicy `bcp`. Parametr `t` podaje wartość zadanego argumentu parametryzacji t. Obliczony punkt krzywej ma być przypisany do parametru wyjściowego `p`.

Algorytm zrealizowany w procedurze `BCHorner2f` jest schematem Hornera przedstawionym wcześniej; wartości kolejno przypisywane zmiennej `b` to współczynniki dwumianowe, a zmiennej `d` są przypisywane kolejne potęgi parametru t.

Procedura `BPHorner2f` (linie 44–75) otrzymuje jako parametry `u`, `v` współrzędne u, v punktu w dziedzinie płata. Parametry wyjściowe `pos` i `nv` są zmiennymi, do których należy przypisać obliczony punkt i wektor normalny płata. Zauważmy, że w przypadku płata płaskiego wektor normalny jest stały i do jego obliczenia pochodne parametryzacji nie są potrzebne.

W pętli w liniach 57–71 obliczane są punkty $\boldsymbol{q}_0, \ldots, \boldsymbol{q}_n$ krzywych Béziera reprezentowanych przez kolumny siatki kontrolnej, dla podanego argumentu parametryzacji v. Wywołanie procedury `BCHorner2f` w linii 72 oblicza punkt krzywej Béziera reprezentowanej przez te punkty, dla podanego parametru u. Do obliczonych w ten sposób współrzędnych x, y w linii 73 doczepiane są współrzędne $z = 0$ i $w = 1$, a w linii 74 jest (trywialne) wygenerowanie współrzędnych wektora normalnego płata.

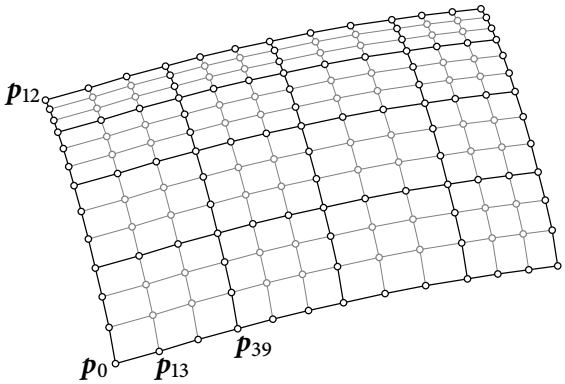
Zwróćmy uwagę na pobieranie punktów kontrolnych z tablicy. Jest to robione jednym z dwóch sposobów. Pierwszy sposób (linie 59–62) polega na korzystaniu z tablicy indeksów. Kolejne elementy tej tablicy, zaczynając od pozycji i_0, będącej iloczynem numeru instancji i wartości zmiennej jednolitej `bezp.stride_p`, numerują punkty w kolejnych $n+1$ kolumnach siatki kontrolnej; każda kolumna składa się z $m+1$ punktów. Indeksy są w linii 60 mnożone przez 2, bo każdy punkt kontrolny płata płaskiego składa się z dwóch współrzędnych.

Sposób drugi wymaga założenia, że tablica punktów kontrolnych zawiera kolejne kolumny prostokątnej macierzy, przy czym wymiary tej macierzy (liczba kolumn i wierszy) mogą być dowolnie duże. Pierwsza współrzędna pierwszego punktu siatki kontrolnej płata Béziera jest na pozycji i_0, obliczanej w liniach 53–55. Przyjęte tu jest założenie, że każda kolumna macierzy zawiera kolumny należące do siatek n_q płatów Béziera (ta liczba jest wartością zmiennej `bezp.nq`). Początki kolumn w siatkach kolejnych *płatów* występują co `bezp.stride_q` miejsc w tablicy, natomiast początki kolejnych kolumn *tablicy* dzieli odległość `bezp.stride_p` miejsc[12].

Przejście do kolejnego punktu w kolumnie siatki kontrolnej wymaga dodania przyrostu `bezp.stride_v`, a przejście do kolejnego punktu w wierszu odbywa się z krokiem `bezp.stride_u`. Jeśli w tablicy mamy tylko upakowane punkty kontrolne jednego płata stopnia (n, m), to wartość zmiennej `bezp.stride_v` powinna być równa 2 (bo płat jest płaski), a zmienna `bezp.stride_u` powinna mieć wartość $2(m+1)$.

Bardziej skomplikowany przykład jest pokazany na rysunku 15.4. Widoczna na nim siatka składa się z siatek reprezentujących 16 płatów Béziera stopnia $(n, m) = (3, 3)$, przy czym siatki sąsiednich płatów mają wspólne brzegi (tj. czwórki punktów kontrolnych), zaznaczone na czarno. W tabeli są pokazane kroki, z którymi należy się poruszać po tej tablicy punktów kontrolnych; zależą one od liczby d współrzędnych tych punktów.

[12] Taką tablicę z siatkami kontrolnymi wielu płatów można otrzymać na podstawie siatki kontrolnej tensorowego płata B-sklejanego. W grafice, a zwłaszcza w zastosowaniach CAD takie płaty są używane często.



$n_p = 4, n_q = 4$

d	2	3	4
stride_p	78	117	156
stride_q	6	9	12
stride_u	26	39	52
stride_v	2	3	4

Rysunek 15.4. Dostęp do siatek kontrolnych płatów Béziera w prostokątnej tablicy

Listing 15.4. (cd.) Szader rozdrabniania — obliczanie punktu i wektora normalnego płata płaskiego

GLSL

```
29: void BCHorner2f ( int n, vec2 bcp[MAX_DEG+1], float t, out vec2 p )
30: {
31:   int   i, b;
32:   float s, d;
33:   vec2  q;
34:
35:   s = 1.0-t;  d = t;  b = n;
36:   q = bcp[0];
37:   for ( i = 1; i <= n; i++ ) {
38:     q = s*q + (b*d)*bcp[i];
39:     d *= t;  b = (b*(n-i))/(i+1);
40:   }
41:   p = q;
42: } /*BCHorner2f*/
43:
44: void BPHorner2f ( float u, float v, out vec4 pos, out vec3 nv )
45: {
46:   vec2 p[MAX_DEG+1], q[MAX_DEG+1], r;
47:   vec4 Pos, Normal;
48:   int  i, j, k, l, i0;
49:
50:   if ( bezp.use_ind )
51:     i0 = instance[0]*bezp.stride_p;
52:   else {
53:     i = instance[0] / bezp.nq;
54:     j = instance[0] % bezp.nq;
55:     i0 = i*bezp.stride_p + j*bezp.stride_q;
56:   }
57:   for ( i = k = 0;  i <= bezp.udeg;  i++ ) {
58:     if ( bezp.use_ind ) {
59:       for ( j = 0;  j <= bezp.vdeg;  j++, k++ ) {
```

```
60:         l = 2*cpi.cpi[i0+k];
61:         p[j] = vec2 ( cp.cp[l], cp.cp[l+1] );
62:       }
63:     }
64:     else {
65:       for ( j = 0, l = i0+i*bezp.stride_u;
66:             j <= bezp.vdeg;
67:             j++, l += bezp.stride_v )
68:         p[j] = vec2 ( cp.cp[l], cp.cp[l+1] );
69:     }
70:     BCHorner2f ( bezp.vdeg, p, v, q[i] );
71:   }
72:   BCHorner2f ( bezp.udeg, q, u, r );
73:   pos = vec4 ( r.xy, 0.0, 1.0 );
74:   nv = vec3 ( 0.0, 0.0, 1.0 );
75: } /*BPHorner2f*/
```

Procedury realizujące obliczanie punktu i wektora normalnego płata trójwymiarowego są przedstawione na listingu 15.4 w liniach 76–111. W przypadku trój- i czterowymiarowym wektor normalny trzeba obliczyć przy użyciu pochodnych parametryzacji. Dlatego pierwsza procedura oblicza punkt i wektor pochodnej krzywej Béziera. Druga procedura posługuje się pierwszą w celu obliczenia punktu i pochodnych cząstkowych płata. Zgodnie z wcześniej podanym opisem teoretycznych własności krzywych procedura `BCHorner3f` oblicza najpierw punkty $\boldsymbol{r}_0$ i $\boldsymbol{r}_1$ dwóch krzywych stopnia $n-1$. Aby otrzymać współrzędne jednorodne, po obliczeniu punktu płata w $\mathbb{R}^3$ doczepiamy współrzędną wagową 1.

Listing 15.4. (cd.) Szader rozdrabniania — obliczanie punktu i wektora normalnego płata w $\mathbb{R}^3$

GLSL

```
76: void BCHorner3f ( int n, vec3 bcp[MAX_DEG+1], float t,
77:                   out vec3 p, out vec3 dp )
78: {
79:   int   i, b;
80:   float s, d, bd;
81:   vec3  r0, r1;
82:
83:   n --;    /* przetwarzamy dwie krzywe stopnia n-1 */
84:   s = 1.0-t;  d = t;  b = n;
85:   r0 = bcp[0];  r1 = bcp[1];
86:   for ( i = 1; i <= n; i++ ) {
87:     bd = b*d;
88:     r0 = s*r0 + bd*bcp[i];
89:     r1 = s*r1 + bd*bcp[i+1];
90:     d *= t;  b = (b*(n-i))/(i+1);
91:   }
92:   p = s*r0 + t*r1;
93:   dp = r1 - r0;  /* mnożenie różnicy przez stopień jest niepotrzebne */
94: } /*BCHorner3f*/
```

```
95:
96: void BPHorner3f ( float u, float v, out vec4 pos, out vec3 nv )
97: {
98:   vec3 p[MAX_DEG+1], q0[MAX_DEG+1], q1[MAX_DEG+1], r, ru, rv, ruv;
99:   int  i, j, k, l, i0;
100:
101:   .... /* obliczenie indeksu i0 tak jak w procedurze BPHorner2f */
102:   for ( i = k = 0;  i <= bezp.udeg;  i++ ) {
103:       .... /* tu instrukcje są takie, jak w procedurze BPHorner2f */
104:       .... /* tylko punkty mają 3, a nie 2 współrzędne */
105:     BCHorner3f ( bezp.vdeg, p, v, q0[i], q1[i] );
106:   }
107:   BCHorner3f ( bezp.udeg, q0, u, r, ru );
108:   BCHorner3f ( bezp.udeg, q1, u, rv, ruv );
109:   pos = vec4 ( r, 1.0 );
110:   nv = cross ( ru, rv );
111: } /*BPHorner3f*/
```

Kolejna część listingu 15.4 przedstawia procedury obliczania punktu i wektora normalnego wymiernego płata Béziera. Zasadnicza zmiana w porównaniu z poprzednim przypadkiem polega na zwiększeniu liczby współrzędnych każdego punktu do 4. Ponadto zamiast wbudowanej (standardowej) funkcji `cross`, obliczającej iloczyn wektorowy w $\mathbb{R}^3$, używamy przedstawionej na listingu 15.1 funkcji `cross4` obliczającej iloczyn trzech wektorów w $\mathbb{R}^4$.

Listing 15.4. (cd.) Szader rozdrabniania — obliczanie punktu i wektora normalnego płata wymiernego

GLSL

```
112:   .... /* tu procedura cross4 z listingu 15.1 */
113:
114: void BCHorner4f ( int n, vec4 bcp[MAX_DEG+1], float t,
115:                   out vec4 p, out vec4 dp )
116: {
117:   .... /* ta procedura jest kopią BCHorner3f, */
118:   .... /* z typem vec3 zamienionym wszędzie na vec4 */
119: } /*BCHorner4f*/
120:
121: void BPHorner4f ( float u, float v, out vec4 pos, out vec3 nv )
122: {
123:   vec4 p[MAX_DEG+1], q0[MAX_DEG+1], q1[MAX_DEG+1], ru, rv, ruv;
124:   int  i, j, k, l, i0;
125:
126:   .... /* obliczenie indeksu i0 tak jak w procedurze BPHorner2f */
127:   for ( i = k = 0;  i <= bezp.udeg;  i++ ) {
128:       .... /* tu instrukcje są takie, jak w procedurze BPHorner2f */
129:       .... /* tylko punkty mają 4, a nie 2 współrzędne */
130:     BCHorner4f ( bezp.vdeg, p, v, q0[i], q1[i] );
131:   }
```

```
132:   BCHorner4f ( bezp.udeg, q0, u, pos, ru );
133:   BCHorner4f ( bezp.udeg, q1, u, rv, ruv );
134:   nv = cross4 ( pos, rv, ru );
135:   pos /= pos.w;
136: } /*BPHorner4f*/
```

Jeszcze jeden szczegół wymaga wyjaśnienia: w linii 135 dzielimy wszystkie współrzędne wektora pos przez współrzędną wagową. Zmiana długości wektora współrzędnych jednorodnych bez zmiany jego kierunku daje tylko inną reprezentację tego samego punktu. Ale chcemy już w tym miejscu otrzymać wektor, którego pierwsze trzy współrzędne są współrzędnymi kartezjańskimi punktu powierzchni. Iloczyn tego wektora i macierzy przekształcenia modelu (której ostatni wiersz składa się z liczb $0, 0, 0, 1$) również zawiera współrzędne kartezjańskie punktu w układzie świata. Zostaną one zapamiętane w zmiennej Out.Position (linia 150) i użyte w alternatywnym obliczeniu wektora normalnego przez szader geometrii i w obliczeniu oświetlenia przez szader fragmentów.

Główna procedura szadera rozdrabniania, zależnie od liczby współrzędnych punktów kontrolnych, wywołuje jedną z trzech opisanych wcześniej procedur obliczających punkt płata i wektor normalny w tym punkcie. Współrzędne u, v punktu w dziedzinie płata, wygenerowane przez etap rozdrabniania dziedziny, szader otrzymuje jako wartości pól x i y zmiennej wbudowanej gl_TessCoord.

Listing 15.4. (cd.) Szader rozdrabniania — procedura main

GLSL

```
137: void main ( void )
138: {
139:   vec4 pos;
140:   vec3 nv;
141:
142:   pos = vec4 ( 0.0 );  /* to dla usunięcia ostrzeżenia */
143:   nv = vec3 ( 0.0 );
144:   switch ( bezp.dim ) {
145: case 2: BPHorner2f ( gl_TessCoord.x, gl_TessCoord.y, pos, nv );  break;
146: case 3: BPHorner3f ( gl_TessCoord.x, gl_TessCoord.y, pos, nv );  break;
147: case 4: BPHorner4f ( gl_TessCoord.x, gl_TessCoord.y, pos, nv );  break;
148:   }
149:   pos = trb.mm * pos;
150:   Out.Position = pos.xyz;
151:   gl_Position = trb.vpm*pos;
152:   if ( !bezp.BezNormals || dot ( nv, nv ) < 1.0e-10 )
153:     Out.Normal = vec3 ( 0.0 );
154:   else
155:     Out.Normal = normalize ( mat3(trb.mmti) * nv );
156:   Out.Colour = bezp.Colour;
157: } /*main*/
```

Po obliczeniu (jednorodnej reprezentacji) punktu i wektora normalnego płata następuje obliczenie w liniach 149–150 współrzędnych kartezjańskich punktu w układzie świata, a w linii 151 jest dokonywane przejście do układu kostki standardowej. W liniach 152–155 jest obliczany wektor normalny. Mamy tu pewien problem. Jeśli pochodne cząstkowe parametryzacji są liniowo zależne, to ich iloczyn wektorowy jest wektorem zerowym. W takim przypadku funkcja `normalize` zostaje zmuszona do dzielenia przez 0, a to jest karygodne[13]. Dlatego zamiast dzielić przez 0 wyprowadzamy wektor zerowy. Wtedy szader geometrii zamiast wektora normalnego powierzchni wyprowadzi wektor normalny płaszczyzny trójkąta, co skutecznie „załata" niepowodzenie obliczenia „prawdziwego" wektora normalnego powierzchni.

Listing 15.5. Procedura `main` szadera geometrii drugiej aplikacji

GLSL

```
1: void main ( void )
2: {
3:   int  i;
4:   vec3 v1, v2, tnv;
5:
6:   v1 = In[1].Position - In[0].Position;
7:   v2 = In[2].Position - In[0].Position;
8:   tnv = normalize ( cross ( v1, v2 ) );
9:   for ( i = 0; i < 3; i++ ) {
10:    gl_Position = gl_in[i].gl_Position;
11:    Out.Position = In[i].Position;
12:    if ( dot ( In[i].Normal, In[i].Normal ) < 1.0e-10 )
13:      Out.Normal = tnv;
14:    else
15:      Out.Normal = In[i].Normal;
16:    Out.TNormal = tnv;
17:    Out.Colour = In[i].Colour;
18:    EmitVertex ();
19:  }
20:  EndPrimitive ();
21: } /*main*/
```

Szader geometrii na listingu 15.5 powstał przez modyfikację szadera z listingu 12.7; niezmienione instrukcje są szare. W tej aplikacji użyjemy szadera fragmentów z listingu 12.8, realizującego lambertowski model oświetlenia.

15.4. Procedury wprowadzania i rysowania płatów Béziera

Teraz pora na procedury w C; zaczniemy od opisu ogólnych procedur tworzących i wyświetlających płaty Béziera, a aplikacja korzystająca z tych procedur będzie opisana dalej. Na listingu 15.6 jest pokazany fragment pliku nagłówkowego `bezpatches.h`, w którym jest

[13] Karą są plamy na obrazie i na reputacji autora programu.

ustalone ograniczenie stopnia płatów (takie samo jak na listingu 15.4)[14], definicja struktury reprezentującej zespół płatów Béziera do narysowania i nagłówki procedur przetwarzania płatów.

Listing 15.6. Struktura danych reprezentująca płaty Béziera

C

```
#define MAX_PATCH_DEGREE 10

typedef struct BezierPatchObjf {
    GLint  udeg, vdeg, dim,
           stride_u, stride_v, stride_p, stride_q, npatches;
    GLuint buf[5];
  } BezierPatchObjf;

void GetAccessToBezPatchStorageBlocks ( GLuint program_id,
                                        char txt, char tess );
BezierPatchObjf* EnterBezierPatches ( .... );      /* listing 15.8 */
BezierPatchObjf* EnterBezierPatchesElem ( .... );  /* listing 15.9 */

void SetBezierPatchTessLevel ( BezierPatchObjf *bp, GLint tesslevel );
void SetBezierPatchNVS ( BezierPatchObjf *bp, GLint nvs );
void SetBezierPatchColour ( BezierPatchObjf *bp, GLfloat *colour );
void BindBezierPatchBuffers ( BezierPatchObjf *bp );
void DrawBezierPatches ( BezierPatchObjf *bp );
void DeleteBezierPatches ( BezierPatchObjf *bp );
```

Pola `udeg`, `vdeg`, `dim`, `stride_u`, `stride_v`, `stride_p`, `stride_q` i `npatches` przechowują odpowiednio stopnie n i m płatów, wymiar (tj. liczbę współrzędnych) punktów kontrolnych, kroki dla procedur pobierania punktów kontrolnych przez szader rozdrabniania (opisane przy listingu 15.4) i liczbę płatów.

Tablica `buf` jest miejscem na przechowanie identyfikatorów SSBO — buforów, w których są umieszczone bloki magazynowe opisujące płaty. W pierwszym z tych buforów ma być blok `BezPatch`, w drugim blok `CPoints` z tablicą współrzędnych punktów kontrolnych, a w trzecim, jeśli ma być używany, blok `CPIndices`, czyli tablica indeksów do tablicy punktów kontrolnych. Czwarty bufor wykorzystamy do teksturowania w aplikacji 2D, a piąty posłuży do przechowania dobranych adaptacyjnie (w podrozdz. 23.5) parametrów rozdrabniania.

Listing 15.7 przedstawia procedurę odczytującą potrzebne informacje z programu szaderów do rysowania płatów Béziera, zmienne, w których te informacje zostają zapamiętane, oraz napisy — nazwy bloków magazynowych zawierających reprezentacje płatów Béziera. W tablicy `bezpbofs` procedura zapamiętuje przesunięcia pól bloku `BezPatch`; pozostałe

[14] Choć trudno o praktyczny powód do tego, można je zwiększyć, ale co najwyżej do 29. Zauważmy, że szader rozdrabniania, wywołując procedury obliczające punkty krzywych Béziera, kopiuje tablice punktów kontrolnych będące parametrami — kopiowanie długich tablic zabiera czas, zatem warto ich długość, zależną od *maksymalnego* stopnia, ograniczyć. Przeprowadziwszy eksperyment, nie zauważyłem jednak istotnej różnicy w szybkości rysowania (mierzonej liczbą klatek na sekundę) dla maksymalnych stopni ustalonych na 5 i 10.

bloki zawierają tylko jedno pole, którego przesunięcie względem początku bloku jest równe 0. Ostatnie dwa parametry sterują odczytywaniem punktów dowiązania opcjonalnych bloków magazynowych, które w programach szaderów mogą być (i w wersji aplikacji opisanej w tym rozdziale są) nieobecne — próba ich odczytania spowodowałaby błąd.

Listing 15.7. Procedura `GetAccessToBezPatchStorageBlocks`

C

```
#define NBEZPATCHUOFFS 13

static GLuint bezpbbp = GL_INVALID_INDEX, cpbbp = GL_INVALID_INDEX,
              cpibbp = GL_INVALID_INDEX, txcbbp = GL_INVALID_INDEX,
              tesspbbp = GL_INVALID_INDEX;
static GLint  bezpbsize;
static GLint  bezpbofs[NBEZPATCHUOFFS];
static const GLchar *UCPNames[] = { "CPoints" };
static const GLchar *UCPINames[] = { "CPIndices" };
static const GLchar *UBezPatchNames[] =
  { "BezPatch", "BezPatch.npatches", "BezPatch.dim", "BezPatch.udeg",
    "BezPatch.vdeg", "BezPatch.stride_u", "BezPatch.stride_v",
    "BezPatch.stride_p", "BezPatch.stride_q", "BezPatch.nq",
    "BezPatch.use_ind", "BezPatch.Colour", "BezPatch.TessLevel",
    "BezPatch.BezNormal" };
static const GLchar *UTexCoordNames[] = { "BezPatchTexCoord" };
static const GLchar *UTessBlockNames[] = { "BezPatchTessParams" };

void GetAccessToBezPatchStorageBlocks ( GLuint program_id,
                                        char txt, char tess )
{
  GLint size, ofs;

  if ( cpbbp == GL_INVALID_INDEX )
    GetAccessToStorageBlock ( program_id, 0, &UCPNames[0],
                              &size, &ofs, &cpbbp );
  if ( cpibbp == GL_INVALID_INDEX )
    GetAccessToStorageBlock ( program_id, 0, &UCPINames[0],
                              &size, &ofs, &cpibbp );
  if ( bezpbbp == GL_INVALID_INDEX )
    GetAccessToStorageBlock ( program_id, NBEZPATCHUOFFS,
                  &UBezPatchNames[0], &bezpbsize, bezpbofs, &bezpbbp );
  if ( txt && txcbbp == GL_INVALID_INDEX )
    GetAccessToStorageBlock ( program_id, 0, &UTexCoordNames[0],
                              &size, &ofs, &txcbbp );
  if ( tess && tesspbbp == GL_INVALID_INDEX )
    GetAccessToStorageBlock ( program_id, 0, &UTessBlockNames[0],
                              &size, &ofs, &tesspbbp );
  ExitIfGLError ( "GetAccessToBezPatchStorageBlocks" );
} /*GetAccessToBezPatchStorageBlocks*/
```

W zmiennych bezpbbp, cpbbp i cpibbp są przechowywane numery punktów dowiązania bloków magazynowych[15], a wartość zmiennej bezpbsize jest wielkością (w bajtach) bufora potrzebnego do pomieszczenia bloku BezPatch. Wielkość pozostałych buforów zależy od stopnia i liczby płatów reprezentowanych przez zawartość tych buforów.

Procedura EnterBezierPatches na listingu 15.8 tworzy obiekt reprezentujący płaty Béziera, których wierzchołki są ustawione w prostokątną macierz; jeśli liczba płatów jest większa niż 1, to siatka kontrolna każdego z nich powstaje przez odrzucenie pewnych wierszy i kolumn tej macierzy. Miejsca w tablicy, z których wybierane są punkty kontrolne, są obliczane na podstawie parametrów stride_p, stride_q, stride_u, stride_v, nq i numeru instancji (czyli numeru płata w zbiorze reprezentowanym przez tworzony obiekt). Nie jest w tym przypadku używana tablica indeksów. Zero wpisane do bufora w liniach 40–41 reprezentuje wartość false pola BezPatch.use_ind, na podstawie którego szader wybiera sposób dostępu do tablicy punktów kontrolnych.

Sprawdzenie poprawności danych obejmuje zbadanie wymiaru, tj. liczby współrzędnych punktu kontrolnego, oraz stopnia płata ze względu na oba parametry; musi on być większy od zera. W liniach 17–20 procedura zapamiętuje potrzebne dane w strukturze typu BezierPatchObjectf, której adres poda jako wartość powrotną.

W liniach 21–43 procedura tworzy i wypełnia danymi bufor dla bloku BezPatch. W liniach 49–50 jest tworzony bufor dla bloku CPoints z tablicą punktów kontrolnych. Wielkość tego bufora jest dostosowana do liczby tych punktów i liczby współrzędnych każdego z nich.

Listing 15.8. Procedura EnterBezierPatches

C

```
1: BezierPatchObjf* EnterBezierPatches ( GLint udeg, GLint vdeg, GLint dim,
2:                       GLint np, GLint nq, int ncp, const GLfloat *cp,
3:                       GLint stride_p, GLint stride_q,
4:                       GLint stride_u, GLint stride_v,
5:                       const GLfloat *colour )
6: {
7:   BezierPatchObjf *bp;
8:   GLint           size, zero = 0, tl = 10;
9:
10:  if ( dim < 2 || dim > 4 || npatches < 1 ||
11:       udeg < 1 || udeg > MAX_PATCH_DEGREE ||
12:       vdeg < 1 || vdeg > MAX_PATCH_DEGREE )
13:    return NULL;
14:  bp = malloc ( sizeof(BezierPatchObjf) );
15:  if ( bp ) {
16:    memset ( bp, 0, sizeof(BezierPatchObjf) );
17:    bp->udeg = udeg;  bp->vdeg = vdeg;  bp->dim = dim;
```

[15]Numery te są jawnie podane w deklaracjach bloków na listingu 15.4, bo, inaczej niż w przypadku bloków zmiennych jednolitych, aplikacja w C nie może zmienić przydziału punktów dowiązania. Aplikacja będzie rozbudowana tak, aby płaty Béziera były przetwarzane przez inne programy szaderów. Szadery w tych innych programach muszą mieć bloki zadeklarowane z tymi samymi numerami punktów dowiązania.

```
18:     bp->npatches = np*nq;
19:     bp->stride_p = stride_p;  bp->stride_q = stride_q;
20:     bp->stride_u = stride_u;  bp->stride_v = stride_v;
21:     bp->buf[0] = NewStorageBuffer ( bezpbsize, bezpbbp );
22:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[0],
23:                       sizeof(GLint), &bp->npatches );
24:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[1],
25:                       sizeof(GLint), &bp->dim );
26:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[2],
27:                       sizeof(GLint), &bp->udeg );
28:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[3],
29:                       sizeof(GLint), &bp->vdeg );
30:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[4],
31:                       sizeof(GLint), &bp->stride_u );
32:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[5],
33:                       sizeof(GLint), &bp->stride_v );
34:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[6],
35:                       sizeof(GLint), &bp->stride_p );
36:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[7],
37:                       sizeof(GLint), &bp->stride_q );
38:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[8],
39:                       sizeof(GLint), &nq );
40:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[9],
41:                       sizeof(GLint), &zero );
42:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[10],
43:                       3*sizeof(GLfloat), colour );
44:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[11],
45:                       sizeof(GLint), &tl );
46:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[12],
47:                       sizeof(GLint), &one );
48:     size = ncp*dim*sizeof(GLfloat);
49:     bp->buf[1] = NewStorageBuffer ( size, cpbbp );
50:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, 0, size, cp );
51:     ExitIfGLError ( "EnterBezierPatch" );
52:     ConstructEmptyVAO ();
53:   }
54:   return bp;
55: } /*EnterBezierPatches*/
```

Procedura `EnterBezierPatchesElem` na listingu 15.9 tworzy obiekt reprezentujący płaty Béziera, które mogą mieć wspólne lub powtarzające się punkty kontrolne[16]. Dlatego tworzone są trzy bufory — trzeci bufor zawiera tablicę indeksów do tablicy punktów kontrolnych. W liniach 30–31 zmiennej `BezPatch.use_ind` przypisywana jest wartość `true` (czyli 1), aby szader używał tablicy indeksów w odwołaniach do tablicy punktów kontrolnych. Zmienna jednolita `BezPatch.stride_p` w linii 27 otrzymuje wartość $(m+1)(n+1)$, bo

[16] Właśnie takie dane reprezentują płaty, z których składa się wyświetlany przez drugą aplikację czajnik.

tyle punktów kontrolnych ma siatka każdego płata stopnia (n, m). Zmienne `BezPatch.nq`, `BezPatch.stride_q`, `BezPatch.stride_u` i `BezPatch.stride_v` nie są używane w tym przypadku i dlatego procedura nie przypisuje im żadnych wartości.

Listing 15.9. Procedura `EnterBezierPatchesElem`

C

```
BezierPatchObjf* EnterBezierPatchesElem ( GLint udeg, GLint vdeg, GLint dim,
                      int npatches, int ncp,
                      const GLfloat *cp, const GLint *ind,
                      const GLfloat *colour )
{
  BezierPatchObjf *bp;
  GLint           size, one = 1, tl = 10;

  if ( dim < 2 || dim > 4 || npatches < 1 ||
       udeg < 1 || udeg > MAX_PATCH_DEGREE ||
       vdeg < 1 || vdeg > MAX_PATCH_DEGREE )
    return NULL;
  bp = malloc ( sizeof(BezierPatchObjf) );
  if ( bp ) {
    memset ( bp, 0, sizeof(BezierPatchObjf) );
    bp->udeg = udeg;  bp->vdeg = vdeg;  bp->dim = dim;
    bp->npatches = npatches;
    bp->buf[0] = NewStorageBuffer ( bezpbsize, bezpbbp );
    glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[0],
                      sizeof(GLint), &bp->npatches );
    glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[1],
                      sizeof(GLint), &bp->dim );
    glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[2],
                      sizeof(GLint), &bp->udeg );
    glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[3],
                      sizeof(GLint), &bp->vdeg );
    bp->stride_p = (udeg+1)*(vdeg+1);
    glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[6],
                      sizeof(GLint), &bp->stride_p );
    glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[7], sizeof(GLint),
                      &one );
    glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[9], sizeof(GLint),
                      &one );
    glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[10],
                      3*sizeof(GLfloat), colour );
    glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[11], sizeof(GLint),
                      &tl );
    glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[12], sizeof(GLint),
                      &one );
    size = ncp*dim*sizeof(GLfloat);
    bp->buf[1] = NewStorageBuffer ( size, cpbbp );
```

```
42:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, 0, size, cp );
43:     size = (udeg+1)*(vdeg+1)*npatches*sizeof(GLint);
44:     bp->buf[2] = NewStorageBuffer ( size, cpibbp );
45:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER, 0, size, ind );
46:     ExitIfGLError ( "EnterBezierPatchesElem" );
47:     ConstructEmptyVAO ();
48:   }
49:   return bp;
50: } /*EnterBezierPatchesElem*/
```

Procedury wprowadzania płatów Béziera wywołują procedurę `ConstructEmptyVAO` z listingu 11.9, której drugie i następne wywołania nie powodują żadnych skutków — pusty obiekt tablicy wierzchołków jest tworzony tylko raz. Podczas rysowania przy użyciu tego VAO do potoku trafi wierzchołek, którego wszelkie atrybuty, w szczególności położenie wzięte ze zmiennej statycznej (zobacz p. 7.2.2) o nieokreślonej wartości, zostaną zignorowane, więc nie musi mieć żadnych. W etapach pobierania wierzchołków i rozdrabniania dziedziny płata wierzchołek ten zostanie powielony: szader wierzchołków, wywołany w odpowiedniej liczbie instancji, spowoduje powstanie właśnie takiej liczby płatów, a szader rozdrabniania dla każdego płata obliczy tyle wierzchołków, ile punktów w dziedzinie płata wyprodukuje etap rozdrabniania dziedziny. Wszystkie atrybuty (położenie, kolor i wektor normalny) nada poszczególnym wierzchołkom szader rozdrabniania i w ten sposób powstaną wierzchołki trójkątów położone na płatach Béziera.

Listing 15.10 przedstawia procedury, których zadaniem jest przypisywanie wartości, podanych przez parametry, polom `TessLevel`, `BezNormals` i `Colour` w bloku magazynowym `BezPatch`[17]. Domyślne wartości, 10 i `true` oraz kolor podany jako parametr, nadała tym polom procedura `EnterBezierPatchs` lub `EnterBezierPatchesElem`, a ta procedura może być używana do późniejszego ich zmieniania.

Listing 15.10. Procedury przypisujące płatom parametry pomocnicze

C

```
1: void SetBezierPatchTessLevel ( BezierPatchObjf *bp, GLint tesslevel )
2: {
3:   glBindBuffer ( GL_SHADER_STORAGE_BUFFER, bp->buf[0] );
4:   glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[11],
5:                     sizeof(GLint), &tesslevel );
6:   ExitIfGLError ( "SetBezierPatchTessLvel" );
7: } /*SetBezierPatchTessLevel*/
8:
9: void SetBezierPatchNVS ( BezierPatchObjf *bp, GLint nvs )
10: {
11:   glBindBuffer ( GL_SHADER_STORAGE_BUFFER, bp->buf[0] );
12:   glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[12],
```

[17] W pierwszym wydaniu polom `TessLevel` i `BezNormals` wartości nadawała jedna procedura. Ale lepiej jest mieć osobną procedurę dla każdego parametru.

```
13:                     sizeof(GLint), &nvs );
14:   ExitIfGLError ( "SetBezierPatchNVS" );
15: } /*SetBezierPatchNVS*/
16:
17: void SetBezierPatchColour ( BezierPatchObjf *bp, GLfloat *colour )
18: {
19:   glBindBuffer ( GL_SHADER_STORAGE_BUFFER, bp->buf[0] );
20:   glBufferSubData ( GL_SHADER_STORAGE_BUFFER, bezpbofs[10],
21:                     3*sizeof(GLfloat), colour );
22:   ExitIfGLError ( "SetBezierPatchColour" );
23: } /*SetBezierPatchColour*/
```

Aby narysować płaty, należy wywołać procedurę `DrawBezierPatches` z listingu 15.11. Wywołuje ona pomocniczą procedurę `BindBezierPatchBuffers`, która przywiązuje bufory z danymi opisującymi płaty Béziera do odpowiednich punktów dowiązania w celu `GL_SHADER_STORAGE_BUFFER`. Następnie w linii 20 do kontekstu OpenGL-a przywiązywany jest pusty VAO (w zasadzie to może być absolutnie dowolny VAO; jest on potrzebny, aby umożliwić rysowanie, ale wszelkie dane opisujące płaty pochodzą z bloków magazynowych). W linii 21 określamy, że płat ma jeden wierzchołek; jego jedynym zadaniem jest uruchomienie potoku przetwarzania grafiki przez procedurę `glDrawArraysInstanced` w linii 22. Liczba instancji (czyli płatów Béziera, które zostaną narysowane) jest ostatnim parametrem tej procedury.

Listing 15.11. Procedury rysowania i likwidacji płatów Béziera

C

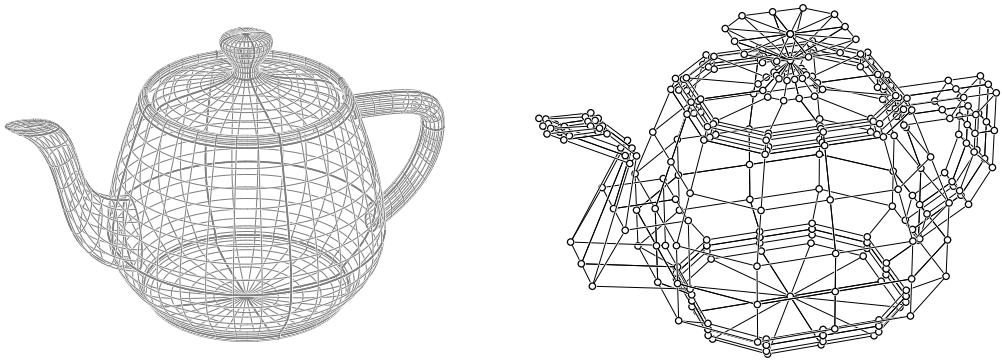
```
1: void BindBezierPatchBuffers ( BezierPatchObjf *bp )
2: {
3:   if ( bp ) {
4:     glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, bezpbbp, bp->buf[0] );
5:     glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, cpbbp, bp->buf[1] );
6:     if ( bp->buf[2] )
7:       glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, cpibbp, bp->buf[2] );
8:     if ( bp->buf[3] )
9:       glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, txcbbp, bp->buf[3] );
10:    if ( bp->buf[4] )
11:      glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, tesspbbp, bp->buf[4] );
12:    ExitIfGLError ( "BindBezierPatchBuffers" );
13:  }
14: } /*BindBezierPatchBuffers*/
15:
16: void DrawBezierPatches ( BezierPatchObjf *bp )
17: {
18:   if ( bp ) {
19:     BindBezierPatchBuffers ( bp );
20:     glBindVertexArray ( empty_vao );
21:     glPatchParameteri ( GL_PATCH_VERTICES, 1 );
```

```
22:     glDrawArraysInstanced ( GL_PATCHES, 0, 1, bp->npatches );
23:     glBindVertexArray ( 0 );
24:     ExitIfGLError ( "DrawBezierPatches" );
25:   }
26: } /*DrawBezierPatches*/
27:
28: void DeleteBezierPatches ( BezierPatchObjf *bp )
29: {
30:   if ( bp ) {
31:     glDeleteBuffers (  5, bp->buf );
32:     free ( bp );
33:   }
34: } /*DeleteBezierPatches*/
```

Procedura `DeleteBezierPatches` likwiduje bufory w pamięci GPU i zwalnia pamięć zajmowaną przez strukturę typu `BezierPatchObjf`. W tablicy `bp->buf` są dwa, trzy albo cztery[18] identyfikatory buforów; jeśli są tylko dwa, to `buf->bp[2] == buf->bp[3] == 0`. Liczba `0` nigdy nie jest identyfikatorem bufora (ani żadnego innego obiektu w OpenGL-u). Procedura `glDeleteBuffers` ignoruje taką liczbę, zamiast zwalniać nieistniejący bufor.

15.5. Czajnik z Utah

Nasza aplikacja rysuje słynny **czajnik z Utah** [59]; stworzył go w 1975 r. na podstawie porcelanowego pierwowzoru Martin Newell. Warto wiedzieć, że początkowo czajnik składał się z 28 płatów Béziera i nie miał dna, które zostało dorobione później[19]. James Blinn przeskalował oryginalny model w kierunku osi z, „spłaszczając" go (tj. zmniejszając wysokość)



Rysunek 15.5. Czajnik i jego siatki kontrolne

o 1/4 i w tej wersji czajnik jest najbardziej znany z licznych portretów i przedstawień. Dane opisujące model wziąłem z książki [27].

[18] w aplikacji 2D i dalszych

[19] Dlatego dane opisujące dno czajnika są podane na końcu tablic, po punktach kontrolnych uchwytu, dziobka i pokrywki.

Na listingu 15.12 jest podany fragment procedury, która umieszcza w pamięci GPU opisaną wyżej reprezentację 32 bikubicznych płatów Béziera składających się na model czajnika. Płaty bikubiczne (stopnia $(3,3)$) mają po 16 punktów kontrolnych, choć wiele z tych punktów pokrywa się z innymi (zobacz np. linie 12, 13 i 15 na listingu). Jest zatem tylko 306 różnych punktów kontrolnych[20], ale tablica indeksów ma $32 \cdot 16 = 512$ elementów.

Listing 15.12. Procedura wprowadzająca czajnik z Utah

C

```
1: #define TEAPOT_NPATCHES  32
2: #define TEAPOT_NPOINTS  306
3:
4: BezierPatchObjf* ConstructTheTeapot ( const GLfloat *colour )
5: {
6:   static GLfloat teapotcp[ TEAPOT_NPOINTS][3] =
7:    {{ 1.40000,  0.00000,  2.40000}, { 1.40000, -0.78400, 2.40000},
8:     { 0.78400, -1.40000,  2.40000}, { 0.00000, -1.40000, 2.40000},
9:     .... /* tu jest jeszcze 300 punktów */
10:    { 0.79800, -1.42500,  0.00000}, { 1.42500, -0.79800, 0.00000}};
11:  static GLint teapotcn[ TEAPOT_NPATCHES][16] =
12:   {{  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15},
13:    {  3, 16, 17, 18,  7, 19, 20, 21, 11, 22, 23, 24, 15, 25, 26, 27},
14:    .... /* tu jest jeszcze 29 wierszy, po 16 liczb w każdym */
15:    {269,269,269,269,299,304,305,278,296,302,303,274, 95, 94, 93, 92}};
16:
17:  return EnterBezierPatchesElem ( 3, 3, 3,  TEAPOT_NPATCHES, TEAPOT_NPOINTS,
18:                           &teapotcp[0][0], &teapotcn[0][0], colour );
19: } /*ConstructTheTeapot*/
```

15.6. Druga aplikacja — część graficzna

Listing 15.13 przedstawia strukturę, która jest opakowaniem wszystkich zmiennych używanych przez część graficzną aplikacji. Wszystkie zmienne związane z rzutowaniem perspektywicznym, czyli wymiary okna, parametry ostrosłupa widzenia oraz reprezentacja przejścia od układu współrzędnych świata do układu obserwatora, są polami struktury typu `Camera`. Pozostałe zmienne przechowują reprezentację czajnika (`myteapot`), bloki z opisami przekształceń i źródeł światła (`TransBl`, `LightBl`), identyfikatory buforów w pamięci GPU z blokami zmiennych jednolitych `TransBlock` i `LSBlock`, wybrany sposób obliczania wektora normalnego płatów (`BezNormals`) i stopień ich rozdrobnienia (`TessLevel`), reprezentację przekształcenia modelu i identyfikator programu szaderów. Zmienna `appdata`, zadeklarowana statycznie, jest widoczna tylko dla procedur części graficznej aplikacji.

[20] Punkty o numerach 204, 205, 215, 222, 270, 271, 272, 273, 282, 283, 284, 291, 292, 293, 300 i 301 są nieużywane — można je usunąć z tablicy `teapotcp`, co pociąga konieczność zmodyfikowania indeksów w tablicy `teapotcn`. Ale ja tu hołubię dane oryginalne.

Listing 15.13. Opakowanie danych części graficznej

C

```
typedef struct Camera {
    int    win_width, win_height;
    float  left, right, bottom, top, near, far, rl, tb;
    float  viewer_pos0[4];
    float  viewer_rvec[3];
    double viewer_rangle;
  } Camera;

typedef struct {
    Camera          camera;
    BezierPatchObjf *myteapot;
    TransBl         trans;
    LightBl         light;
    GLint           BezNormals, TessLevel;
    float           model_rot_axis[3];
    double          model_rot_angle;
    GLuint          program_id;
  } AppData;

static AppData appdata;
```

Listing 15.14. Procedura `InitMyWorld`

C

```
void InitMyWorld ( int argc, char *argv[], int width, int height )
{
  float axis[4] = {0.0,0.0,1.0};

  memset ( &appdata, 0, sizeof(AppData) );
  LoadBPShaders ( &appdata.program_id );
  appdata.trans.trbuf = NewUniformTransBlock ();
  appdata.light.lsbuf = NewUniformLightBlock ();
  TimerInit ();
  SetupModelMatrix ( &appdata, axis, 0.0 );
  InitCamera ( &appdata, width, height );
  appdata.TessLevel = 10;
  appdata.BezNormals = GL_TRUE;
  ConstructMyTeapot ( &appdata );
  InitLights ( &appdata );
} /*InitMyWorld*/
```

Pierwsza procedura wywołana przez `InitMyWorld` kompiluje szadery i łączy je w program (listing 15.15), a potem uzyskuje dostęp do bloków zmiennych jednolitych z opisami przekształceń i źródeł światła oraz do bloków magazynowych z danymi opisującymi płaty.

Użyte do tego procedury są przedstawione na listingach 10.7, 10.9 i 15.7. Na końcu szadery są kasowane i program jest gotowy.

Listing 15.15. Procedura `LoadBPShaders`

```C
void LoadBPShaders ( GLuint *program_id )
{
  static const char *filename[] =
    { "app2.vert.glsl", "app2.tesc.glsl", "app2.tese.glsl",
      "app2.geom.glsl", "app2.frag.glsl" };
  static const GLuint shtype[6] =
    { GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER,
      GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER };
  GLuint shader_id[5], prog_id;
  int    i;

  for ( i = 0; i < 5; i++ )
    shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
  *program_id = prog_id = LinkShaderProgram ( 5, shader_id, NULL );
  GetAccessToTransBlockUniform ( prog_id );
  GetAccessToLightBlockUniform ( prog_id );
  GetAccessToBezPatchStorageBlocks ( prog_id, false, false );
  for ( i = 0; i < 5; i++ )
    glDeleteShader ( shader_id[i] );
  ExitIfGLError ( "LoadBPShaders" );
} /*LoadBPShaders*/
```

Pozostałe procedury inicjalizacji są pokazane na listingu 15.16. Procedura `SetupModelMatrix` konstruuje macierz przekształcenia modelu i zapamiętuje ją w polu `trans` i w buforze w pamięci GPU. Przekształcenie to jest złożeniem dwóch przekształceń: skalowania, które przywraca czajnikowi oryginalne proporcje, i obrotu wokół osi pionowej (początkowo o kąt 0). Procedura `InitCamera` nadaje wartości początkowe zmiennym w polu `camera` (w szczególności ustala położenie początkowe obserwatora), po czym wywołuje procedurę `_ResizeMyWorld`, której zadaniem jest skonstruowanie macierzy przejścia do układu kostki standardowej dla okna o podanych wymiarach.

Zadaniem procedury `ConstructMyTeapot` jest wywołanie procedury z listingu 15.12, aby powstała reprezentacja czajnika w odpowiednim kolorze[21], i nadanie wartości zmiennym określającym stopień rozdrobnienia płatów i wybór wektora normalnego. Procedura `InitLights` wprowadza opis jednego źródła światła; jest on identyczny z opisem w aplikacji 1B, ale pierwszy parametr wywoływanych procedur jest adresem pola `light` struktury opakowującej dane aplikacji.

[21] białej porcelany, z której został zrobiony oryginalny czajnik, przechowywany obecnie w Muzeum Historii Komputerów w Mountain View w Kalifornii

Listing 15.16. Procedury wywoływane podczas inicjalizacji

C

```
void SetupModelMatrix ( AppData *ad, float axis[3], float angle )
{
#define SCF 0.33
  GLfloat ms[16], mr[16], mt[16], ma[16];

  memcpy ( ad->model_rot_axis, axis, 3*sizeof(float) );
  ad->model_rot_angle = angle;
  M4x4Scalef ( ms, SCF, SCF, SCF*4.0/3.0 );
  M4x4Translatef ( mt, 0.0, 0.0, -0.6 );
  M4x4Multf ( ma, mt, ms );
  M4x4RotateVf ( mr, axis[0], axis[1], axis[2], angle );
  M4x4Multf ( ms, mr, ma );
  LoadMMatrix ( &ad->trans, ms );
} /*SetupModelMatrix*/

static void _ResizeMyWorld ( AppData *ad, int width, int height )
{
  float lr;

  glViewport ( 0, 0, ad->camera.win_width = width,
               ad->camera.win_height = height );
  lr = 0.5533*(float)width/(float)height;
  M4x4Frustumf ( ad->trans.pm, NULL,
                 ad->camera.left = -lr, ad->camera.right = lr,
                 ad->camera.bottom = -0.5533, ad->camera.top = 0.5533,
                 ad->camera.near = 5.0, ad->camera.far = 15.0 );
  ad->camera.rl = 2.0*lr;  ad->camera.tb = 2.0*0.5533;
  LoadVPMatrix ( &ad->trans );
} /*_ResizeMyWorld*/

void InitCamera ( AppData *ad, int width, int height )
{
  static const float viewer_rvec[3] = {1.0,0.0,0.0};
  static const float viewer_pos0[4] = {0.0,0.0,10.0,1.0};

  memcpy ( ad->camera.viewer_rvec, viewer_rvec, 3*sizeof(float) );
  memcpy ( ad->camera.viewer_pos0, viewer_pos0, 4*sizeof(float) );
  memcpy ( ad->trans.eyepos, viewer_pos0, 4*sizeof(GLfloat) );
  ad->camera.viewer_rangle = 0.0;
  M4x4InvTranslatefv ( ad->trans.vm, viewer_pos0 );
  _ResizeMyWorld ( ad, width, height );
} /*InitCamera*/

void ConstructMyTeapot ( AppData *ad )
{
```

```
46:   const GLfloat MyColour[4] = { 1.0, 1.0, 1.0, 1.0 };
47:
48:   ad->myteapot = ConstructTheTeapot ( MyColour );
49:   SetBezierPatchTessLevel ( ad->myteapot, ad->TessLevel );
50:   SetBezierPatchNVS ( ad->myteapot, ad->BezNormals );
51: } /*ConstructMyTeapot*/
52:
53: void InitLights ( AppData *ad )
54: {
55:   GLfloat amb0[3] = { 0.2, 0.2, 0.3 };
56:   GLfloat dir0[3] = { 0.8, 0.8, 0.8 };
57:   GLfloat pos0[4] = { 0.0, 1.0, 1.0, 0.0 };
58:   GLfloat atn0[3] = { 1.0, 0.0, 0.0 };
59:
60:   SetLightAmbient ( &ad->light, 0, amb0 );
61:   SetLightDirect ( &ad->light, 0, dir0 );
62:   SetLightPosition ( &ad->light, 0, pos0 );
63:   SetLightAttenuation ( &ad->light, 0, atn0 );
64:   SetLightOnOff ( &ad->light, 0, 1 );
65: } /*InitLights*/
```

Listing 15.17 przedstawia procedury wywoływane przez część okienkową aplikacji po zmianie wymiarów okna i po wydaniu polecenia obrócenia obserwatora. Ponieważ zmienna `appdata` jest dla części okienkowej niewidoczna, procedura `ResizeMyWorld` wywołuje procedurę `_ResizeMyWorld`, która wykonuje zasadniczą pracę (czyli oblicza macierz przejścia do układu kostki standardowej), mając dostęp do danych wskazywanych przez parametr.

Listing 15.17. Procedury interfejsu z częścią okienkową

C

```
1: void ResizeMyWorld ( int width, int height )
2: {
3:   _ResizeMyWorld ( &appdata, width, height );
4: } /*ResizeMyWorld*/
5:
6: static void _RotateViewer ( AppData *ad, double delta_xi, double delta_eta )
7: {
8:   float   vi[3], lgt, vk[3];
9:   double  angi, angk;
10:  GLfloat tm[16];
11:
12:  vi[0] = (float)delta_eta*ad->camera.rl/(float)ad->camera.win_height;
13:  vi[1] = (float)delta_xi*ad->camera.tb/(float)ad->camera.win_width;
14:  vi[2] = 0.0;
15:  lgt = sqrt ( V3DotProductf ( vi, vi ) );
16:  vi[0] /= lgt;  vi[1] /= lgt;
17:  angi = -0.052359878;  /* -3 stopnie */
18:  V3CompRotationsf ( vk, &angk, ad->camera.viewer_rvec,
```

```
                      ad->camera.viewer_rangle, vi, angi );
  memcpy ( ad->camera.viewer_rvec, vk, 3*sizeof(float) );
  ad->camera.viewer_rangle = angk;
  M4x4RotateVfv ( ad->trans.vm, ad->camera.viewer_rvec,
                  -ad->camera.viewer_rangle );
  M4x4MultMTVf ( ad->trans.eyepos, ad->trans.vm, ad->camera.viewer_pos0 );
  M4x4InvTranslateMfv ( ad->trans.vm, ad->camera.viewer_pos0 );
  LoadVPMatrix ( &ad->trans );
} /*_RotateViewer*/

void RotateViewer ( double delta_xi, double delta_eta )
{
  if ( delta_xi == 0.0 && delta_eta == 0.0 )
    return;  /* natychmiast uciekamy - nie chcemy dzielić przez 0 */
  _RotateViewer ( &appdata, delta_xi, delta_eta );
} /*RotateViewer*/
```

Podobnie procedura `RotateViewer`, wywoływana przez część okienkową, wywołuje procedurę `_RotateViewer`, dając jej dostęp do danych aplikacji. Składanie wykonanego wcześniej obrotu obserwatora wokół sceny z obrotem skonstruowanym na podstawie przesunięcia kursora w oknie jest wykonywane tak samo jak w aplikacji 1A; dodatkowo, tak jak w aplikacji 1C, procedura oblicza i zapamiętuje położenie obserwatora.

Listing 15.18 przedstawia procedurę rysowania, wywoływaną w odpowiednich chwilach przez część okienkową, oraz procedurę, której zadaniem jest wykonanie polecenia wydanego przez naciśnięcie klawisza. Wartość powrotna `true` zawiadamia część okienkową, że po wykonaniu polecenia obraz jest nieaktualny i należy wywołać procedurę rysowania. Wykonywane przez aplikację polecenia to zwiększenie i zmniejszenie stopnia rozdrobnienia płatów i przełączenie sposobu wybierania wektora normalnego do obliczeń oświetlenia.

Listing 15.18. Procedury rysowania i interpretowania wejścia z klawiatury

C

```
void RedrawMyWorld ( void )
{
  glClearColor ( 1.0, 1.0, 1.0, 1.0 );
  glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glEnable ( GL_DEPTH_TEST );
  glUseProgram ( appdata.program_id );
  DrawBezierPatches ( appdata.myteapot );
  glUseProgram ( 0 );
  glFlush ();
  ExitIfGLError ( "RedrawMyWorld" );
} /*RedrawMyWorld*/

char ProcessCharCommand ( char charcode )
{
  switch ( toupper ( charcode ) ) {
```

```
case '+':
    if ( appdata.TessLevel < MAX_TESS_LEVEL ) {
      SetBezierPatchTessLevel ( appdata.myteapot, ++appdata.TessLevel );
      return true;
    }
    else return false;
case '-':
    if ( appdata.TessLevel > MIN_TESS_LEVEL ) {
      SetBezierPatchTessLevel ( appdata.myteapot, --appdata.TessLevel );
      return true;
    }
    else return false;
case 'N':
    SetBezierPatchNVS ( appdata.myteapot,
                        appdata.BezNormals = appdata.BezNormals == 0 );
    return true;
default:
    return false;
  }
} /*ProcessCharCommand*/
```

Procedura animacji (listing 15.19), wywoływana co chwila w pętli komunikatów części okienkowej, od analogicznej procedury z pierwszej aplikacji różni się tylko miejscem, w którym jest zapamiętywany bieżący kąt obrotu modelu.

Listing 15.19. Procedura animacji

C

```
char MoveOn ( void )
{
  if ( (appdata.model_rot_angle += ANGULAR_VELOCITY * TimerTocTic ())
        >= PI )
    appdata.model_rot_angle -= 2.0*PI;
  SetupModelMatrix ( &appdata, appdata.model_rot_axis,
                     appdata.model_rot_angle );
  return true;
} /*MoveOn*/
```

Listing 15.20 przedstawia procedurę sprzątania, która nie wymaga objaśnień.

Listing 15.20. Procedura sprzątania

C

```
void DeleteMyWorld ( void )
{
  glUseProgram ( 0 );
  glDeleteProgram ( appdata.program_id );
  glDeleteBuffers ( 1, &appdata.trans.trbuf );
  glDeleteBuffers ( 1, &appdata.light.lsbuf );
  DeleteBezierPatches ( appdata.myteapot );
```

```
  DeleteEmptyVAO ();
} /*DeleteMyWorld*/
```

15.7. Druga aplikacja — część okienkowa

Podstawą drugiej aplikacji jest szkielet z listingu 3.2, przy czym główna pętla komunikatów jest realizowana przez procedurę `MessageLoop` z listingu 3.3, która umożliwia animację (np. „samoczynne" obracanie modelu).

Para procedur obsługujących komunikaty o zdarzeniach wygenerowanych przez mysz (naciśnięcie/zwolnienie przycisku i przesunięcie myszy), pokazanych na listingu 15.21, działa identycznie jak w pierwszej aplikacji; naciskając lewy przycisk i przesuwając mysz, powodujemy obracanie obserwatora wokół obiektu. Inne są tylko listy parametrów tych procedur oraz nazwy makrodefinicji identyfikujące przycisk i zdarzenie. Parametry procedury `MouseFunc` nie podają informacji o położeniu kursora w oknie. Dlatego informacja ta jest uzyskiwana przez wywołanie procedury `glfwGetCursorPos`.

Listing 15.21. Zmienne okienkowej części aplikacji i procedury obsługi komunikatów myszy

C

```
GLFWwindow  *mywindow;
double last_xi, last_eta;
int         app_state = STATE_NOTHING;
char        redraw;
char        animate = false;
static void (*idlefunc)(void) = NULL;
int         opti = 0;

void MouseFunc ( GLFWwindow *win, int button, int action, int mods )
{
  switch ( app_state ) {
case STATE_NOTHING:
    if ( button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS ) {
      glfwGetCursorPos ( win, &last_xi, &last_eta );
      app_state = STATE_TURNING;
    }
    break;
case STATE_TURNING:
    if ( button == GLFW_MOUSE_BUTTON_LEFT && action != GLFW_PRESS )
      app_state = STATE_NOTHING;
    break;
default:
    break;
  }
} /*MouseFunc*/

```

```
27: void MotionFunc ( GLFWwindow *win, double x, double y )
28: {
29:   switch ( app_state ) {
30: case STATE_TURNING:
31:     if ( x != last_xi || y != last_eta ) {
32:       RotateViewer ( x-last_xi, y-last_eta );
33:       last_xi = x,  last_eta = y;
34:       redraw = true;
35:     }
36:     break;
37: default:
38:     break;
39:   }
40: } /*MotionFunc*/
```

Listing 15.22. Procedury obsługi komunikatów klawiatury

C

```
1: void KeyFunc ( GLFWwindow *win, int key, int scancode,
2:                int action, int mode )
3: {
4:   if ( action == GLFW_PRESS || action == GLFW_REPEAT ) {
5:     switch ( key ) {
6:   case GLFW_KEY_ESCAPE:
7:       glfwSetWindowShouldClose ( mywindow, 1 );
8:       break;
9:   default:
10:      break;
11:    }
12:  }
13: } /*KeyFunc*/
14:
15: void CharFunc ( GLFWwindow *win, unsigned int charcode )
16: {
17:   switch ( charcode ) {
18: case ' ':
19:     ToggleAnimation ();
20:     break;
21: default:
22:     redraw = ProcessCharCommand ( charcode );
23:     break;
24:   }
25: } /*CharFunc*/
```

Komunikat o naciśnięciu klawisza `Esc` jest odbierany przez procedurę `KeyFunc` (listing 15.22), zarejestrowaną przez `glfwSetKeyCallback`, a procedura `CharFunc`, zarejestrowana przez `glfwSetCharCallback`, otrzymuje komunikaty o napisanych na klawiaturze

znakach. Naciśnięcie klawisza spacji powoduje włączenie albo wyłączenie animacji, tj. ruchu obrotowego czajnika, za pomocą procedury z listingu 15.23. Pozostałe znaki są przekazywane procedurze `ProcessCharCommand`, do której należy ich interpretacja.

Listing 15.23. Procedury animacji

C

```
void IdleFunc ( void )
{
  if ( MoveOn () )
    redraw = true;
} /*IdleFunc*/

void SetIdleFunc ( void(*IdleFunc)(void) )
{
  idlefunc = IdleFunc;
} /*SetIdleFunc*/

void ToggleAnimation ( void )
{
  if ( (animate = !animate) ) {
    TimerTic ();
    SetIdleFunc ( IdleFunc );
  }
  else
    SetIdleFunc ( NULL );
} /*ToggleAnimation*/
```

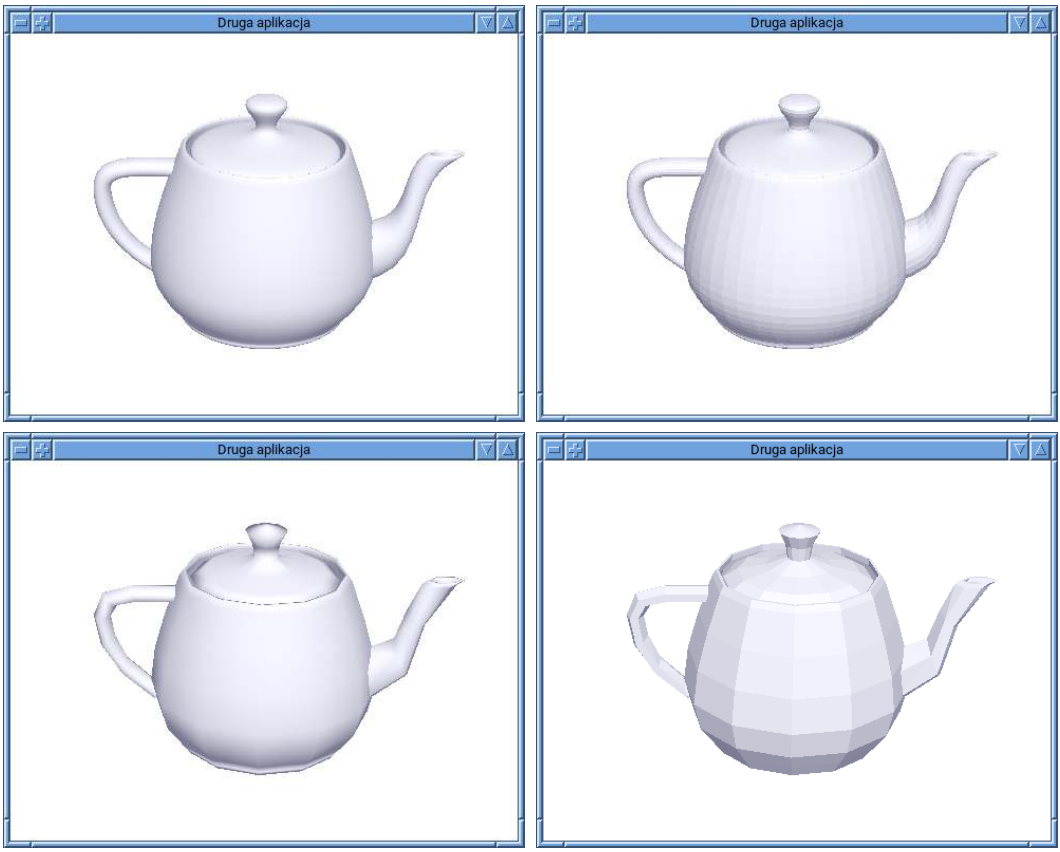
Rysunek 15.6 przedstawia okno aplikacji z czterema obrazami czajnika; w górnym wierszu parametry podziału płatów mają wartość 10, a w dolnym najmniejszą dopuszczalną, czyli 3. Obrazy z lewej strony zostały wykonane przy użyciu wektorów normalnych płatów obliczonych przez szader rozdrabniania[22]. Podczas wykonywania obrazów z prawej strony szader rozdrabniania zamiast obliczonych wektorów normalnych płatów Béziera wyprowadził wektor zerowy, w odpowiedzi na co szader geometrii obliczył wektory normalne trójkątów otrzymanych w wyniku rozdrabniania płatów, czego skutkiem jest obraz pokazujący płaskie fragmenty faktycznie narysowanej powierzchni przybliżającej oryginalne, gładkie płaty zakrzywione.

15.8. *Uzupełnienia

15.8.1. Położenia atrybutów wierzchołków

Przedstawione dotąd szadery wierzchołków otrzymują na wejściu atrybuty wierzchołków w „prostych” zmiennych typu `vec4`. Ale atrybuty mogą być też podawane w zmiennych typu

[22] Z wyjątkiem sytuacji, gdy obliczony został wektor zerowy i zadanie obliczania wektora normalnego przejął szader geometrii. Ma to miejsce w najwyższym punkcie pokrywki i w środku dna czajnika.



Rysunek 15.6. Okno drugiej aplikacji — różne obrazy czajnika

`int`, `float` lub `double`[23] oraz typów wektorowych lub macierzowych takich jak `ivec2` lub `mat3x4`. Przydzielając numery miejsc atrybutom, trzeba wiedzieć, że każdy atrybut typu `int`, `float` lub typu wektorowego o składowych tych dwóch typów zajmuje *jedno miejsce*. Każdy atrybut typu `double`, `dvec2`, `dvec3` lub `dvec4` zajmuje *dwa miejsca*. Atrybuty macierzowe zajmują tyle miejsc, ile zajęłyby ich kolumny podane osobno jako wektory, a zatem na przykład atrybut typu `mat3x2` zajmuje trzy miejsca.

Atrybut wierzchołka może być też tablicą zmiennych liczbowych, wektorowych lub macierzowych i wtedy należy pomnożyć liczbę miejsc zajmowanych przez jeden element przez długość tablicy. Choć lepiej tego unikać, zadeklarowane w treści szadera (w kwalifikatorach `layout(location=`*n*`)`) miejsca zajmowane przez atrybuty wejściowe wierzchołka mogą się nakładać. Miejsca zajmowane przez atrybuty wyjściowe *muszą* być rozłączne[24].

[23] Nie mogą być typu `bool`.

[24] Przydział miejsc dokonywany przez kompilator GLSL-a, na przykład dla pól w bloku wyjściowym, spełnia ten warunek. Autor szadera musi zapewnić jego spełnienie, jeśli sam deklaruje numery miejsc. Zewnętrzny kompilator `glslangValidator`, który przetwarza tekst w GLSL-u na kod SPIR-V, *wymaga* podania w tym tekście położeń atrybutów w blokach interfejsu między etapami potoku przetwarzania grafiki, zobacz p. 11.5.2.

Liczba miejsc, na których można przekazywać atrybuty wierzchołka, zależy od implementacji, przy czym specyfikacja OpenGL-a gwarantuje dostępność co najmniej 16 miejsc. Liczbę miejsc obecnych w danej implementacji można poznać, wywołujac procedurę `glGetIntegerv` z parametrem `GL_MAX_VERTEX_ATTRIBS`.

15.8.2. Atrybuty wierzchołków indywidualnych instancji

Dla każdej instancji rysowanego prymitywu (np. taśmy trójkątowej lub płata) etap pobierania wierzchołków podaje szaderowi wierzchołków *te same* wierzchołki, które mogą dalej być poddawane różnym przekształceniom lub „podmieniane" na inne wierzchołki, określone przez numer instancji — tak jak w aplikacji przedstawionej w tym rozdziale. Istnieje też możliwość zróżnicowania wierzchołków poszczególnych instancji w chwili wprowadzania ich do potoku, tj. już w etapie pobierania wierzchołków, choć w tym momencie każda instancja prymitywu musi mieć tyle samo wierzchołków[25]. Co więcej, pewne atrybuty wierzchołków mogą być inne dla każdej instancji, podczas gdy pozostałe atrybuty będą w każdej instancji takie same.

W tym celu, opisując w obiekcie tablicy wierzchołków miejsce, skąd ma być pobierany konkretny atrybut, można wywołać procedurę `glVertexAttribDivisor`. Procedura ta ma dwa parametry: numer miejsca atrybutu i związany z tym atrybutem dzielnik. Jeśli dzielnik ma domyślną wartość `0`, to i-ty wierzchołek wszystkich instancji prymitywu ma odpowiedni atrybut o tej samej wartości. Jeśli dzielnik ma wartość $k > 0$, to w każdej instancji prymitywu wszystkie wierzchołki mają ten atrybut taki sam. Wierzchołki pierwszych k instancji otrzymają go z pierwszego opisanego w VAO miejsca w odpowiednim buforze wierzchołków, kolejne k instancji z drugiego itd. Jeśli na przykład atrybut jest kolorem, to dla dzielnika `1` każda kolejna instancja może mieć inny kolor — ale w tablicy musi być co najmniej tyle kolorów, ile instancji zażyczyliśmy sobie narysować, wywołując procedurę `glDrawArraysInstanced` lub `glDrawElementsInstanced`. Jeśli dzielnik jest równy 2, to w każdym kolorze będą narysowane po dwa obiekty (i wtedy tablica kolorów może być dwa razy krótsza).

Podsumowując, jeśli prymityw jest opisany przez N wierzchołków i rysujemy jego M instancji, to dla każdego atrybutu mamy dwie możliwości: gdy dzielnik jest równy `0`, wierzchołek i-ty w j-tej instancji ma ten atrybut identyczny dla każdego $j = 0, \ldots, M-1$. Jeśli dzielnik nie jest zerem, to wszystkie wierzchołki w j-tej instancji mają ten atrybut taki sam.

Rysunek 15.7 przedstawia scenę, w której kilkanaście czajników (razem 416 płatów Béziera) zostało narysowanych jednym wywołaniem procedury `glDrawArraysInstanced`. Jego wykonanie wymagało wprowadzenia następujących zmian: procedura `main` szadera rozdrabniania oblicza iloraz i resztę z dzielenia numeru instancji (`instance[0]`) przez liczbę płatów podaną w zmiennej `bezp.npatches`; reszta z dzielenia jest używana zamiast numeru instancji w obliczeniach indeksów do tablicy punktów kontrolnych, natomiast iloraz jest numerem czajnika. Numer ten służy do wybrania macierzy przekształcenia modelu (nadającego czajnikowi odpowiednią wielkość i położenie w przestrzeni). Macierze przekształceń modelu i odwrotności ich transpozycji (potrzebne do przekształcania wektora normalnego)

[25] Szader sterowania rozdrabnianiem lub szader geometrii mogą jeszcze zmienić liczbę wierzchołków przetwarzanych dalej.

Rysunek 15.7. Wiele instancji płata w scenie

są podane w tablicy w osobnym buforze magazynowym. Wierzchołek jest najpierw przekształcany do układu świata przy użyciu macierzy przekształcenia modelu wybranej przy użyciu numeru instancji, a w dalszych przekształceniach (do układu kostki standardowej) są używane tylko macierze `vm` i `pm`, a dokładniej ich iloczyn `vpm` przechowywany w bloku `trb`. Kolor czajnika jest atrybutem wierzchołka podanym przez etap pobierania wierzchołków i dalej przekazywanym przez kolejne szadery aż do szadera fragmentów. Dzielnik dla tego atrybutu (określony za pomocą procedury `glVertexAttribDivisor`) jest liczbą płatów, z których składa się jeden czajnik.

Do kodu źródłowego w C trzeba było dodać instrukcje, które tworzą bufor z kolorami i rejestrują go w obiekcie tablicy wierzchołków, oraz bufor magazynowy z odpowiednimi macierzami przekształceń modelu. Macierze te trzeba oczywiście obliczać na podstawie odczytów zegara dla każdej klatki animacji, która w najprostszym przypadku powoduje krążenie małych czajników wokół dużego. Ponadto do procedury `DrawBezierPatches` dodałem parametr typu `int` o nazwie `copies`; liczba instancji podawana w wywołaniu procedury `glDrawArraysInstanced` jest iloczynem liczby płatów jednego czajnika i wartości tego parametru. Samodzielne wprowadzenie opisanych wyżej zmian i uruchomienie aplikacji polecam jako ćwiczenie.

Korzyści z rysowania wielu instancji obiektu za pomocą jednego wywołania procedury OpenGL-a są najbardziej widoczne, gdy tych instancji jest dużo, bo wtedy oszczędności czasu, który byłby zajmowany przez komunikację między CPU a GPU osobno dla każdego „egzemplarza" tego obiektu są bardzo duże. Typowe zastosowania to rysowanie trawy lub liści na drzewach, futra składającego się z wielu włosów albo pola asteroid orbitujących wokół planety. Tylko w ten sposób jest możliwe animowanie i rysowanie scen z takimi obiektami w czasie rzeczywistym.

16

Aplikacja druga A

Odrobinę rozszerzymy możliwości drugiej aplikacji. Na życzenie ma ona wyświetlać siatki kontrolne płatów Béziera oraz „kreskowy" obraz czajnika.

16.1. Wyświetlanie siatek kontrolnych — szadery

Na obraz siatki kontrolnej płata Béziera stopnia (n, m) składa się $(m+1)n + (n+1)m = 2mn + m + n$ odcinków. Aby narysować siatki p płatów, możemy wszystkie odcinki wyświetlić, wydając polecenie narysowania *jednego* odcinka w *odpowiedniej liczbie* instancji. Problem sprowadza się do wyznaczenia końców odcinka na podstawie numeru instancji. Szader wierzchołków, który to robi (listing 16.1), jest dostosowany do struktur danych (bloków magazynowych) opisanych w poprzednim rozdziale.

W liniach 16 i 17 obliczane są liczby punktów kontrolnych i liczby odcinków siatki każdego płata. W liniach 18 i 19 na podstawie numeru instancji obliczane są odpowiednio numer płata i numer odcinka w siatce, który należy narysować.

Warunek sprawdzany w linii 20 odróżnia odcinki należące do wierszy i kolumn siatki kontrolnej; przyjęte jest, że odcinki należące do wierszy mają w obrębie siatki numery od 0 do $(m+1)n - 1$.

W liniach 20–46 następuje obliczenie indeksu do tablicy punktów kontrolnych, pod którym znajduje się pierwsza współrzędna potrzebnego punktu kontrolnego, który jest końcem odcinka. Odcinek ma dwa końce; wybór właściwego końca jest dokonywany na podstawie zmiennej wbudowanej `gl_VertexID`, której wartość jest numerem wierzchołka rysowanego prymitywu: dla odcinka jest to liczba 0 albo 1.

Przyjęta w poprzednim rozdziale reprezentacja zbioru płatów Béziera dopuszcza dwie możliwości: z użyciem albo bez użycia tablicy indeksów. W pierwszym przypadku (linie 22–23) odpowiedni indeks jest pobierany z tablicy `cpi.cpi`, z miejsca o numerze

$$(n+1)(m+1)n_p + n_l \quad \text{albo} \quad (n+1)(m+1)n_p + n_l + m + 1,$$

zależnie od tego, który koniec odcinka jest potrzebny; n_p oznacza numer płata, a n_l oznacza

numer odcinka w siatce (jeden koniec każdego odcinka w wierszu jest w jednym z pierwszych $(m + n)n$ punktów kontrolnych, a drugi koniec jest w następnej kolumnie siatki). Indeks odczytany z tablicy `cpi.cpi` jest mnożony przez liczbę współrzędnych punktu (podaną w zmiennej `bezp.dim`).

Listing 16.1. Szader wierzchołków programu wyświetlania siatek kontrolnych

GLSL

```
#version  430 core

#define MAX_DEG 10

layout(location=0) in vec4 in_Position;

 .... /* tu deklaracje  bloków takie, */
 .... /* jak w liniach 15-28 na listingu 15.4 */

void main ( void )
{
  int  n, m, nn, nlines, np, nl, a, b, i;
  vec4 p;

  n = bezp.udeg;  m = bezp.vdeg;
  nn = (n+1)*(m+1);
  nlines = 2*m*n+m+n;           /* liczba odcinków siatki */
  np = gl_InstanceID / nlines;  /* numer siatki */
  nl = gl_InstanceID % nlines;  /* numer odcinka w siatce */
  if ( nl < (m+1)*n )   /* odcinek wiersza */
    if ( bezp.use_ind )
      i = cpi.cpi[nn*np +
                  ((gl_VertexID == 0) ? nl : nl+m+1)] * bezp.dim;
    else {
      a = nl / n;  /* numer wiersza */
      b = nl % n;  /* numer odcinka w wierszu */
      if ( gl_VertexID != 0 ) b ++;
      i = (np / bezp.nq)*bezp.stride_p + (np % bezp.nq)*bezp.stride_q +
          a*bezp.stride_v + b*bezp.stride_u;
    }
  }
  else {                  /* odcinek kolumny */
    nl -= (m+1)*n;
    a = nl / m;  /* numer kolumny */
    b = nl % m;  /* numer odcinka w kolumnie */
    if ( bezp.use_ind ) {
      i = cpi.cpi[nn*np + a*(m+1) +
                  ((gl_VertexID == 0) ? b : b+1)] * bezp.dim;
    }
    else {
```

```
41:       a = nl / m;  /* numer kolumny */
42:       b = nl % m;  /* numer odcinka w kolumnie */
43:       i = (np / bezp.nq)*bezp.stride_p + (np % bezp.nq)*bezp.stride_q +
44:           a*bezp.stride_u + b*bezp.stride_v;
45:     }
46:   }
47:   switch ( bezp.dim ) {
48: case 2: p = vec4 ( cp.cp[i], cp.cp[i+1], 0.0, 1.0 );  break;
49: case 3: p = vec4 ( cp.cp[i], cp.cp[i+1], cp.cp[i+2], 1.0 );  break;
50: case 4: p = vec4 ( cp.cp[i], cp.cp[i+1], cp.cp[i+2], cp.cp[i+3] );  break;
51: default: p = vec4 ( 0.0 );  break;
52:   }
53:   gl_Position = trb.vpm * (trb.mm * p);
54:  /*main*/
```

Jeśli tablica indeksów nie jest używana, to odpowiedni indeks do tablicy punktów kontrolnych jest obliczany na podstawie kroków, z jakimi punkty kontrolne są rozmieszczone w tablicy `cp.cp` — kroki te są podane w zmiennych `bezp.stride_p` (krok do następnej „kolumny" płatów w siatce), `bezp.stride_q` (krok do następnego „wiersza" płatów w siatce), `bezp.nq` (liczba płatów w kolumnie), `bezp.stride_u` (krok do następnej kolumny siatki) i `bezp.stride_v` (krok do następnego wiersza siatki).

Podobnie wygląda obliczenie indeksu wierzchołka, który jest końcem odcinka należącego do kolumny, w liniach 33–45. W liniach 47–52 jest zrealizowane pobieranie wierzchołka, tj. odczytywanie jego współrzędnych z tablicy `cp.cp`. Dopuszczalne jest podanie dwóch, trzech lub czterech współrzędnych — w pierwszych dwóch przypadkach współrzędne nieobecne otrzymują wartości domyślne, $z = 0$, $w = 1$. W linii 53 następuje obliczenie współrzędnych wierzchołka w układzie kostki standardowej. Można oczywiście rozwiązać rysowanie siatek inaczej — umieścić punkty kontrolne w VBO i użyć odpowiedniej tablicy indeksów. Ale użyty tu sposób nie wymaga żadnych zmian struktur danych reprezentujących płaty Béziera, co ma swoje zalety.

Listing 16.2. Szader fragmentów programu wyświetlania siatek kontrolnych

GLSL

```
1: #version 420 core
2:
3: out vec4 out_Colour;
4:
5: void main ( void )
6: {
7:   out_Colour = vec4 ( 0.0, 0.7, 0.0, 1.0 );
8: } /*main*/
```

Szader fragmentów na listingu 16.2 jest bardzo prościutki: tylko przypisuje zmiennej wyjściowej wartość reprezentującą kolor zielony. Oczywiście, można go zmienić na coś bardziej wyrafinowanego, jeśli ktoś poczuje, że tak trzeba.

16.2. Wyświetlanie siatek kontrolnych — procedury w C

Rysowanie siatki kontrolnej realizuje procedura pokazana na listingu 16.3. Przywiązuje ona do odpowiednich punktów dowiązania bufory z reprezentacją zbioru płatów (tak samo jak procedura `DrawBezierPatches` na listingu 15.11), a następnie (w linii 8) oblicza całkowitą liczbę odcinków siatki kontrolnej jednego płata i przekazuje iloczyn tej liczby i liczby płatów (czyli siatek) jako ostatni parametr w wywołaniu procedury `glDrawArraysInstanced`. Procedura ta ma narysować *jeden* odcinek (o dwóch końcach — liczbę końców odcinka określa trzeci parametr) w wielu instancjach. Wierzchołki, formalnie będące końcami tego jednego odcinka, są ignorowane (szader wierzchołków ma własne zdanie na ten temat), dlatego są wprowadzane do potoku przetwarzania grafiki przy użyciu pustego obiektu tablicy wierzchołków.

Listing 16.3. Procedura `DrawBezierNets`

C

```
1: void DrawBezierNets ( BezierPatchObjf *bp )
2: {
3:   int nlines;
4:
5:   if ( bp ) {
6:     BindBezierPatchBuffers ( bp );
7:     glBindVertexArray ( empty_vao );
8:     nlines = 2*bp->udeg*bp->vdeg + bp->udeg + bp->vdeg;
9:     glDrawArraysInstanced ( GL_LINES, 0, 2, bp->npatches*nlines );
10:    glBindVertexArray ( 0 );
11:    ExitIfGLError ( "DrawBezierNets" );
12:  }
13: } /*DrawBezierNets*/
```

16.3. Nowe i zmienione procedury aplikacji

Listing 16.4 przedstawia opis struktury opakowującej dane części graficznej aplikacji 2A; są w niej dodane pola `cnet` i `skeleton`, których wartości `true` powodują odpowiednio wyświetlanie siatek kontrolnych płatów i krawędzi trójkątów otrzymanych w wyniku rozdrabniania płatów Béziera. Pole `program_id` stało się tablicą o długości 2, pierwszy jej element służy do przechowania identyfikatora programu rysowania płatów, a w drugim jest pamiętany identyfikator programu rysowania siatek kontrolnych.

Listing 16.4. Zmiany struktury `AppData`

C

```
1: typedef struct {
2:     Camera          camera;
3:     BezierPatchObjf *myteapot;
```

```
    TransBl         trans;
    LightBl         light;
    GLint           BezNormals, TessLevel;
    char            cnet, skeleton;
    float           model_rot_axis[3];
    double          model_rot_angle;
    GLuint          program_id[2];
  } AppData;
```

Listing 16.5 przedstawia zmiany procedury LoadBPShaders, której zadaniem jest teraz skompilowanie i przygotowanie do pracy dwóch programów szaderów. Drugi program składa się z szaderów przedstawionych na listingach 16.1 i 16.2.

Listing 16.5. Procedura LoadBPShaders

C

```
void LoadBPShaders ( GLuint program_id[2] )
{
  static const char *filename[] =
    { "app2.vert.glsl", "app2.tesc.glsl", "app2.tese.glsl",
      "app2.geom.glsl", "app2.frag.glsl",
      "app2a1.vert.glsl", "app2a1.frag.glsl" };
  static const GLuint shtype[7] =
    { GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER,
      GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER,
      GL_VERTEX_SHADER, GL_FRAGMENT_SHADER };
  GLuint shader_id[7];
  int    i;

  for ( i = 0; i < 7; i++ )
    shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
  program_id[0] = LinkShaderProgram ( 5, shader_id, "0" );
  program_id[1] = LinkShaderProgram ( 2, &shader_id[5], "1" );
  GetAccessToTransBlockUniform ( program_id[0] );
  GetAccessToLightBlockUniform ( program_id[0] );
  GetAccessToBezPatchStorageBlocks ( program_id[0], false, false );
  AttachUniformTransBlockToBP ( program_id[1] );
  for ( i = 0; i < 7; i++ )
    glDeleteShader ( shader_id[i] );
  ExitIfGLError ( "LoadBPShaders" );
} /*LoadBPShaders*/
```

Wywołania procedur glUseProgram i DrawBezierPatches przez procedurę RedrawMyWorld zostały przeniesione do pomocniczej procedury DrawMyTeapot, a ponadto dodałem osobną procedurę rysującą siatki kontrolne płatów (listing 16.6).

Listing 16.6. Zmienione i nowe procedury rysowania

C

```
void DrawMyTeapot ( AppData *ad )
{
  glUseProgram ( ad->program_id[0] );
  if ( ad->skeleton )
    glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
  else
    glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
  DrawBezierPatches ( ad->myteapot );
} /*DrawMyTeapot*/

void DrawMyCNet ( BezierPatchObjf *bp, GLuint prog_id )
{
  glUseProgram ( prog_id );
  DrawBezierNets ( bp );
} /*DrawMyCNet*/

void RedrawMyWorld ( void )
{
  glClearColor ( 1.0, 1.0, 1.0, 1.0 );
  glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glEnable ( GL_DEPTH_TEST );
  DrawMyTeapot ( &appdata );
  if ( appdata.cnet )
    DrawMyCNet ( appdata.myteapot, appdata.program_id[1] );
  glUseProgram ( 0 );
  glFlush ();
  ExitIfGLError ( "RedrawMyWorld" );
} /*RedrawMyWorld*/
```

Procedura `ProcessCharCommand` interpretuje dwa dodatkowe znaki jako polecenia do wykonania: litera `C` włącza i wyłącza rysowanie siatek kontrolnych, a litera `S` przełącza sposób rysowania płatów — wypełnionych trójkątów lub tylko ich krawędzi (listing 16.7).

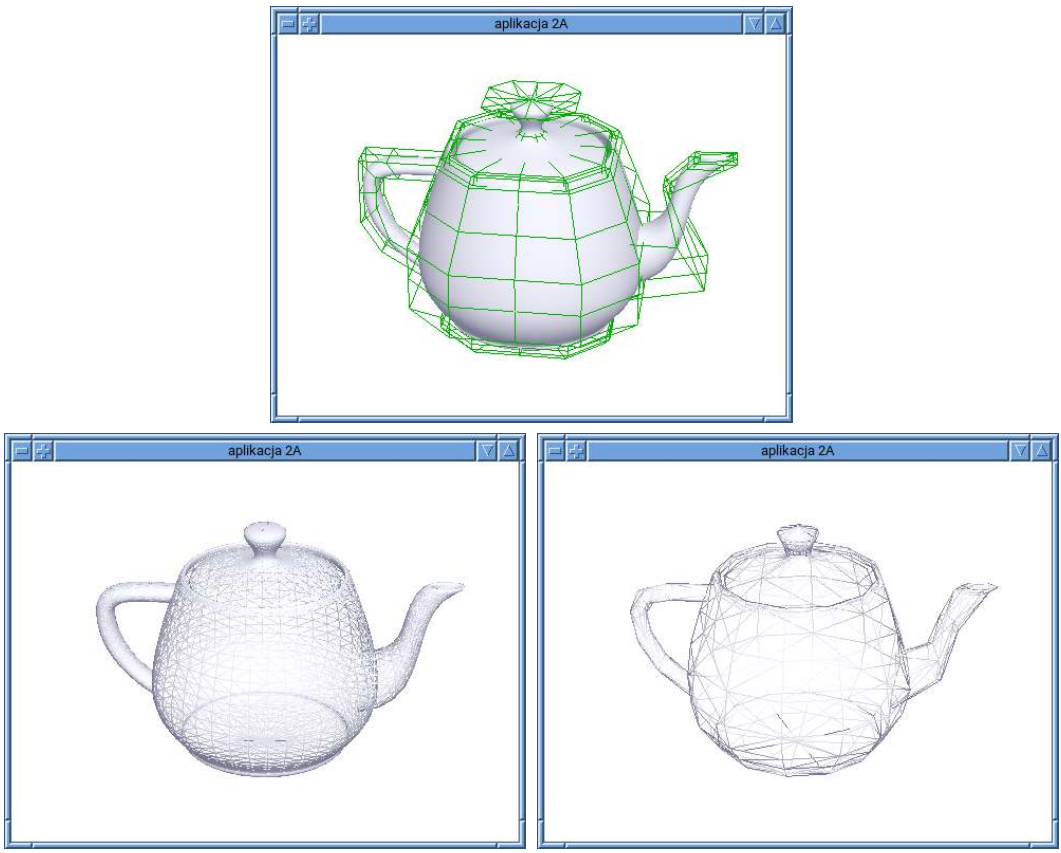
Listing 16.7. Zmiany w procedurze `ProcessCharCommand`

C

```
char ProcessCharCommand ( char charcode )
{
  switch ( toupper ( charcode ) ) {
.... /* obsługa dotychczas używanych klawiszy bez zmian */
case 'C':
    appdata.cnet = !appdata.cnet;
    return true;
case 'S':
    appdata.skeleton = !appdata.skeleton;
    return true;
```

```
11: default:
12:     break;
13:   }
14:   return false;
15: } /*ProcessCharCommand*/
```

Dodatkowe zadanie procedury `DeleteMyWorld` to zlikwidowanie dwóch programów szaderów zamiast jednego, a w części okienkowej nie trzeba wprowadzać żadnych zmian, z wyjątkiem nowego tytułu okna (nazwy aplikacji) i nazw plików włączanych dyrektywą `#include` (nazwałem te pliki `app2a.h` i `app2aproc.h`).



Rysunek 16.1. Czajnik z siatkami kontrolnymi oraz „kreskowe" obrazy dla różnych poziomów rozdrobnienia dziedziny płatów

Obrazki w dolnej części rysunku 16.1 przedstawiają czajnik narysowany „po nowemu". Zwróćmy uwagę, że kolory pikseli składających się na obrazy odcinków są obliczone na podstawie realizowanego przez szadery modelu oświetlenia powierzchni. Wprawdzie wektor normalny odcinka w przestrzeni jest nieokreślony, ale w obliczeniu kolorów używany jest wektor normalny odpowiedniego płata Béziera przybliżanego przez trójkąt, którego bokiem jest rysowany odcinek.

16.4. Ćwiczenia

1. Dodaj możliwość rysowania tylko wierzchołków trójkątów, a także interakcyjnego wybierania (w pewnych granicach) średnicy kropki będącej obrazem punktu.
2. Zmień parametr `GL_FRONT_AND_BACK` w wywołaniu procedury `glPolygonMode` na `GL_FRONT` i zobacz, co z tego wyjdzie.
3. *Zmodyfikuj szader z listingu 16.1 i procedurę z listingu 16.3 tak, aby zamiast jednego odcinka w obliczonej liczbie instancji wyświetlać łamane — kolumny, a potem wiersze siatek kontrolnych płatów Béziera, w odpowiednio mniejszej liczbie instancji. Osobne rysowanie kolumn i wierszy jest konieczne, jeśli stopień n płata ze względu na parametr u (czyli liczba odcinków w wierszu) jest inny niż stopień m ze wzlędu na v. Przetwarzając wierzchołek łamanej, szader wierzchołków otrzyma w zmiennej `gl_VertexID` jego numer, od 0 do m albo od 0 do n.

   Powstaną takie same obrazy, ale te modyfikacje sprawią, że każdy wierzchołek (każdej instancji) łamanej będzie odczytywany z bloku magazynowego i przekształcany do układu kostki standardowej tylko raz.

16.5. Uzupełnienia

16.5.1. Pionizowanie obserwatora

If you can read this sentence, one of us is in trouble.

Napis na samochodzie terenowym

Obie aplikacje nie mają jak dotąd wbudowanych ograniczeń na obroty obserwatora wokół sceny. W pewnych zastosowaniach (np. w grach i w tworzeniu filmów animowanych) może być pożądane „pionizowanie" obserwatora. Ma ono na celu sprawienie, aby obraz osi y układu obserwatora miał ten sam kierunek i zwrot co obraz ustalonego wektora, który oznaczymy symbolem $\boldsymbol{z}$ (bo on wyznacza zenit — na ogół będzie to wersor osi z układu współrzędnych świata). Aby to zrobić, należy odpowiednio obrócić obserwatora wokół osi z *jego* układu, przy czym ma to sens pod warunkiem, że ta oś nie ma kierunku zenitu[1].

Gdy pionizowanie będzie w aplikacji włączone, procedura `_RotateViewer` po złożeniu obrotu określonego przez ostatnie przemieszczenie kursora z obrotem dotychczasowym wykona dodatkowy obrót obserwatora o kąt ϑ znaleziony sposobem opisanym niżej. Wersor $\boldsymbol{z}'$ osi z układu obserwatora jest wektorem jednostkowym osi tego obrotu. Obraz $R_{\boldsymbol{z}',\vartheta}\boldsymbol{y}'$ wektora $\boldsymbol{y}'$, będącego wersorem osi y układu obserwatora, oraz wektory $\boldsymbol{z}'$ i $\boldsymbol{z}$ mają leżeć w jednej płaszczyźnie (czyli wszystkie trzy wektory mają być liniowo zależne). Warunek ten jest zapisany w równaniu

$$\det[R_{\boldsymbol{z}',\vartheta}\boldsymbol{y}', \boldsymbol{z}', \boldsymbol{z}] = \langle (R_{\boldsymbol{z}',\vartheta}\boldsymbol{y}') \wedge \boldsymbol{z}', \boldsymbol{z} \rangle = 0.$$

[1] Czyli nie wtedy, gdy obserwator patrzy pionowo w dół lub do góry.

Na podstawie wzoru (5.17) możemy napisać[2]

$$\begin{aligned}(R_{\boldsymbol{z}',\vartheta}\boldsymbol{y}') \wedge \boldsymbol{z}' &= \left(\boldsymbol{z}'\boldsymbol{z}'^T\boldsymbol{y}' + c(\boldsymbol{y}' - \boldsymbol{z}'\boldsymbol{z}'^T\boldsymbol{y}') + s\boldsymbol{z}' \wedge \boldsymbol{y}'\right) \wedge \boldsymbol{z}' \\ &= c\boldsymbol{y}' \wedge \boldsymbol{z}' - s(\boldsymbol{y}' \wedge \boldsymbol{z}') \wedge \boldsymbol{z}' = c\boldsymbol{x}' + s\boldsymbol{y}'.\end{aligned}$$

Nowe symbole oznaczają $\cos\vartheta = c$, $\sin\vartheta = s$, wektor $\boldsymbol{y}' \wedge \boldsymbol{z}' = \boldsymbol{x}'$ jest wersorem osi x układu obserwatora, w związku z czym $\boldsymbol{x}' \wedge \boldsymbol{z}' = -\boldsymbol{y}'$. Ma zatem być $\langle c\boldsymbol{x}' + s\boldsymbol{y}', \boldsymbol{z}\rangle = 0$, skąd łatwo jest wywieść, że

$$\operatorname{tg}\vartheta = \frac{s}{c} = \frac{-\langle \boldsymbol{x}', \boldsymbol{z}\rangle}{\langle \boldsymbol{y}', \boldsymbol{z}\rangle}.$$

Do obliczenia kąta ϑ aplikacja może użyć funkcji `atan2` ze standardowej biblioteki funkcji matematycznych `libm`. Zwracam uwagę, że jeśli znaki obu jej argumentów (licznika i mianownika) zostaną zmienione, to tangens ϑ pozostanie ten sam, ale wartość funkcji `atan2` zmieni się o π lub $-\pi$, więc zły wybór znaku prowadzi do odwrócenia obserwatora do góry nogami. Znaki przyjęte we wzorze podanym wyżej dają poprawny wynik; głowa do góry.

Jest jeszcze jeden problem: jeśli kąt między osią z układu obserwatora a prostą o kierunku wektora $\boldsymbol{z}$ jest mały, to kąt ϑ obrotu potrzebnego do spionizowania obserwatora może być bardzo duży. Kiedy tak jest, obroty obserwatora wokół sceny „szaleją", co nie wydaje się pożądanym efektem. Można wypróbować różne sposoby przeciwdziałania temu zjawisku. Wypróbowane przeze mnie rozwiązanie polega na uzależnieniu kąta φ obrotu obserwatora od jego położenia. Jest on równy 3°, jeśli oś z układu obserwatora jest prostopadła do kierunku zenitu, i maleje, gdy kąt między tą osią a zenitem maleje.

Listing 16.8 przedstawia zmienioną procedurę `_RotateViewer`, wywoływaną, gdy użytkownik po naciśnięciu przycisku przesuwa mysz. Jej instrukcje konstruujące i przesyłające macierz przejścia do układu obserwatora i położenie obserwatora do bufora w pamięci GPU zostały przeniesione do nowej procedury `LoadViewMatrix`, którą `_RotateViewer` wywołuje na końcu. Pozostałe zmiany w tej procedurze to obliczenie kąta obrotu spowodowanego przesunięciem myszy i wywołanie procedury `Verticalise`, która dokonuje obrotu pionizującego. Te instrukcje są wykonywane, jeśli pionizowanie jest włączone — przypisanie wartości `true` lub `false` (czyli 1 lub 0) zmiennej `ad->vertical`, dodanej do struktury `AppData`, jest sterowane odpowiednim klawiszem.

Procedura `Verticalise` w linii 15 konstruuje macierz 4×4 reprezentującą obrót obserwatora w układzie świata przed spionizowaniem[3]. Dalej potrzebny jest tylko górny lewy blok 3×3 tej macierzy, oznaczymy go symbolem $R_{\boldsymbol{z}_i,\varphi_i}$. Jego odwrotność, $R_{\boldsymbol{z}_i,-\varphi_i}$, opisuje część liniową przejścia od układu świata do układu obserwatora; blok ten jest macierzą ortogonalną (bo obrót jest izometrią), zatem jego odwrotność jest jego transpozycją. Kolumny

[2] Pamiętamy, że iloczyn wektorowy nie jest działaniem łącznym, a zmiana kolejności wektorów zmienia zwrot ich iloczynu wektorowego na przeciwny. Ponadto układ współrzędnych obserwatora powstaje przez obrót i przesunięcie układu świata, tak więc jest to układ izometryczny i zorientowany zgodnie z układem świata (zobacz rozdz. 5 i 6). Jego wersory osi spełniają równania $\boldsymbol{x}' \wedge \boldsymbol{y}' = \boldsymbol{z}'$, $\boldsymbol{y}' \wedge \boldsymbol{z}' = \boldsymbol{x}'$, $\boldsymbol{z}' \wedge \boldsymbol{x}' = \boldsymbol{y}'$.

[3] Zwracam uwagę na znak kąta będącego ostatnim parametrem procedury `M4x4RotateVfv`.

Listing 16.8. Pionizowanie obserwatora

C

```
void LoadViewMatrix ( AppData *ad )
{
  M4x4RotateVfv ( ad->trans.vm, ad->camera.viewer_rvec,
                  -ad->camera.viewer_rangle );
  M4x4MultMTVf ( ad->trans.eyepos, ad->trans.vm, ad->camera.viewer_pos0 );
  M4x4InvTranslateMfv ( ad->trans.vm, ad->camera.viewer_pos0 );
  LoadVPMatrix ( &ad->trans );
} /*LoadViewMatrix*/

void Verticalise ( float vk[3], double *angk )
{
  float  zenith[3] = {0.0,0.0,1.0}, R[16], s, c, vr[3];
  double theta, ang;

  M4x4RotateVfv ( R, vk, *angk );
  s = -V3DotProductf ( &R[0], zenith );
  c = V3DotProductf ( &R[4], zenith );
  theta = atan2 ( s, c );
  V3CompRotationsf ( vr, &ang, &R[8], theta, vk, *angk );
  memcpy ( vk, vr, 3*sizeof(float) );
  *angk = ang;
} /*Verticalise*/

void _RotateViewer ( ApData *ad, double delta_xi, double delta_eta )
{
  float  vi[3], lgt, vk[3];
  double angi, ankg;

  vi[0] = (float)delta_eta*ad->camera.rl/float)ad->camera.win_height;
  vi[1] = (float)delta_xi*ad->camera.tb/float)ad->camera.win_width;
  vi[2] = 0.0;
  lgt = sqrt ( V3DotProductf ( vi, vi ) );
  vi[0] /= lgt;  vi[1] /= lgt;
  angi = -0.052359878;
  if ( ad->vertical )
    angi *= 0.01 + 0.99*fabs ( ad->trans.vm[1]*zenith[0] +
            ad->trans.vm[5]*zenith[1] + ad->trans.vm[9]*zenith[2] );
  V3CompRotationsf ( vk, &angk, ad->camera.viewer_rvec,
                     ad->camera.viewer_rangle, vi, angi );
  if ( ad->vertical )
    Verticalise ( vk, &angk );
  memcpy ( ad->camera.viewer_rvec, vk, 3*sizeof(float) );
  ad->camera.viewer_rangle = angk;
  LoadViewMatrix ( ad );
} /*_RotateViewer*/
```

macierzy R_{z_i,φ_i} opisują (w układzie współrzędnych świata) wersory $\boldsymbol{x}'$, $\boldsymbol{y}'$, $\boldsymbol{z}'$ osi układu obserwatora. Zatem, parametry `&R[0]`, `&R[4]` i `&R[8]` w liniach 16, 17 i 19 przekazują wywoływanym procedurom te wersory. Natomiast w liniach 36–37 mamy obliczenie iloczynu skalarnego wektora $\boldsymbol{z}$ i wersora $\boldsymbol{y}'$ układu obserwatora *przed obróceniem go* za pomocą myszy. Ponieważ w zmiennej `trans.vm` jest w tym momencie przechowywana macierz przejścia do poprzedniego układu obserwatora, część liniowa tego przekształcenia jest opisana przez macierz $R_{z_{i-1},-\varphi_{i-1}} = R^T_{z_{i-1},\varphi_{i-1}}$. Stąd współrzędne wersora $\boldsymbol{y}'$ są pierwszymi trzema liczbami w drugim wierszu tej macierzy. Jako że liczby te w tablicy `trans.vm` nie sąsiadują ze sobą, procedura, zamiast wywołać funkcję `V3DotProductf`, oblicza iloczyn skalarny „na piechotę”.

Obliczony w liniach 36–37 iloczyn skalarny jest kosinusem kąta między osią y układu obserwatora a zenitem. Nominalny kąt obrotu obserwatora, 3°, jest mnożony przez czynnik $0.01 + 0.99|\langle \boldsymbol{y}', \boldsymbol{z}\rangle|$, który przyjmuje wartości z przedziału $[0.01, 1]$. Dzięki temu, jeśli pionizowanie jest włączone, obroty obserwatora są odpowiednio hamowane, co dosyć skutecznie stabilizuje zachowanie programu.

Listing 16.9 przedstawia dodatek do procedury `ProcessCharCommand`. Włączenie pionizowania powinno spowodować natychmiastową korektę układu obserwatora, dlatego wywoływane są procedury `Verticalise` i `LoadViewMatrix` i zgłaszana jest potrzeba wykonania nowego obrazu. W tym momencie wyjaśniło się, po co powstała osobna procedura `LoadViewMatrix` z instrukcjami przeniesionymi z procedury `_RotateViewer`.

Listing 16.9. Włączanie i wyłączanie pionizowania

C

```
1: char ProcessCharCommand ( char charcode )
2: {
3:   switch ( toupper ( charcode ) ) {
4: .... /* pozostałe klawisze bez zmian */
5: case 'V':
6:     if ( (appdata.vertical = !appdata.vertical) ) {
7:       Verticalise ( appdata.camera.viewer_rvec,
8:                     &appdata.camera.viewer_rangle );
9:       LoadViewMatrix ( &appdata );
10:      return true;
11:    }
12:    else
13:      return false;
14: default:
15:    break;
16:  }
17:  return false;
18: } /*ProcessCharCommand*/
```

16.5.2. *Zaawansowane procedury rysowania

W OpenGL-u mamy do dyspozycji procedury, które dają więcej swobody wyboru wierzchołków rysowanych prymitywów niż dotychczas przedstawione procedury `glDrawArrays`, `glDrawElements`, `glDrawArraysInstanced` i `glDrawElementsInstanced`. Poniżej jest krótki opis niektórych z nich; szczegółowy opis wszystkich można znaleźć w specyfikacji [1].

Pierwszy parametr wszystkich procedur rysujących określa tryb, czyli rodzaj prymitywów wprowadzanych do potoku przetwarzania grafiki.

Parametry drugi i trzeci procedury `glMultiDrawArrays` są tablicami liczb całkowitych, których długość n jest określona przez czwarty parametr. Elementy pierwszej tablicy określają początki, a drugiej długości ciągów kolejnych wierzchołków poszczególnych prymitywów danego typu — zatem wywołanie tej procedury jest równoważne n kolejnym wywołaniom procedury `glDrawArrays` z drugimi i trzecimi parametrami o wartościach kolejnych elementów tablic, choć zabiera mniej czasu.

Podobnie działa procedura `glMultiDrawElements`, której parametr drugi jest tablicą długości ciągów wierzchołków, czwarty jest tablicą początków ciągów indeksów w buforze indeksów przywiązanym do celu `GL_ELEMENT_BUFFER`, piąty określa liczbę tych ciągów, a parametr trzeci określa typ indeksów w buforze, `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT` albo `GL_UNSIGNED_INT`. Procedury `glMultiDrawArrays` i `glMultiDrawElements` umożliwiają m.in. narysowanie za jednym wywołaniem wielu łamanych zamkniętych lub wielu wachlarzy trójkątów.

Jeśli (pod)ciąg wierzchołków wprowadzany do potoku przetwarzania grafiki przez jedną z procedur ze słowem `Arrays` w nazwie ma długość n i zaczyna się od miejsca f, to zmienna wejściowa `gl_VertexID` szadera wierzchołków dla i-tego wierzchołka otrzymuje wartość $f + i$. Dla wierzchołków wprowadzanych przez procedury, których nazwa zawiera słowo `Elements`, wartość tej zmiennej jest brana z tablicy indeksów z miejsca $f + i$.

Nazwę `glDrawArraysInstanced` można wydłużyć o ciąg liter `BaseInstance`. Do nazwy `glDrawElements` można dołączyć przyrostek `BaseVertex`, `InstancedBaseVertex` albo `InstancedBaseVertexBaseInstance`. Otrzymamy wtedy nazwy procedur, które mają jeden lub dwa dodatkowe parametry, nazwane `baseinstance` i `basevertex`. Wartość parametru `baseinstance` jest dodawana do indeksu atrybutu o niezerowym dzielniku k (określonym za pomocą procedury `glVertexAttribDivisor`, zobacz p. 15.8.2) przed sięgnięciem do tablicy wierzchołków po ten atrybut. Wartość parametru `basevertex` jest dodawana do identyfikatora wierzchołka przypisanego zmiennej `gl_VertexID`.

17

Aplikacja druga B

Trochę wzbogacimy scenę: dodamy torus lewitujący i obracający się nad wylotem dziobka czajnika. Zastosujemy te same szadery co poprzednio, ale w większym stopniu wykorzystamy ich możliwości[1].

17.1. Iloczyn sferyczny i powierzchnie obrotowe

Mając dwie płaskie krzywe parametryczne,

$$\boldsymbol{e}(u) = \begin{bmatrix} x_{\boldsymbol{e}}(u) \\ y_{\boldsymbol{e}}(u) \end{bmatrix} \quad \text{i} \quad \boldsymbol{m}(v) = \begin{bmatrix} x_{\boldsymbol{m}}(v) \\ y_{\boldsymbol{m}}(v) \end{bmatrix},$$

możemy określić ich **iloczyn sferyczny**, czyli płat powierzchni, którego parametryzacja jest dana wzorem

$$\boldsymbol{p}(u,v) = \begin{bmatrix} x_{\boldsymbol{e}}(u)x_{\boldsymbol{m}}(v) \\ y_{\boldsymbol{e}}(u)x_{\boldsymbol{m}}(v) \\ y_{\boldsymbol{m}}(v) \end{bmatrix}.$$

Dziedziną parametryzacji jest prostokąt — iloczyn kartezjański przedziałów będących dziedzinami krzywych $\boldsymbol{e}$ i $\boldsymbol{m}$. Nazwa opisanego wyżej działania wzięła się stąd, że jeśli krzywa $\boldsymbol{e}$ jest okręgiem jednostkowym o środku w punkcie $\mathbf{0}$, a krzywa $\boldsymbol{m}$ jest półokręgiem o promieniu R, którego środek i oba końce mają współrzędną x równą 0, to iloczyn sferyczny tych krzywych jest sferą o promieniu R.

Jeśli krzywa $\boldsymbol{e}$ jest opisanym wyżej okręgiem, to iloczyn sferyczny jest powierzchnią obrotową, której tworzącą jest krzywa $\boldsymbol{m}$. W szczególności, jeśli krzywa $\boldsymbol{m}$ jest okręgiem nieprzecinającym osi y, to otrzymamy torus.

[1] Przy tej okazji poprawiłem trochę błędów w pierwszej wersji tych szaderów.

Łatwo możemy sprawdzić, że jeśli krzywe $\boldsymbol{e}$ i $\boldsymbol{m}$ są krzywymi Béziera,

$$\boldsymbol{e}(u) = \sum_{i=0}^{n} \begin{bmatrix} x_{\boldsymbol{e}i} \\ y_{\boldsymbol{e}i} \end{bmatrix} B_i^n(u), \qquad \boldsymbol{m}(v) = \sum_{j=0}^{m} \begin{bmatrix} x_{\boldsymbol{m}j} \\ y_{\boldsymbol{m}j} \end{bmatrix} B_j^m(v),$$

to ich iloczyn sferyczny jest płatem Béziera stopnia (n, m), o punktach kontrolnych

$$\boldsymbol{p}_{ij} = \begin{bmatrix} x_{\boldsymbol{e}i} x_{\boldsymbol{m}j} \\ y_{\boldsymbol{e}i} x_{\boldsymbol{m}j} \\ y_{\boldsymbol{m}j} \end{bmatrix}.$$

Ponieważ nie istnieje wielomianowa parametryzacja okręgu[2], nie jest możliwe otrzymanie powierzchni obrotowej z wielomianowych krzywych Béziera. Ale możemy w tym celu użyć krzywych wymiernych. Płaskie krzywe wymierne reprezentujemy w postaci jednorodnej, tzn. jako krzywe wielomianowe w przestrzeni trójwymiarowej:

$$\boldsymbol{E}(u) = \sum_{i=0}^{n} \begin{bmatrix} X_{\boldsymbol{e}i} \\ Y_{\boldsymbol{e}i} \\ W_{\boldsymbol{e}i} \end{bmatrix} B_i^n(u), \qquad \boldsymbol{M}(v) = \sum_{j=0}^{m} \begin{bmatrix} X_{\boldsymbol{m}j} \\ Y_{\boldsymbol{m}j} \\ W_{\boldsymbol{m}j} \end{bmatrix} B_j^m(v).$$

Punkty kontrolne płata jednorodnego (w przestrzeni $\mathbb{R}^4$) reprezentującego iloczyn sferyczny naszych krzywych wymiernych są dane wzorem

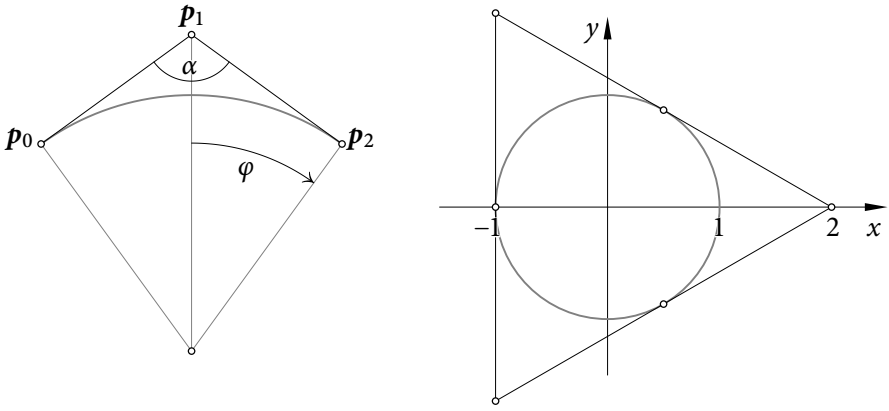
$$\boldsymbol{P}_{ij} = \begin{bmatrix} X_{\boldsymbol{e}i} X_{\boldsymbol{m}j} \\ Y_{\boldsymbol{e}i} X_{\boldsymbol{m}j} \\ W_{\boldsymbol{e}i} Y_{\boldsymbol{m}j} \\ W_{\boldsymbol{e}i} W_{\boldsymbol{m}j} \end{bmatrix}.$$

Pozostaje skonstruować reprezentację Béziera okręgu. Cały okrąg można reprezentować jako krzywą wymierną stopnia 5, a półokrąg jako krzywą stopnia 3, co jest raczej niewygodne. Ale łuk będący dowolną inną częścią okręgu ma reprezentację drugiego stopnia, pokazaną na rysunku 17.1.

Łuk okręgu odpowiadający kątowi 2φ może być reprezentowany jako wymierna krzywa Béziera stopnia 2, której łamana kontrolna ma dwa odcinki o tej samej długości połączone pod kątem $\alpha = \pi - 2\varphi$. Środek okręgu jest punktem przecięcia prostych prostopadłych do tych odcinków, wystawionych w punktach końcowych łamanej, $\boldsymbol{p}_0$ i $\boldsymbol{p}_2$. Wagi tych dwóch punktów powinny być równe 1 i wtedy waga punktu $\boldsymbol{p}_1$ musi być równa $\cos\varphi$. Mając współrzędne kartezjańskie wszystkich trzech punktów, mnożymy je przez odpowiednie wagi i doczepiamy wagę każdego punktu jako ostatnią (trzecią) współrzędną — dostajemy w ten sposób punkty kontrolne $\boldsymbol{P}_0, \boldsymbol{P}_1, \boldsymbol{P}_2 \in \mathbb{R}^3$ jednorodnej reprezentacji łuku.

Cały okrąg możemy otrzymać jako połączenie trzech łuków o tej samej długości; dla każdego z tych łuków jest $\varphi = \frac{\pi}{3} = 60^\circ$ oraz $\cos\varphi = \frac{1}{2}$. Reprezentacja okręgu jednostkowego o środku w początku układu współrzędnych może wyglądać tak, jak na rysunku 17.1 po prawej stronie.

[2] Łuki okręgu można z dużą dokładnością przybliżać łukami wielomianowymi, dzięki czemu na przykład korpus i pokrywka czajnika z Utah wyglądają jak powierzchnie obrotowe, choć nimi nie są (zobacz przypis na s. 571).



Rysunek 17.1. Reprezentacja Béziera łuku okręgu i okrąg złożony z trzech części

17.2. Konstruowanie reprezentacji torusa

Na listingu 17.1 mamy reprezentację okręgu jednostkowego z rysunku 17.1 i dwie procedury, z których pierwsza konstruuje (i wprowadza do pamięci GPU) iloczyn sferyczny wymiernych krzywych Béziera, a druga przygotowuje tworzącą torusa i wywołuje pierwszą procedurę, aby otrzymać reprezentację torusa gotową do rysowania przez GPU.

W liniach 3–5 jest zadeklarowana tablica z wierzchołkami łamanej składającej się z połączonych łamanych kontrolnych trzech jednorodnych krzywych Béziera reprezentujących okrąg jednostkowy. Symbol `SQRT3` reprezentuje liczbę $\sqrt{3}$; druga krzywa ma wspólne punkty kontrolne z krzywymi sąsiednimi.

Pierwsze 4 parametry procedury `EnterRSphericalProduct` opisują krzywą $\boldsymbol{e}$ złożoną z pewnej liczby (podanej jako wartość parametru `eqarcs`) płaskich wymiernych krzywych Béziera stopnia n (podanego w parametrze `eqdeg`). Zmienna `eqstride` określa odległość w tablicy `eqcp` między początkami reprezentacji kolejnych krzywych[3].

W taki sam sposób kolejne 4 parametry opisują krzywą $\boldsymbol{m}$, złożoną z `marcs` krzywych Béziera stopnia `mdeg`. Parametr `colour` określa kolor powierzchni.

W liniach 16 i 17 obliczane są liczby punktów kontrolnych w tablicach `eqcp` i `mcp`, a następnie, w linii 18, jest rezerwowana tablica, w której będą zapamiętane punkty kontrolne powierzchni. W liniach 20–26, na podstawie wzorów podanych wcześniej, są obliczane współrzędne punktów kontrolnych płata jednorodnego reprezentującego iloczyn sferyczny wymiernych krzywych Béziera; zewnętrzna pętla przebiega po kolumnach siatki kontrolnej (z których każda odpowiada jednemu punktowi krzywej $\boldsymbol{e}$), a wewnętrzna pętla przebiega po kolejnych elementach kolumny.

[3] Jeśli krzywe stopnia n mają wspólne punkty kontrolne, jak w wykorzystywanej tu reprezentacji okręgu, to krok powinien być równy n. Ale krzywe mogą być niepołączone i skoro wtedy każda z nich jest reprezentowana przez kolejne $n+1$ punktów w tablicy, trzeba przyjąć krok $n+1$.

Listing 17.1. Procedury konstrukcji iloczynu sferycznego i torusa

C

```
#define SQRT3 1.7320508

GLfloat UnitCircle[7][3] = {{0.5,0.5*SQRT3,1.0},
   {-0.5,0.5*SQRT3,0.5},{-1.0,0.0,1.0},{-0.5,-0.5*SQRT3,0.5},
   {0.5,-0.5*SQRT3,1.0},{1.0,0.0,0.5},{0.5,0.5*SQRT3,1.0}};

BezierPatchObjf* EnterRSphericalProduct (
                    int eqdeg, int eqarcs, int eqstride, GLfloat eqcp[][3],
                    int mdeg, int marcs, int mstride, GLfloat mcp[][3],
                    GLfloat *colour )
{
  GLfloat         *cp;
  int             i, j, k, nw, nk;
  BezierPatchObjf *spr;

  nw = (eqarcs-1)*eqstride + eqdeg + 1;
  nk = (marcs-1)*mstride + mdeg + 1;
  if ( !(cp = malloc ( nw*nk*4*sizeof(GLfloat)) ) )
    return NULL;
  for ( i = k = 0;  i < nw;  i++ )
    for ( j = 0;  j < nk;  j++, k += 4 ) {
      cp[k+0] = eqcp[i][0]*mcp[j][0];
      cp[k+1] = eqcp[i][1]*mcp[j][0];
      cp[k+2] = eqcp[i][2]*mcp[j][1];
      cp[k+3] = eqcp[i][2]*mcp[j][2];
    }
  spr = EnterBezierPatches ( eqdeg, mdeg, 4, eqarcs, marcs, nw*nk, cp,
                             nk*eqstride*4, mstride*4, 4*nk, 4, colour );
  free ( cp );
  return spr;
} /*EnterRSphericalProduct*/

BezierPatchObjf* EnterTorus ( float R, float r, GLfloat *colour )
{
  GLfloat circ[7][3];
  int     i;

  for ( i = 0; i < 7; i++ ) {
    circ[i][0] = r*UnitCircle[i][0] + R*UnitCircle[i][2];
    circ[i][1] = r*UnitCircle[i][1];
    circ[i][2] = UnitCircle[i][2];
  }
  return EnterRSphericalProduct ( 2, 3, 2, UnitCircle, 2, 3, 2, circ,
                                  colour );
} /*EnterTorus*/
```

Po obliczeniu wszystkich punktów kontrolnych, w liniach 27–28, jest wywoływana procedura `EnterBezierPatches` z listingu 15.8, która tworzy odpowiednie bufory (SSBO) w pamięci GPU i przesyła do nich dane opisujące płaty. Rysowanie tych płatów będzie odbywać się bez używania tablicy indeksów. Kroki, z jakimi szadery będą „poruszać się" po tablicy punktów kontrolnych, są podane w jednostkach odpowiadających długości *jednej* liczby zmiennopozycyjnej[4]. Tablicę, w której procesor (CPU) umieścił obliczone punkty, można po przesłaniu danych do pamięci GPU oddać do recyklingu (linia 29).

Procedura `EnterTorus` konstruuje reprezentację torusa wyznaczonego przez promień szkieletu R i promień tworzącej r. Korzystając z reprezentacji okręgu jednostkowego w tablicy `UnitCircle`, procedura konstruuje tworzącą torusa, tj. okrąg o promieniu r, którego środek jest w punkcie $(R, 0)$. W tym celu wystarczy poddać łamane kontrolne jednokładności o skali r, a następnie przesunąć. Zwróćmy uwagę na sposób obliczania przesunięcia — mając wektor współrzędnych jednorodnych, należy współrzędne wektora przesunięcia pomnożyć przez wagę przesuwanego punktu, tak jak w linii 39.

Obie krzywe, których iloczynem sferycznym jest torus, są reprezentowane przez 7 punktów kontrolnych, przy czym krok w każdej z tablic (parametry trzeci i siódmy w linii 43) jest równy 2.

17.3. Zmiany w aplikacji

Ruchy czajnika i torusa są w pewnym sensie niezależne — czajnik ma się obracać, jeśli użytkownik wyda takie polecenie (naciskając klawisz spacji), torus zaś ma się obracać nad dziobkiem czajnika przez cały czas. Dlatego każdy z tych obiektów będzie miał własną macierz przekształcenia modelu, uaktywnianą (tj. przypisywaną do odpowiedniej zmiennej jednolitej) przed jego rysowaniem.

Listing 17.2 przedstawia strukturę danych aplikacji ze zmienionymi polami. Pole `mytorus` jest wskaźnikiem struktury reprezentującej torus. Zamiast jednej macierzy przekształcenia modelu (tj. przejścia od układu współrzędnych modelu do układu świata) są dwie macierze, odpowiednio dla czajnika i torusa. Są też dwa niezależnie obliczane kąty obrotu tych obiektów, ale ich osie obrotu mają ten sam kierunek, reprezentowany przez wektor pamiętany w polu `model_rot_axis` (co można oczywiście zmienić). Zamiast jednej są dwie różne prędkości obrotowe, zapisane w pokazanych na listingu makrodefinicjach.

Listing 17.2. Zmiany struktury danych aplikacji

C

```
1: #define ANGULAR_VELOCITY1 (0.25*PI)
2: #define ANGULAR_VELOCITY2 (-2.0*PI)
3:
```

[4] Jednostka miary kroków podawanych jako parametry procedury `EnterRSphericalProduct` jest długością reprezentacji jednego *punktu*, składającego się z trzech liczb. Czy lepiej jest ujednolicić jednostkę używaną w całym programie, czy stosować w każdym miejscu jednostkę najwygodniejszą lub najbardziej naturalną? Nie znam oczywistej odpowiedzi na ten dylemat.

```
4: typedef struct {
5:     Camera          camera;
6:     BezierPatchObjf *myteapot, *mytorus;
7:     TransBl         trans;
8:     LightBl         light;
9:     GLint           BezNormals, TessLevel;
10:    char            cnet, skeleton, animate;
11:    float           model_rot_axis[3];
12:    double          teapot_rot_angle, torus_rot_angle;
13:    GLfloat         teapot_mmatrix[16], torus_mmatrix[16];
14:    GLuint          program_id[2];
15:  } AppData;
```

Każdy z obiektów ma teraz swoją procedurę obliczającą macierz przekształcenia modelu (listing 17.3). Procedury te nie przesyłają macierzy modelu do bufora w pamięci GPU, ponieważ to będzie robione bezpośrednio przed rysowaniem każdego obiektu (i ewentualnie jego siatek kontrolnych). Macierz czajnika reprezentuje złożenie (niezmiennego) skalowania osi o współczynniki 1/3, 1/3 i 4/9 z obrotem wokół osi z układu świata o kąt, którego miara jest podana w zmiennej `teapot_rot_angle`. Przekształcenie torusa jest nieco bardziej skomplikowane. Stała część jest jednokładnością w skali 1/10, złożoną z obrotem o kąt prosty wokół osi x. Przekształcenie zależne od czasu dokonuje obrotu wokół osi z i przemieszcza obrócony torus do punktu nad dziobkiem czajnika. W układzie współrzędnych modelu (czajnika) punkt ten ma współrzędne $(3.1, 0, 2.9)$. Jego współrzędne w układzie świata można obliczyć, znając macierz przekształcenia czajnika, dlatego procedurę `SetupTeapotMatrix` trzeba wywoływać zawsze przed procedurą `SetupTorusMatrix`, która przeprowadza to obliczenie w linii 18. W linii 19 macierz złożenia przekształceń opisanych wcześniej jest mnożona (z lewej strony) przez macierz przesunięcia torusa.

Procedura `InitMyWorld` konstruuje te macierze dla początkowych zerowych kątów obrotu i dodatkowo wywołuje procedurę tworzenia torusa.

Listing 17.3. Zmienione i nowe procedury inicjalizacji

C

```
1: void SetupTeapotMatrix ( AppData *ad )
2: {
3: #define SCF (1.0/3.0)
4:   M4x4Translatef ( ad->teapot_mmatrix, 0.0, 0.0, -0.6 );
5:   M4x4MScalef ( ad->teapot_mmatrix, SCF, SCF, SCF*4.0/3.0 );
6:   M4x4MRotateVfv ( ad->teapot_mmatrix,
7:                    ad->model_rot_axis, ad->teapot_rot_angle );
8: #undef SCF
9: } /*SetupTeapotMatrix*/
10:
11: void SetupTorusMatrix ( AppData *ad )
12: {
13:   GLfloat p[4] = {3.1,0.0,2.9,1.0}, q[4];
```

```
14:
15:   M4x4RotateZf ( ad->torus_mmatrix, ad->torus_rot_angle );
16:   M4x4MRotateXf ( ad->torus_mmatrix, 0.5*PI );
17:   M4x4MScalef ( ad->torus_mmatrix, 0.1, 0.1, 0.1 );
18:   M4x4MultMVf ( q, ad->teapot_mmatrix, p );
19:   M4x4TranslateMfv ( ad->torus_mmatrix, q );
20: } /*SetupTorusMatrix*/
21:
22: void ConstructMyTorus ( AppData *ad )
23: {
24:   GLfloat MyColour[4] = { 0.2, 0.3, 1.0, 1.0 };
25:
26:   ad->mytorus = EnterTorus ( 1.0, 0.5, MyColour );
27:   SetBezierPatchTessLevel ( ad->mytorus, ad->TessLevel );
28:   SetBezierPatchNVS ( ad->mytorus, ad->BezNormals );
29: } /*ConstructMyTorus*/
30:
31: void InitMyWorld ( int argc, char *argv[], int width, int height )
32: {
33:   float axis[4] = {0.0,0.0,1.0};
34:
35:   memset ( &appdata, 0, sizeof(AppData) );
36:   LoadBPShaders ( appdata.program_id );
37:   appdata.trans.trbuf = NewUniformTransBlock ();
38:   appdata.light.lsbuf = NewUniformLightBlock ();
39:   TimerInit ();
40:   memcpy ( appdata.model_rot_axis, axis, 4*sizeof(float) );
41:   appdata.teapot_rot_angle = appdata.torus_rot_angle = 0.0;
42:   SetupTeapotMatrix ( &appdata );
43:   SetupTorusMatrix ( &appdata );
44:   InitCamera ( &appdata, width, height );
45:   appdata.TessLevel = 10;
46:   appdata.BezNormals = GL_TRUE;
47:   appdata.cnet = appdata.skeleton = false;
48:   ConstructMyTeapot ( &appdata );
49:   ConstructMyTorus ( &appdata );
50:   InitLights ( &appdata );
51: } /*InitMyWorld*/
```

Listing 17.4 przedstawia nową procedurę rysowania torusa i instrukcje dodane do procedury `RedrawMyWorld`. Siatki kontrolne obu zespołów płatów Béziera, jeśli mają być rysowane, to są rysowane po płatach, z wykorzystaniem odpowiednich macierzy przekształcenia modelu.

W aplikacji 2B torus obraca się przez cały czas, natomiast polecenie włączenia lub wyłączenia animacji, przez naciśnięcie klawisza spacji, powoduje uruchomienie lub zatrzymanie obracania czajnika. Z tego powodu zmienna `animate` została przeniesiona z części okienkowej aplikacji do części graficznej — stała się polem struktury `AppData` (listing 17.2, linia 10).

Listing 17.4. Nowe procedury rysowania

C

```
void DrawMyTorus ( AppData *ad )
{
  if ( ad->skeleton )
    glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
  else
    glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
  glUseProgram ( ad->program_id[0] );
  DrawBezierPatches ( ad->mytorus );
} /*DrawMyTorus*/

void RedrawMyWorld ( void )
{
  glClearColor ( 1.0, 1.0, 1.0, 1.0 );
  glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glEnable ( GL_DEPTH_TEST );
  LoadMMatrix ( &appdata.trans, appdata.teapot_mmatrix );
  DrawMyTeapot ( &appdata );
  if ( appdata.cnet )
    DrawMyCNet ( appdata.myteapot, appdata.program_id[1] );
  LoadMMatrix ( &appdata.trans, appdata.torus_mmatrix );
  DrawMyTorus ( &appdata );
  if ( appdata.cnet )
    DrawMyCNet ( appdata.mytorus, appdata.program_id[1] );
  glUseProgram ( 0 );
  glFlush ();
  ExitIfGLError ( "RedrawMyWorld" );
} /*RedrawMyWorld*/
```

Procedura `ProcessCharCommand` (listing 17.5) po otrzymaniu spacji przypisuje temu polu wartość `true`, po czym uruchamia stoper, albo `false`. Jeśli pole `animate` ma wartość `true`, to kąt obrotu czajnika jest zwiększany o iloczyn prędkości kątowej i czasu, który upłynął. Kąt obrotu torusa jest zwiększany zawsze, bez względu na wartość pola `animate`. Dodatkowe zmiany procedury `ProcessCharCommand` mają na celu przypisanie parametru rozdrobnienia płatów i sposobu obliczania wektora normalnego dla *obu* obiektów — czajnika i torusa.

Procedura `CharFunc` w części okienkowej (listing 17.6) przekazuje wszystkie znaki do zinterpretowania procedurze `ProcessCharCommand`. W procedurze `main` aplikacji przed wywołaniem procedury `MessageLoop` adres procedury `IdleFunc` jest „raz na zawsze" przypisywany zmiennej `idlefunc`, a procedura `ToggleAnimation` została usunięta.

Listing 17.5. Procedury animacji

C

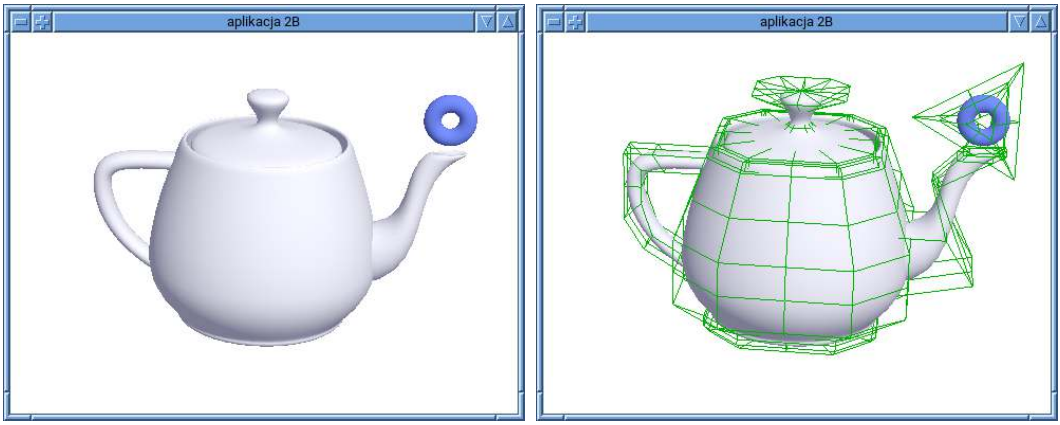
```
char MoveOn ( void )
{
  double dt;
```

```
4:
5:   dt = TimerTocTic ();
6:   if ( appdata.animate ) {
7:     if ( (appdata.teapot_rot_angle += ANGULAR_VELOCITY1 * dt) >= PI )
8:       appdata.teapot_rot_angle -= 2.0*PI;
9:     SetupTeapotMatrix ( &appdata );
10:  }
11:  if ( (appdata.torus_rot_angle += ANGULAR_VELOCITY2 * dt) >= PI )
12:    appdata.torus_rot_angle -= 2.0*PI;
13:  SetupTorusMatrix ( &appdata );
14:  return true;
15: } /*MoveOn*/
16:
17: char ProcessCharCommand ( char charcode )
18: {
19:   switch ( toupper ( charcode ) ) {
20: case '+':
21:     if ( appdata.TessLevel < MAX_TESS_LEVEL ) {
22:       SetBezierPatchTessLevel ( appdata.myteapot, ++appdata.TessLevel );
23:       SetBezierPatchTessLevel ( appdata.mytorus, appdata.TessLevel );
24:       return true;
25:     }
26:     else return false;
27: case '-':
28:     if ( appdata.TessLevel > MIN_TESS_LEVEL ) {
29:       SetBezierPatchTessLevel ( appdata.myteapot, --appdata.TessLevel );
30:       SetBezierPatchTessLevel ( appdata.mytorus, appdata.TessLevel );
31:       return true;
32:     }
33:     else return false;
34: case 'N':
35:     appdata.BezNormals = appdata.BezNormals == 0;
36:     SetBezierPatchNVS ( appdata.myteapot, appdata.BezNormals );
37:     SetBezierPatchNVS ( appdata.mytorus, appdata.BezNormals );
38:     return true;
39:     .... /* inne instrukcje bez zmian */
40: case ' ':
41:     if ( (appdata.animate = !appdata.animate) )
42:       TimerTic ();
43:     return appdata.animate;
44: default:
45:     return false;
46:   }
47: } /*ProcessCharCommand*/
```

Do procedury sprzątającej `DeleteMyWorld` trzeba dodać wywołanie procedury `DeleteBezierPatches` w celu zlikwidowania reprezentacji torusa.

Listing 17.6. Procedura `CharFunc`

```C
void CharFunc ( GLFWwindow *win, unsigned int charcode )
{
  redraw = ProcessCharCommand ( charcode );
} /*CharFunc*/
```



Rysunek 17.2. Czajnik z torusem oraz ich siatki kontrolne

17.4. Ćwiczenia

1. Napisz procedurę, która oblicza punkty kontrolne płata Béziera będącego iloczynem sferycznym płaskich wielomianowych krzywych Béziera i dołącz do aplikacji rysowanie obiektu będącego takim płatem.
2. Utwórz reprezentację półokręgu (złożoną z dwóch wymiernych krzywych Béziera opisujących ćwiartki okręgu) i użyj jej do skonstruowania sfery, którą następnie dodaj do sceny.
3. Zmień aplikację tak, aby można było niezależnie włączać i wyłączać obroty czajnika i torusa.
4. Jeśli płat tensorowy $\boldsymbol{p}$ jest iloczynem sferycznym krzywych $\boldsymbol{e}$ i $\boldsymbol{m}$, to można przesłać do bufora w pamięci GPU tylko reprezentacje tych dwóch krzywych. Szader rozdrabniania, mając współrzędne u, v punktu w dziedzinie płata, może obliczyć punkty $\boldsymbol{e}(u)$ i $\boldsymbol{m}(v)$ (albo punkty $\boldsymbol{E}(u)$ i $\boldsymbol{M}(v)$ krzywych jednorodnych), a następnie obliczyć punkt $\boldsymbol{p}(u, v)$, podstawiając odpowiednie współrzędne obliczonych punktów krzywych do wzorów podanych na początku tego rozdziału. Taka reprezentacja płata zajmuje mniej miejsca, a obliczenie punktu zabiera mniej czasu niż pełna reprezentacja i ogólny algorytm przetwarzania płatów tensorowych. Zatem, napisz odpowiedni szader rozdrabniania i procedury

w C przesyłające taką reprezentację iloczynu sferycznego krzywych Béziera do pamięci GPU, zmodyfikuj aplikację (dodając taki płat), *uruchom ją* i wykonaj eksperymenty.

17.5. *Uzupełnienia

17.5.1. Modyfikowanie triangulacji płata

Jeśli krzywa Béziera ma stopień 1, to jest odcinkiem (lub punktem), a wtedy można ją narysować bez rozdrabniania. Podobnie, jeśli stopień płata ze względu na parametr u jest równy 1, to wszystkie krzywe stałego parametru v są odcinkami, a jeśli stopień ze względu na v jest równy 1, to krzywe stałego parametru u są odcinkami[5]. Zobaczmy, jak można na tej podstawie przyspieszyć rysowanie płatów Béziera.

Listing 17.7. Szader sterowania rozdrabnianiem

GLSL

```
#version 430 core
layout(vertices=1) out;
layout(location=0) in int instance[];  /* zobacz listing 15.3 */
layout(location=0) out int inst[];

layout(std430,binding=2) buffer BezPatch { .... /* listing 15.4 */ } bezp;

void main ( void )
{
  if ( gl_InvocationID == 0 ) {
    gl_TessLevelOuter[1] = gl_TessLevelOuter[3] = gl_TessLevelInner[0] =
              bezp.udeg > 1 ? bezp.TessLevel : 3;
    gl_TessLevelOuter[0] = gl_TessLevelOuter[2] = gl_TessLevelInner[1] =
              bezp.vdeg > 1 || bezp.udeg == 1 ? bezp.TessLevel : 3;
    inst[gl_InvocationID] = instance[gl_InvocationID];
  }
} /*main*/
```

Wszystkie parametry rozdrabniania dziedziny płata muszą być większe lub równe 3; zapewnia to jedną „warstwę" trójkątów wokół brzegu dziedziny i niepuste „wnętrze" podzielone na co najmniej dwa trójkąty (rys. 15.3). Dlatego pokazany na listingu 17.7 szader sterowania rozdrabnianiem, dla płatów stopnia 1 ze względu na u, do tablic `gl_TessLevelOuter` i `gl_TessLevelInner` wpisuje liczbę 3, aby na tyle części podzielić dziedzinę płata odcinkami pionowymi. Jeśli stopień ze względu na u jest większy, to parametry podziału otrzymują wartość zmiennej `bezp.TessLevel`. Podobny algorytm jest użyty do podziału dziedziny odcinkami poziomymi, w zależności od stopnia płata ze względu parametr v, przy czym jeśli stopień płata ze względu na *oba* parametry jest równy 1, to pionowe boki dziedziny pozostaną

[5] W obu przypadkach mamy do czynienia z tak zwaną **powierzchnią prostokreślną**.

podzielone na `bezp.TessLevel` części. Gdyby nie były podzielone, to płat zostałby dalej zamieniony na dwa trójkąty; poza przypadkiem, gdy ten płat jest płaskim czworokątem, byłoby to słabe jego przybliżenie.

Listing 17.8 pokazuje zmiany wprowadzone do szadera rozdrabniania w celu dodatkowego zmniejszenia liczby trójkątów otrzymanych z podziału dziedziny płata. Jeśli jego stopień ze względu na u jest równy 1, a współrzędna u punktu w dziedzinie, przekazana przez etap rozdrabniania dziedziny, jest mniejsza niż 1, to jest zastępowana przez 0 — czyli punkt jest rzutowany na lewy bok kwadratu będącego dziedziną (rys. 17.3). W rezultacie wszystkie trójkąty, których żaden wierzchołek nie leży na prawym boku kwadratu, będą zdegenerowane do odcinków na lewym boku (i będą miały jeden bok o długości 0), a pozostałe trójkąty zostaną odpowiednio „rozciągnięte". Podobne przekształcenie jest stosowane wtedy, gdy stopień płata ze względu na v jest równy 1.

Listing 17.8. Zmieniona procedura `main` szadera rozdrabniania

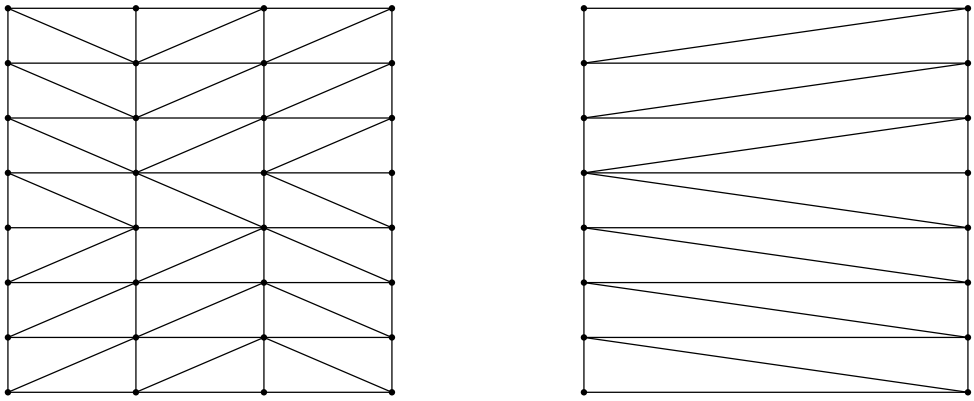
GLSL

```
1: void main ( void )
2: {
3:   vec4  pos, nv;
4:   float u, v;
5:
6:   pos = nv = vec4 ( 0.0 );
7:   u = bezp.udeg == 1 && gl_TessCoord.x < 1.0 ? 0.0 : gl_TessCoord.x;
8:   v = bezp.udeg > 1 && bezp.vdeg == 1 && gl_TessCoord.y < 1.0 ?
9:           0.0 : gl_TessCoord.y;
10:  switch ( bezp.dim ) {
11: case 2: BPHorner2f ( u, v, pos, nv );  break;
12: case 3: BPHorner3f ( u, v, pos, nv );  break;
13: case 4: BPHorner4f ( u, v, pos, nv );  break;
14:  }
15:  gl_Position = trb.mvpm*pos;
16:  .... /* dalej tak, jak na listingu 15.4 */
17: } /*main*/
```

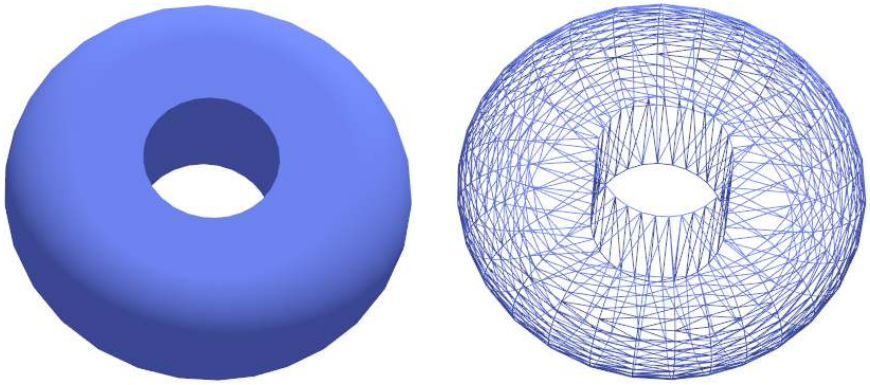
Kolejną zmianę warto wprowadzić do szadera geometrii (zobacz listing 15.5), zastępując instrukcję obliczającą wektor normalny płaszczyzny trójkąta (w linii 8) przez

```
if ( length ( tnv = cross ( v1, v2 ) ) == 0.0 )
  return;
tnv = normalize ( tnv );
```

Jeśli trójkąt jest zdegenerowany, to obliczony iloczyn wektorowy jest wektorem zerowym. Dzięki temu, że szader geometrii po wykryciu takiego stanu rzeczy natychmiast zakończy działanie (bez wchodzenia w pętlę z wywołaniami procedury `EmitVertex`), żaden zdegenerowany trójkąt nie przedostanie się do etapów obcinania i rasteryzacji, w których tylko marnowałby nam cenny czas i energię.



Rysunek 17.3. Zmiana triangulacji dziedziny



Rysunek 17.4. Powierzchnia obrotowa z płatów Béziera stopnia $(2,2)$ i $(2,1)$

Rysunek 17.4 przedstawia trójkąty przybliżające powierzchnię obrotową zbudowaną z wymiernych płatów Béziera — to jest krążek do przekładania na wieżach Hanoi. Powierzchnia składa się z części torusa oraz z powierzchni bocznej walca i dwóch płaskich dysków. Walec i dyski zostały otrzymane jako iloczyn sferyczny okręgu i łamanej złożonej z trzech odcinków. Narysowane krawędzie trójkątów uwidoczniają sposób rozdrobnienia płatów przez szadery zmodyfikowane w opisany wyżej sposób.

17.5.2. Powierzchnie zakreślane

Iloczyn sferyczny jest szczególnym przypadkiem znacznie ogólniejszej konstrukcji znanej jako **zakreślanie**. Krzywą zwaną **przekrojem** przesuwamy wzdłuż innej krzywej, zwanej **prowadnicą**, poddając ją dodatkowym przekształceniom (skalowaniom i obrotom), które naj-

wygodniej jest określić przy użyciu trzech dodatkowych krzywych. Powierzchnia zakreślana składa się z punktów przestrzeni „dotkniętych" przez przemieszczający się przekrój.

Parametryzacja przekroju, oznaczana niżej literą $\boldsymbol{q}$, jest określona w przedziale $[v_0, v_1]$; dziedziną parametryzacji $\boldsymbol{r}$ prowadnicy oraz krzywych dodatkowych $\boldsymbol{w}_1$, $\boldsymbol{w}_2$ i $\boldsymbol{w}_3$ jest przedział $[u_0, u_1]$. Wtedy parametryzacja powierzchni zakreślanej jest dana wzorem

$$\boldsymbol{p}(u,v) = \boldsymbol{r}(u) + \boldsymbol{w}_1(u)x_{\boldsymbol{q}}(v) + \boldsymbol{w}_2(u)y_{\boldsymbol{q}}(v) + \boldsymbol{w}_3(u)z_{\boldsymbol{q}}(v). \tag{17.1}$$

Symbole $x_{\boldsymbol{q}}(v)$, $y_{\boldsymbol{q}}(v)$ i $z_{\boldsymbol{q}}(v)$ oznaczają współrzędne kartezjańskie punktu $\boldsymbol{q}(v)$, przez które mnożymy punkty krzywych dodatkowych traktowane jak wektory swobodne. Zauważmy, że dla każdego ustalonego u macierz 4×4

$$A(u) = \begin{bmatrix} \boldsymbol{w}_1(u) & \boldsymbol{w}_2(u) & \boldsymbol{w}_3(u) & \boldsymbol{r}(u) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

reprezentuje pewne przekształcenie afiniczne. Obraz przekroju w tym przekształceniu jest krzywą stałego parametru u powierzchni zakreślanej.

Jeśli w konstrukcji są użyte krzywe Béziera lub B-sklejane (zobacz dodatek B), to na podstawie punktów kontrolnych tych krzywych łatwo jest znaleźć punkty kontrolne płata zakreślanego — wyprowadzenie odpowiednich wzorów, niewiele bardziej skomplikowanych niż wzory podane w podrozdziale 17.1, polecam jako **ćwiczenie**. Jednak podobnie jak w przypadku iloczynu sferycznego można w pamięci GPU umieścić punkty kontrolne krzywych. Wtedy szader rozdrabniania mógłby obliczać punkty płata, znajdując punkty krzywych dla danych liczb u i v i podstawiając je do wzoru (17.1), co skutkowałoby znaczną oszczędnością miejsca w pamięci i przyspieszeniem obliczeń (zobacz ćwiczenie 4 na s. 428).

Przykładami powierzchni, które można otrzymać przez zakreślanie, są uchwyt i dziobek czajnika z Utah. Polecam niezbyt trudne **ćwiczenie** z „inżynierii odwrotnej": na podstawie punktów kontrolnych uchwytu i dziobka znajdź przekrój (w obu przypadkach ten sam), prowadnicę i pozostałe krzywe, z których powstają uchwyt i dziobek. **Wskazówka:** postawione w ćwiczeniu zadanie ma nieskończenie wiele rozwiązań, takich że wszystkie krzywe użyte do skonstruowania tych powierzchni są płaskie i każda krzywa składa się z dwóch połączonych krzywych Béziera trzeciego stopnia. Najprostsze do znalezienia są takie rozwiązania, w których punkty kontrolne prowadnic są punktami kontrolnymi uchwytu albo dziobka (zobacz rys. 24.1 na s. 625).

18

Aplikacja druga C

Lambertowski model oświetlenia zastąpimy **modelem Blinna-Phonga**, który umożliwia osiągnięcie lepszego realizmu obrazu, dzięki wytworzeniu efektu zwierciadlanego odbicia światła. Umożliwia on tworzenie obrazów obiektów, których powierzchnie nie są matowe. Trzeba wyraźnie podkreślić[1], że model ten nie jest oparty na analizie fizycznego oddziaływania światła z odbijającą je powierzchnią. Model ten jest empirycznym wzorem, który po dobraniu występujących w nim parametrów dosyć dobrze przybliża skutki oglądania oświetlonej powierzchni. Występujące we wzorze parametry opisują własności materiału i powierzchni, w tym koloru i połysku.

18.1. Modele oświetlenia Phonga i Blinna-Phonga

Aby Słońce nie odbijało się w przyrządach celowniczych, należy je przedtem okopcić (*przyrządy, nie Słońce*).

Anna

W modelu oświetlenia opracowanym w 1975 r. przez Bui Tuong Phonga do wyrażenia po prawej stronie wzoru (10.1) jest dodany składnik opisujący dodatkowe światło, które dotarło do obserwatora po odbiciu w powierzchni będącej niedoskonałym lustrem:

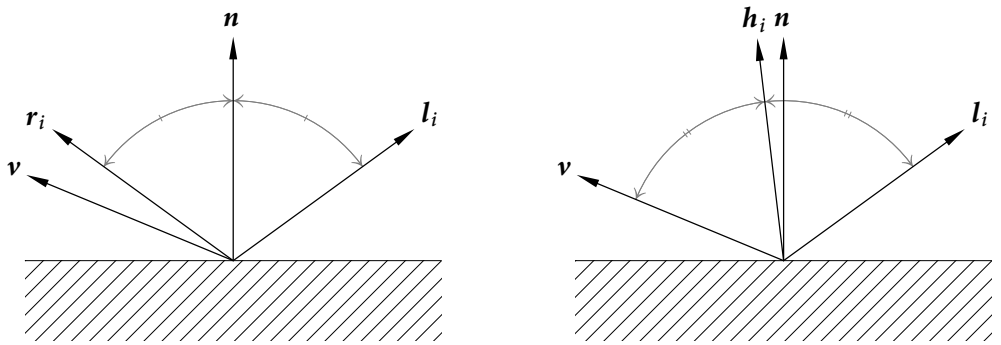
$$L = \boldsymbol{a} \sum_{i=0}^{n-1} \left(I_i^{\text{amb}} + I_i^{\text{dir}} v_i |\langle \boldsymbol{l}_i, \boldsymbol{n} \rangle| \right) + \boldsymbol{s} \sum_{i=0}^{n-1} I_i^{\text{dir}} v_i W(\boldsymbol{n}, \boldsymbol{l}_i, \boldsymbol{v}) \langle \boldsymbol{r}_i, \boldsymbol{v} \rangle^m. \tag{18.1}$$

We wzorze tym symbole $\boldsymbol{n}$, $\boldsymbol{l}_i$, $\boldsymbol{v}$ oraz $\boldsymbol{r}_i$ oznaczają wektory *jednostkowe*, odpowiednio **wektor normalny powierzchni**, **wektor do źródła światła**, **wektor do obserwatora** i **wektor idealnego odbicia**. Ten ostatni wektor wyznacza kierunek, w którym foton padający na dany

[1] wężykiem

punkt powierzchni z kierunku wektora $\boldsymbol{l}_i$ poleciałby po odbiciu, gdyby ta powierzchnia była idealnym lustrem. Można go obliczyć na podstawie wzoru (zobacz podrozdz. 5.5)

$$\boldsymbol{r}_i = 2\langle \boldsymbol{n}, \boldsymbol{l}_i \rangle \boldsymbol{n} - \boldsymbol{l}_i.$$



Rysunek 18.1. Wektory w modelach oświetlenia Phonga i Blinna-Phonga

Iloczyn skalarny wektorów jednostkowych $\boldsymbol{r}_i$ i $\boldsymbol{v}$ jest kosinusem kąta między tymi wektorami; im kąt ten jest mniejszy, tym bliżej prostej, wzdłuż której porusza się światło idealnie odbite, znajduje się obserwator. Patrząc z kierunków bliskich kierunku wektora $\boldsymbol{r}_i$, obserwator widzi odblask. Im lepiej powierzchnia jest wypolerowana, tym silniejszy jest ten odblask i jednocześnie tym szybciej zanika on w miarę, jak obserwator oddala się od kierunku idealnego odbicia. Wykładnik m określa szybkość zanikania odblasku (co na całym obrazie zakrzywionej powierzchni przekłada się na wielkość obszaru odblasku). Najczęściej w praktyce wykładnik ten jest rzędu kilkunastu lub kilkudziesięciu.

Czynnik v_i, jak poprzednio, ma wartość 1, jeśli punkt na powierzchni jest bezpośrednio oświetlony przez i-te źródło światła i 0 w przeciwnym razie, tj. gdy obserwator znajduje się po przeciwnej stronie powierzchni niż źródło światła lub jeśli między źródłem światła a danym punktem znajduje się obiekt rzucający cień (ale cieniami na razie jeszcze się nie zajmujemy).

Wektor $\boldsymbol{s}$ opisuje zdolność powierzchni do lustrzanego odbijania światła o różnych długościach fali; tak jak kolor $\boldsymbol{a}$ matowej farby (określającej zdolność do odbijania światła w sposób rozproszony) jest on trójką liczb opisujących odbijanie światła czerwonego, zielonego i niebieskiego. Można zaobserwować, że odblaski na wielu przedmiotach mają kolor zbliżony do koloru światła *padającego* na powierzchnię. Dlatego na ogół wektor ten jest inny niż wektor $\boldsymbol{a}$; jego współrzędne są najczęściej liczbami dodatnimi niewiele się różniącymi.

We wzorze występuje jeszcze funkcja W. Często przyjmuje się, że jest to funkcja stała, ale lepszy realizm można osiągnąć, biorąc funkcję kąta między wektorami $\boldsymbol{n}$ a $\boldsymbol{l}_i$ (można ją przedstawić jako funkcję jednej zmiennej — kosinusa tego kąta). Dla kąta 0 (tj. światła padającego prostopadle do powierzchni) funkcja ta powinna być równa 1, a ze wzrostem tego kąta jej wartość powinna rosnąć, a następnie maleć[2].

[2]Przedstawione tu modele oświetlenia Phonga i Blinna-Phonga spełniają opisaną w podrozdziale 28.1 zasadę Helmholtza, jeśli użyta w nich funkcja W jest iloczynem $|\langle \boldsymbol{l}_i, \boldsymbol{n} \rangle| Z(\boldsymbol{n}, \boldsymbol{l}_i, \boldsymbol{v})$, w którym czynnik $Z(\boldsymbol{n}, \boldsymbol{l}_i, \boldsymbol{v})$ jest funkcją symetryczną, tj. $Z(\boldsymbol{n}, \boldsymbol{l}_i, \boldsymbol{v}) = Z(\boldsymbol{n}, \boldsymbol{v}, \boldsymbol{l}_i)$ dla dowolnych wektorów $\boldsymbol{l}_i$ i $\boldsymbol{v}$.

W 1977 r. James Blinn [45] zmodyfikował model Phonga, otrzymując model trochę prostszy do implementacji. Wzór realizujący ten model ma postać

$$L = \boldsymbol{a} \sum_{i=0}^{n-1} \left(I_i^{\text{amb}} + I_i^{\text{dir}} v_i |\langle \boldsymbol{l}_i, \boldsymbol{n} \rangle| \right) + \boldsymbol{s} \sum_{i=0}^{n-1} I_i^{\text{dir}} v_i W(\boldsymbol{n}, \boldsymbol{l}_i, \boldsymbol{v}) \langle \boldsymbol{h}_i, \boldsymbol{n} \rangle^{2m}. \tag{18.2}$$

Zamiast kosinusa kąta między wektorami $\boldsymbol{r}_i$ a $\boldsymbol{v}$ oblicza się tu kosinus kąta między wektorami $\boldsymbol{h}_i$ a $\boldsymbol{n}$; wektor $\boldsymbol{h}_i$ ma kierunek dwusiecznej kąta między wektorami $\boldsymbol{l}_i$ a $\boldsymbol{v}$. Można go łatwo obliczyć, normalizując sumę $\boldsymbol{l}_i + \boldsymbol{v}$, tj. biorąc

$$\boldsymbol{h}_i = \frac{1}{\|\boldsymbol{l}_i + \boldsymbol{v}\|} (\boldsymbol{l}_i + \boldsymbol{v}).$$

Zauważmy, że gdyby wektory $\boldsymbol{l}_i$, $\boldsymbol{r}_i$ i $\boldsymbol{v}$ były współpłaszczyznowe, to wektor $\boldsymbol{h}_i$ też leżałby w ich płaszczyźnie, a kąt między wektorami $\boldsymbol{h}_i$ a $\boldsymbol{n}$ byłby dokładnie połową kąta między wektorami $\boldsymbol{r}_i$ a $\boldsymbol{v}$ (zobacz rys. 18.1). Aby przybliżyć czynnik $\langle \boldsymbol{r}_i, \boldsymbol{v} \rangle^m$ występujący w oryginalnym modelu Phonga, iloczyn skalarny $\langle \boldsymbol{h}_i, \boldsymbol{n} \rangle$ we wzorze (18.2) jest podnoszony do potęgi $2m$.

Oprócz odbijania światła powierzchnie mogą je też emitować. Aby zobrazować ten efekt, do intensywności światła odbitego możemy dodać składnik opisany wzorem

$$L_{\text{e}} = L_{\text{e}0} + \langle \boldsymbol{n}, \boldsymbol{v} \rangle^k L_{\text{e}1}. \tag{18.3}$$

Symbol $L_{\text{e}0}$ opisuje stałą intensywność światła wysyłanego we wszystkich kierunkach, a symbol $L_{\text{e}1}$ opisuje intensywność światła emitowanego prostopadle do powierzchni, zanikającego ze wzrostem kąta między wektorami $\boldsymbol{n}$ a $\boldsymbol{v}$. Wykładnik k określa szybkość tego zaniku. Obiekty rysowane przez aplikację 2C nie będą świecić, ale będą mogły — opisany dalej szader i współpracujące z nim procedury w C są na to gotowe.

18.2. Szadery

Użyjemy nowego szadera fragmentów, w którym będą zrealizowane oba modele oświetlenia, do wyboru przez aplikację. Pozostałe szadery zostawimy niezmienione. Wybór modelu oświetlenia zrealizujemy za pomocą zmiennej jednolitej typu `int`, na podstawie której instrukcja `switch` spowoduje wywołanie jednego z dwóch podprogramów realizujących różne modele: Lamberta i Blinna-Phonga[3]. Ponadto dodamy możliwość wybierania źródła opisu własności materiału. Dotychczas kolor powierzchni był podawany na wejście szadera fragmentów w strukturze `FVertex` (zobacz listingi 10.4 i 15.5). Pozostawimy tę możliwość. Na podstawie wartości kolejnej zmiennej jednolitej typu `int`, której zadaniem jest sterowanie wyborem, szader albo użyje tego koloru i nada pozostałym parametrom opisującym materiał wartości domyślne, albo przyjmie kompletny opis materiału podany w osobnym bloku zmiennych jednolitych.

[3] W pierwszym wydaniu użyłem wskaźników do podprogramów w GLSL-u. Opis, jak to zrobić i dlaczego moim zdaniem jednak nie warto, jest w p. 18.4.4.

Szader fragmentów jest przedstawiony na listingach 18.1–18.4. Szader ten powstał przez modyfikację szadera z listingu 12.8. Na pierwszym listingu są przedstawione globalne deklaracje typów i zmiennych; struktura wejściowa `FVertex`, zmienna wyjściowa `out_Colour` oraz bloki zmiennych jednolitych: `TransBlock` (opisującego przekształcenia) i `LSBlock` (opisującego śródła światła), skopiowane bez żadnych zmian, są podane w skrócie.

Listing 18.1. Szader fragmentów — deklaracje typów i zmienne globalne

GLSL

```
#version 420 core

#define MAX_MATERIALS 10

in FVertex { .... } In;  /* blok wejściowy taki jak na listingu 12.8 */
out vec4 out_Colour;     /* kolejne deklaracje takie jak na listingu 10.4 */
uniform TransBlock { .... } trb;
struct LSPar { .... };
uniform LSBlock { .... } light;

struct Material {
    vec4  emission0, emission1;  /* światło emitowane */
    vec3  diffref;               /* odbijanie rozproszone */
    vec3  specref;               /* odbijanie zwierciadlane światła */
    float shininess, wa, we;     /* parametry połysku */
  };

uniform MatBlock {
    int      mtn;
    Material mat[MAX_MATERIALS];
  } mat;

uniform int LightingModel = 0, ColourSource = 0;

Material mm;
vec3     normal, tnormal;
```

W liniach 11–16 jest zadeklarowany typ struktury `Material`, której pola opisują emisję światła oraz kolory farby rozpraszającej światło dochodzące z otoczenia i od źródła (`diffref`) jak też farby odbijającej światło w sposób zwierciadlany (`specref`). Pole `shininess` przechowuje wykładnik $2m$ we wzorze (18.2), a w polach `wa` i `we` są parametry a i e funkcji W, danej wzorem[4]

$$W(x) = x\big(1 + (a-1)(1-x)^e\big).$$

Blok `MatBlock` zawiera tablicę struktur typu `Material` i pole `mtn`, będące indeksem do tej tablicy wybierającym materiał, z którego jest wykonany obiekt rysowany w danej chwili. Zmienna jednolita `ColourSource` służy do wybierania źródła danych opisujących materiał.

[4] Funkcja ta jest dobrana „na wyczucie". Jej argument x jest kosinusem kąta między wektorami $\boldsymbol{l}_i$ a $\boldsymbol{n}$.

Zależnie od jej wartości na początku szader przypisze polom zmiennej mm zadeklarowanej w linii 25 wartości skopiowane z tablicy w bloku MatBlock albo wartości domyślne, w tym kolor farby wzięty ze struktury wejściowej FVertex.

Wspomniane wcześniej zmienne jednolite są zadeklarowane w linii 23. Mają one wartość początkową 0, a aplikacja będzie im nadawać wartości 1 i 0. Przedstawiona na listingu 18.2 procedura main przypisuje zmiennym normal i tnormal wektory normalne powierzchni i przybliżającego ją trójkąta, wywołuje procedurę GetMaterial, która przygotowuje opis materiału, wywołuje jedną z procedur realizujących model oświetlenia i przekazuje na wyjście obliczony przez tę procedurę kolor poddany korekcji gamma.

Listing 18.2. Szader fragmentów — procedury wspólne

GLSL

```
1: void GetMaterial ( void )
2: {
3:   switch ( ColourSource ) {
4: default:
5:     mm.emission0 = mm.emission1 = vec4 ( 0.0, 0.0, 0.0, 1.0 );
6:     mm.diffref = In.Colour;
7:     mm.specref = vec3 ( 0.25 );
8:     mm.shininess = 20.0;  mm.wa = mm.we = 1.0;
9:     break;
10: case 1:
11:     mm = mat.mat[mat.mtn];
12:     break;
13:   }
14: } /*GetMaterial*/
15:
16: vec3 posDifference ( vec4 p, vec3 pos, out float dist ) { .... }
17: float attFactor ( vec3 att, float dist ) { .... }
18:
19: void main ( void )
20: {
21:   vec3 Colour;
22:
23:   normal = normalize ( In.Normal );
24:   tnormal = In.TNormal;
25:   GetMaterial ();
26:   switch ( LightingModel ) {
27: default: Colour = LambertLighting ();     break;
28:  case 1: Colour = BlinnPhongLighting ();  break;
29:   }
30:   out_Colour = vec4 ( AGamma ( Colour ), 1.0 );
31: } /*main*/
```

Procedura GetMaterial nadaje wartości polom zmiennej mm. Jeśli zmienna jednolita ColourSource ma wartość 1, to następuje przypisanie danych z bloku MatBlock (o prywatnej nazwie mat), a jeśli ma dowolną inną wartość, to procedura przypisuje kolor przekazany

przez szader geometrii oraz domyślne wartości pozostałych parametrów. Do wyboru jest użyta instrukcja przełącznika (switch), ponieważ w przyszłości szader może zostać (i zostanie) rozbudowany o dodatkowe sposoby określania własności materiału, na przykład opisane przez teksturę.

Procedury pomocnicze posDifference i attFactor, używane przez procedury realizujące oba modele oświetlenia, są takie jak na listingu 10.4.

Listing 18.3 przedstawia zmienione instrukcje procedury LambertLighting. W porównaniu z procedurą z listingu 12.8 odwołania do zmiennej wejściowej In.Colour zostały zastąpione przez odwołania do pola diffref zmiennej mm. Jeśli taka jest wola aplikacji (a właściwie jej użytkownika), to procedura GetMaterial przypisze temu polu wartość zmiennej wejściowej In.Colour i obiekty będą mieć wygląd identyczny jak w poprzedniej aplikacji. W przeciwnym razie zostaną użyte parametry materiału z bloku zmiennych jednolitych MatBlock. Ponadto zmienna Colour zamiast zera otrzymuje wartość początkową opisującą światło emitowane przez powierzchnię. Wykładnik k we wzorze (18.3) jest podany w czwartej współrzędnej wektora mm.emission1.

Listing 18.3. Zmienione instrukcje procedury LambertLighting

GLSL

```
1: vec3 MatEmission ( float cosnv )
2: {
3:   if ( cosnv > 0.0 )
4:     return mm.emission0.rgb +
5:            mm.emission1.rgb * pow ( cosnv, mm.emission1.a );
6:   else
7:     return mm.emission0.rgb;
8: } /*MatEmission*/
9:
10: vec3 LambertLighting ( void )
11: {
12:   ....
13:   Colour = MatEmission ( dot ( normal, vv ) );
14:   for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
15:     if ( (light.mask & mask) != 0 ) {
16:       Colour += light.ls[i].ambient * mm.diffref;
17:       ....
18:           Colour += (d * light.ls[i].direct) * mm.diffref;
19:       ....
20:           Colour -= (d * light.ls[i].direct) * mm.diffref;
21:     }
22:   return clamp ( Colour, 0.0, 1.0 );
23: } /*LambertLighting*/
```

Na listingu 18.4 jest podana procedura realizująca model oświetlenia Blinna-Phonga, razem z pomocniczym podprogramem wFactor obliczającym wartość funkcji W we wzorze (18.2). Procedura BlinnPhongLighting oblicza składniki oświetlenia modelu lambertowskiego i dodatkowo (w liniach 27–29 oraz 39–41) składniki odpowiedzialne za odbicie

Listing 18.4. Szader fragmentów — model Blinna-Phonga

GLSL

```
float wFactor ( float lvn, float wa, float we )
{
  return lvn*(1.0+(wa-1.0)*pow ( 1.0-lvn, we ));
} /*wFactor*/

vec3 BlinnPhongLighting ( void )
{
  vec3  normal, lv, vv, hv, Colour;
  float a, d, e, f, dist;
  uint  i, mask;

  vv = posDifference ( trb.eyepos, In.Position, dist );
  e = dot ( vv, tnormal );
  Colour = MatEmission ( dot ( normal, vv ) );
  for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
    if ( (light.mask & mask) != 0 ) {
      Colour += light.ls[i].ambient * mm.diffref;
      lv = posDifference ( light.ls[i].position, In.Position, dist );
      d = dot ( lv, normal );
      if ( e > 0.0 ) {
        if ( d > 0.0 ) {
          if ( light.ls[i].position.w != 0.0 )
            a = attFactor ( light.ls[i].attenuation, dist );
          else
            a = 1.0;
          Colour += (a*d*light.ls[i].direct) * mm.diffref;
          hv = normalize ( lv+vv );
          f = pow ( dot ( hv, normal ), mm.shininess );
          Colour += (a*f*wFactor(d,mm.wa,mm.we)) * mm.specref;
        }
      }
      else {
        if ( d < 0.0 ) {
          if ( light.ls[i].position.w != 0.0 )
            a = attFactor ( light.ls[i].attenuation, dist );
          else
            a = 1.0;
          Colour -= (a*d*light.ls[i].direct) * mm.diffref;
          hv = normalize ( lv+vv );
          f = pow ( -dot ( hv, normal ), mm.shininess );
          Colour += (a*f*wFactor(-d,mm.wa,mm.we)) * mm.specref;
        }
      }
    }
  return clamp ( Colour, 0.0, 1.0 );
```

```
46: } /*BlinnPhongLighting*/
```

zwierciadlane. Pętla przebiegająca po wszystkich określonych i włączonych źródłach światła jest taka sama jak w procedurze `LambertLighting`. W liniach 27 i 39 jest obliczany wektor jednostkowy $\boldsymbol{h}_i$.

W liniach 28 i 40 jest obliczany czynnik $|\langle \boldsymbol{h}_i, \boldsymbol{n} \rangle|^{2m}$ i-tego składnika sumy intensywności światła odbitego w sposób zwierciadlany. Sam składnik jest obliczany i dodawany w liniach 29 i 41. Warunki sprawdzane w liniach 20, 21 i 33 wybierają odpowiednie instrukcje w zależności od tego, czy obserwator znajduje się po tej samej stronie płaszczyzny trójkąta i płaszczyzny stycznej do powierzchni, czy po przeciwnej niż i-te źródło światła, tak jak w procedurze dla modelu Lamberta. Końcowe obliczenie koloru fragmentu w linii 45 (obcinające współrzędne r, g, b „wystające" poza przedział $[0, 1]$) jest takie jak w procedurze `LambertLighting`.

Na listingu 18.5 są pokazane struktury do opisu bloku z opisami materiałów. Struktura typu `MatBl` zawiera identyfikator bufora z blokiem `MatBlock` oraz kopię tablicy opisów materiałów. Wartość pola `nmat` (początkowo 0) jest liczbą zdefiniowanych opisów. Długość tablicy może być zmieniona, ale trzeba dbać o to, aby makrodefinicja `MAX_MATERIALS` była taka sama jak w treści szadera[5].

Listing 18.5. Struktury do opisu bloku `MatBlock`

C

```
1: #define MAX_MATERIALS 10
2:
3: typedef struct Material {
4:     GLfloat emission0[4], emission1[4];
5:     GLfloat diffref[3];
6:     GLfloat specref[3];
7:     GLfloat shininess, wa, we;
8:   } Material;
9:
10: typedef struct MatBl {
11:     GLuint matbuf;
12:     int    nmat;
13:     Material mat[MAX_MATERIALS];
14:   } MatBl;
```

Listing 18.6 przedstawia procedury obsługi bloku z opisami materiałów. Dostęp do bloków z opisami źródeł światła i materiałów zapewnia procedura `GetAccessToLightMatUniformBlocks`, która z programu szaderów odczytuje przesunięcia pól w tych blokach i przydziela im punkty dowiązania, albo, jeśli to już zostało zrobione, przypisuje kolejnemu programowi odpowiednie numery punktów dowiązania.

[5] Opisy materiałów też mogą być zmieniane stosownie do potrzeb, na przykład w celu użycia modeli oświetlenia opisanych w podrozdziale 28.4. Ale zestaw procedur obsługi bloku może być zachowany — ważne jest, aby były one zebrane w jednym miejscu i aby miały monopol na przesyłanie danych do tego bloku, bo wtedy łatwiej jest unikać błędów podczas dokonywania modyfikacji.

Procedura NewUniformMatBlock rezerwuje bufor dla bloku MatBlock. Do wprowadzeia opisu materiału służy procedura SetupMaterial. Jej pierwszy parametr jest wskaźnikiem struktury MatBl, a drugi parametr jest numerem materiału. Jeśli jest on liczbą ujemną, to procedura, za pomocą AllocMaterialID, rezerwuje kolejne miejsce w tablicy; podając liczbę nieujemną, można zmienić parametry materiału wprowadzone wcześniej. Parametry materiału są zapisywane w tablicy w pamięci CPU i przesyłane do bufora w pamięci GPU. W instrukcji powrotu procedura przekazuje numer materiału. Procedura SetupMaterial przypisuje polom opisującym emisję domyślne wartości zerowe; jeśli materiał ma świecić, to trzeba dodatkowo wywołać procedurę SetMaterialEmission.

Listing 18.6. Procedury obsługi bloku z opisami materiałów

C

```
#define NLSOFFS   7
#define NMATUOFFS 9

static const GLchar *ULSNames[NLSOFFS+1] =
  { "LSBlock", "LSBlock.nls", "LSBlock.mask", "LSBlock.ls[0].ambient",
    "LSBlock.ls[0].direct", "LSBlock.ls[0].position",
    "LSBlock.ls[0].attenuation", "LSBlock.ls[1].ambient" };
static const GLchar *UMatNames[NMATUOFFS+1] =
  { "MatBlock", "MatBlock.mtn", "MatBlock.mat[0].emission0",
    "MatBlock.mat[0].emission1", "MatBlock.mat[0].diffref",
    "MatBlock.mat[0].specref", "MatBlock.mat[0].shininess",
    "MatBlock.mat[0].wa", "MatBlock.mat[0].we",
    "MatBlock.mat[1].diffref" };

static GLuint lsbbp = GL_INVALID_INDEX, matbbp = GL_INVALID_INDEX;
static GLint  lsbsize, lsbofs[NLSOFFS], matbsize, matbofs[NMATUOFFS];

void GetAccessToLightMatUniformBlocks ( GLuint program_id )
{
  if ( lsbbp == GL_INVALID_INDEX )
    GetAccessToUniformBlock ( program_id, NLSOFFS, &ULSNames[0],
                              &lsbsize, lsbofs, &lsbbp );
  else
    AttachUniformBlockToBP ( program_id, ULSNames[0], lsbbp );
  if ( matbbp == GL_INVALID_INDEX )
    GetAccessToUniformBlock ( program_id, NMATUOFFS, &UMatNames[0],
                              &matbsize, matbofs, &matbbp );
  else
    AttachUniformBlockToBP ( program_id, UMatNames[0], matbbp );
} /*GetAccessToLightMatUniformBlocks*/

GLuint NewUniformMatBlock ( void )
{
  return NewUniformBuffer ( matbsize, matbbp );
```

```
35: } /*NewUniformMatBlock*/
36:
37: int AllocMaterialID ( MatBl *mat )
38: {
39:   if ( mat->nmat >= MAX_MATERIALS )
40:     ExitOnError ( "AllocMaterialID" );
41:   return mat->nmat ++;
42: } /*AllocMaterialID*/
43:
44: int SetupMaterial ( MatBl *matbl, int m, const GLfloat diffr[3],
45:                 const GLfloat specr[3], GLfloat shn, GLfloat wa, GLfloat we )
46: {
47:   GLint ofs;
48:   GLfloat em[4] = { 0.0, 0.0, 0.0, 1.0 };
49:
50:
51:   if ( m < 0 )
52:     m = AllocMaterialID ( matbl );
53:   if ( m < MAX_MATERIALS ) {
54:     memcpy ( matbl->mat[m].diffref, diffr, 4*sizeof(GLfloat) );
55:     memcpy ( matbl->mat[m].specref, specr, 4*sizeof(GLfloat) );
56:     matbl->mat[m].shininess = shn;
57:     matbl->mat[m].wa = wa;
58:     matbl->mat[m].we = we;
59:     ofs = m*(matbofs[NMATUOFFS-1]-matbofs[1]);
60:     glBindBuffer ( GL_UNIFORM_BUFFER, matbl->matbuf );
61:     glBufferSubData ( GL_UNIFORM_BUFFER, ofs+matbofs[1],
62:                       4*sizeof(GLfloat), em );
63:     glBufferSubData ( GL_UNIFORM_BUFFER, ofs+matbofs[2],
64:                       4*sizeof(GLfloat), em );
65:     glBufferSubData ( GL_UNIFORM_BUFFER, ofs+matbofs[3],
66:                       3*sizeof(GLfloat), diffr );
67:     glBufferSubData ( GL_UNIFORM_BUFFER, ofs+matbofs[4],
68:                       3*sizeof(GLfloat), specr );
69:     glBufferSubData ( GL_UNIFORM_BUFFER, ofs+matbofs[5],
70:                       sizeof(GLfloat), &shn );
71:     glBufferSubData ( GL_UNIFORM_BUFFER, ofs+matbofs[6],
72:                       sizeof(GLfloat), &wa );
73:     glBufferSubData ( GL_UNIFORM_BUFFER, ofs+matbofs[7],
74:                       sizeof(GLfloat), &we );
75:     ExitIfGLError ( "SetupMaterial" );
76:   }
77:   return m;
78: } /*SetupMaterial*/
79:
80: void SetMaterialEmission ( MatBl *matbl, int m,
81:                            const GLfloat em0[4], const GLfloat em1[4] )
82: {
```

```
  GLint ofs;

  if ( m >= 0 && m < matbl->nmat ) {
    ofs = m*(matbofs[NMATUOFFS-1]-matbofs[1]);
    glBindBuffer ( GL_UNIFORM_BUFFER, matbl->matbuf );
    memcpy ( matbl->mat[m].emission0, em0, 4*sizeof(GLfloat) );
    memcpy ( matbl->mat[m].emission1, em1, 4*sizeof(GLfloat) );
    glBufferSubData ( GL_UNIFORM_BUFFER, ofs+matbofs[1],
                      4*sizeof(GLfloat), em0 );
    glBufferSubData ( GL_UNIFORM_BUFFER, ofs+matbofs[2],
                      4*sizeof(GLfloat), em1 );
    ExitIfGLError ( "SetMaterialEmission" );
  }
} /*SetMaterialEmission*/

void ChooseMaterial ( MatBl *matbl, GLint m )
{
  if ( m < 0 || m >= matbl->nmat )
    ExitOnError ( "ChooseMaterial" );
  glBindBuffer ( GL_UNIFORM_BUFFER, matbl->matbuf );
  glBufferSubData ( GL_UNIFORM_BUFFER, matbofs[0], sizeof(GLint), &m );
  ExitIfGLError ( "ChooseMaterial" );
} /*ChooseMaterial*/
```

Aby wybrać materiał, z którego ma być wykonany obiekt, wystarczy przed jego rysowaniem wywołać procedurę `ChooseMaterial`, która przypisze podany numer materiału polu `mtn` w bloku `MatBlock`.

18.3. Zmiany w aplikacji

Listing 18.7 przedstawia strukturę danych aplikacji z wyodrębnionym opakowaniem programów używanych do rysowania; oprócz tablicy z identyfikatorami programów opakowanie to zawiera położenia zmiennych jednolitych `LightingModel` i `ColourSource` oraz bieżące wartości tych zmiennych. Struktura `AppData`, która jest opakowaniem zmiennych części graficznej aplikacji, ma dwa dodatkowe pola: strukturę `mat` typu `MatBl` i opakowanie programów szaderów.

Listing 18.7. Zmiany opakowania danych

C

```
typedef struct {
    GLuint program_id[2];
    GLint  LightingModelLoc, lighting_model,
           ColourSourceLoc, colour_source;
  } BPRenderPrograms;

```

```
typedef struct {
    Camera           camera;
    BezierPatchObjf  *myteapot, *mytorus;
    TransBl          trans;
    LightBl          light;
    MatBl            mat;
    GLint            BezNormals, TessLevel;
    char             cnet, skeleton, animate;
    float            model_rot_axis[3];
    double           teapot_rot_angle, torus_rot_angle;
    GLfloat          teapot_mmatrix[16], torus_mmatrix[16];
    BPRenderPrograms brprog;
  } AppData;
```

Na listingu 18.8 jest pokazana procedura kompilacji szaderów, która łączy je w programy używane do rysowania płatów i siatek kontrolnych. W porównaniu z aplikacją 2B pierwszy program ma nowy (opisany w poprzednim podrozdziale) szader fragmentów, zapisany w pliku `app2c.frag.glsl`, a drugi program jest niezmieniony. Z pierwszego programu procedura musi odczytać więcej informacji, aby można go było używać.

Listing 18.8. Procedura kompilacji szaderów

C

```
void LoadBPShaders ( BPRenderPrograms *brprog )
{
  static const char *filename[] =
    { "app2.vert.glsl", "app2.tesc.glsl", "app2.tese.glsl",
      "app2.geom.glsl", "app2c.frag.glsl",
      "app2a1.vert.glsl", "app2a1.frag.glsl" };
  static const GLuint shtype[7] =
    { GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER,
      GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER,
      GL_VERTEX_SHADER, GL_FRAGMENT_SHADER };
  static const GLchar *UVarNames[] = { "ColourSource", "LightingModel" };
  GLuint shader_id[7];
  int    i;

  for ( i = 0; i < 7; i++ )
    shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
  brprog->program_id[0] = LinkShaderProgram ( 5, shader_id, "0" );
  brprog->program_id[1] = LinkShaderProgram ( 2, &shader_id[5], "1" );
  GetAccessToTransBlockUniform ( brprog->program_id[0] );
  GetAccessToLightMatUniformBlocks ( brprog->program_id[0] );
  GetAccessToBezPatchStorageBlocks ( brprog->program_id[0], false, false );
  brprog->ColourSourceLoc =
      glGetUniformLocation ( brprog->program_id[0], UVarNames[0] );
  brprog->LightingModelLoc =
      glGetUniformLocation ( brprog->program_id[0], UVarNames[1] );
```

```
26:   brprog->lighting_model = brprog->colour_source = 0;
27:   AttachUniformTransBlockToBP ( brprog->program_id[1] );
28:   for ( i = 0; i < 7; i++ )
29:     glDeleteShader ( shader_id[i] );
30:   ExitIfGLError ( "LoadBPShaders" );
31: } /*LoadBPShaders*/
```

Po skompilowaniu szaderów i złączeniu programów procedura `LoadBPShaders` uzyskuje dostęp do bloku zmiennych jednolitych `TransBlock`), a potem wywołuje przedstawioną na listingu 18.6 procedurę `GetAccessToLightMatUniformBlocks`, która podobnie daje dostęp do bloków z opisami źródeł światła i materiału. W liniach 22–25 są odczytywane z programu i zapamiętywane **położenia** zmiennych jednolitych `LightingModel` i `ColourSource`, które znajdują się w tzw. **domyślnym bloku zmiennych jednolitych programu**[6]. Wartości takim zmiennym przypisuje się za pomocą procedur z rodziny `glUniform*`, na przykład do nadania wartości zmiennej typu `int` (a także `bool`) służy procedura `glUniform1i`. Deklaracje zmiennych `LightingModel` i `ColourSource` nadają im wartości początkowe; zmienne `lighting_model` i `colour_source` otrzymują te same wartości w linii 26.

Procedura `InitMyWorld` (listing 18.9) zamiast tablicy, do której mają trafić identyfikatory programów, przekazuje procedurze kompilacji adres zmiennej `appdata.brprog`, a potem tworzy bufor z przeznaczeniem na blok z opisami materiałów. Procedury wprowadzające opisy obiektów (czajnika i torusa) wprowadzają też opisy materiałów dla tych obiektów. Materiały te otrzymują kolejne numery `0` i `1`.[7]

Listing 18.9. Procedury konstrukcji czajnika i torusa oraz procedura inicjalizacji

C

```
1: void ConstructMyTeapot ( AppData *ad )
2: {
3:   const GLfloat MyColour[4] = { 1.0, 1.0, 1.0, 1.0 };
4:   const GLfloat diffr[3] = { 0.75, 0.6, 0.2 };
5:   const GLfloat specr[3] = { 0.7, 0.7, 0.6 };
6:   const GLfloat shn = 60.0, wa = 5.0, we = 5.0;
7:
8:   ad->myteapot = ConstructTheTeapot ( MyColour );
9:   SetBezierPatchTessLevel ( ad->myteapot, ad->TessLevel );
10:  SetBezierPatchNVS ( ad->myteapot, ad->BezNormals );
11:  SetupMaterial ( &ad->mat, -1, diffr, specr, shn, wa, we );
12: } /*ConstructMyTeapot*/
```

[6] Szadery przeznaczone do współpracy z aplikacjami standardu Vulkan *nie mogą* mieć domyślnego bloku zmiennych jednolitych; wszystkie zmienne jednolite muszą być umieszczone w buforach widocznych dla tych szaderów jako nazwane bloki zmiennych jednolitych.

[7] W bardziej skomplikowanych aplikacjach, na przykład takich, w których zbiór obiektów ulega zmianie podczas działania, należałoby zapamiętywać numery materiałów w reprezentacjach obiektów, a ponadto rozbudować podsystem rezerwowania i zwalniania miejsc w tablicy materiałów. Zostawiam to do zrobienia Czytelnikom chcącym tworzyć takie aplikacje.

```
13:
14: void ConstructMyTorus ( AppData *ad )
15: {
16:   GLfloat MyColour[4] = { 0.2, 0.3, 1.0, 1.0 };
17:   const GLfloat diffr[3] = { 0.0, 0.4, 1.0 };
18:   const GLfloat specr[3] = { 0.7, 0.7, 0.7 };
19:   const GLfloat shn = 20.0, wa = 2.0, we = 5.0;
20:
21:   ad->mytorus = EnterTorus ( 1.0, 0.5, MyColour );
22:   SetBezierPatchTessLevel ( ad->mytorus, ad->TessLevel );
23:   SetBezierPatchNVS ( ad->mytorus, ad->BezNormals );
24:   SetupMaterial ( &ad->mat, -1, diffr, specr, shn, wa, we );
25: } /*ConstructMyTorus*/
26:
27: void InitMyWorld ( int argc, char *argv[], int width, int height )
28: {
29:   float axis[4] = {0.0,0.0,1.0};
30:
31:   memset ( &appdata, 0, sizeof(AppData) );
32:   LoadBPShaders ( &appdata.brprog );
33:   appdata.trans.trbuf = NewUniformTransBlock ();
34:   appdata.light.lsbuf = NewUniformLightBlock ();
35:   appdata.mat.matbuf = NewUniformMatBlock ();
36:   .... /* pozostałe instrukcje bez zmian */
37: } /*InitMyWorld*/
```

Procedury rysujące są przedstawione na listingu 18.10. Przed wywołaniem procedury `DrawBezierPatches` jest wybierany materiał za pomocą procedury `ChooseMaterial` z listingu 18.6.[8]

Listing 18.10. Procedury rysowania czajnika i torusa

C

```
1: void DrawMyTeapot ( AppData *ad )
2: {
3:   if ( ad->skeleton )
4:     glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
5:   else
6:     glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
7:   glUseProgram ( ad->brprog.program_id[0] );
8:   ChooseMaterial ( &ad->mat, 0 );
9:   DrawBezierPatches ( ad->myteapot );
```

[8] W pierwszym wydaniu książki w bloku `MatBlock` umieściłem tylko jeden opis materiału. Zmiana materiału polegała na dowiązaniu innego bufora. Podczas rysowania program szaderów musi mieć dostęp do tablicy z opisami *wszystkich* źródeł światła i tylko do jednego opisu materiału. Ale indeks do tablicy materiałów można uczynić atrybutem wierzchołka, co pozwoliłoby rysować jednocześnie stanowiące części jednego obiektu trójkąty „wykonane" z różnych materiałów. Obecne rozwiązanie jest więc bardziej elastyczne.

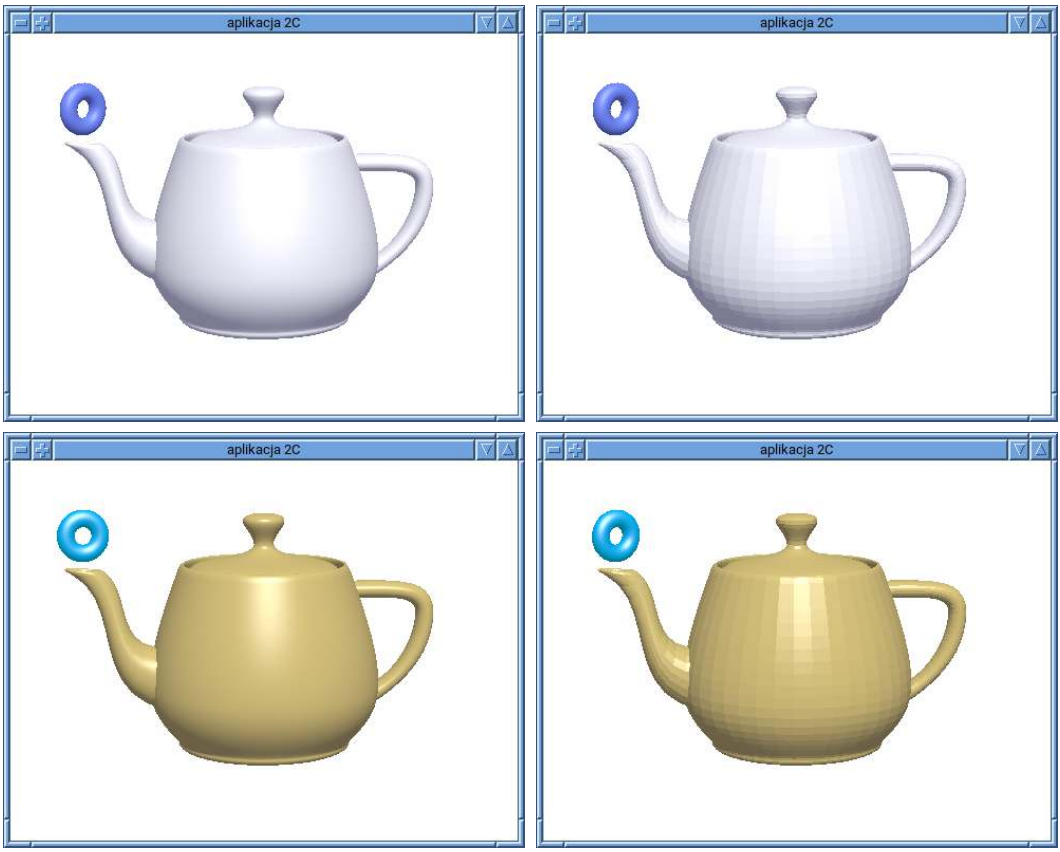
```
} /*DrawMyTeapot*/

void DrawMyTorus ( AppData *ad )
{
  if ( ad->skeleton )
    glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
  else
    glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
  glUseProgram ( ad->brprog.program_id[0] );
  ChooseMaterial ( &ad->mat, 1 );
  DrawBezierPatches ( ad->mytorus );
} /*DrawMyTorus*/
```

Listing 18.11. Zmiany w procedurze `ProcessCharCommand`

C

```
char ProcessCharCommand ( char charcode )
{
  switch ( toupper ( charcode ) ) {
    .... /* dotychczasowe instrukcje bez zmian */
case 'B':
    glUseProgram ( appdata.brprog.program_id[0] );
    glUniform1i ( appdata.brprog.LightingModelLoc,
          appdata.brprog.lighting_model = !appdata.brprog.lighting_model );
    glUseProgram ( 0 );
    return true;
case '0':
    glUseProgram ( appdata.brprog.program_id[0] );
    glUniform1i ( appdata.brprog.ColourSourceLoc,
                  appdata.brprog.colour_source = 0 );
    glUseProgram ( 0 );
    return true;
case '1':
    glUseProgram ( appdata.brprog.program_id[0] );
    glUniform1i ( appdata.brprog.ColourSourceLoc,
                  appdata.brprog.colour_source = 1 );
    glUseProgram ( 0 );
    return true;
default:
    break;
  }
  return false;
} /*ProcessCharCommand*/
```

Listing 18.11 przedstawia instrukcje dodane do procedury `ProcessCharCommand`, która wykonuje dodatkowe trzy polecenia wydawane za pomocą klawiatury. Napisanie litery `B` powoduje przełączenie modelu oświetlenia — z modelu Lamberta na model Blinna-Phonga lub



Rysunek 18.2. Efekt użycia modelu oświetlenia Blinna-Phonga

w drugą stronę. Polega to na przypisaniu zmiennej `LightingModel` wartości `1` albo `0`, przy czym przed wywołaniem procedury, która to robi (`glUniform1i`), należy (przy użyciu procedury `glUseProgram`) wybrać program, w którego domyślnym bloku zmiennych jednolitych ta zmienna się znajduje. Przypisanie zmiennej `redraw` sygnalizuje, że trzeba wykonać nowy obraz[9]. Napisanie cyfry `0` lub `1` powoduje przypisanie (w podobny sposób) wartości 0 albo 1 zmiennej jednolitej `ColourSource`; w pierwszym przypadku czajnik będzie miał kolor pamiętany w opisie płatów Béziera (czyli biały), a w drugim będzie wykonany z materiału opisanego w bloku `MatBlock`; ta sama zmienna wybiera też kolor torusa.

Do procedury `DeleteMyWorld` trzeba tylko dopisać wywołanie procedury `glDeleteBuffers`, aby zwolnić pamięć zajmowaną przez bufory z opisami materiałów, o czym piszę na wszelki wypadek, choć chciałbym mieć nadzieję, że już nie muszę.

Okno aplikacji z obrazami wykonanymi przy użyciu modelu Blinna-Phonga jest pokazane na rysunku 18.2. Z lewej strony są obrazy, dla których wektory normalne zostały

[9] To nie ma znaczenia, bo animacja w tej aplikacji trwa cały czas; czajnik może być nieruchomy, ale torus się obraca.

obliczone przez szader rozdrabniania (a następnie poddane interpolacji), a z prawej strony wektory normalne obliczył szader geometrii (i też zostały one poddane interpolacji, która dla każdego punktu trójkąta wytworzyła ten sam wektor normalny)[10].

18.4. Uzupełnienia

18.4.1. Test nożyczek

Można ograniczyć rysowanie w oknie do dowolnego prostokąta określonego niezależnie od klatki; sprawdzenie, czy dany fragment (piksel) jest w tym prostokącie, nazywa się **testem nożyczek** (*scissor test*). Aby określić ten prostokąt, należy wywołać procedurę

```
glScissor ( x, y, w, h );
```

której parametry określają współrzędne (w układzie okna OpenGL-a) dolnego lewego wierzchołka oraz szerokość i wysokość prostokąta w pikselach. Uaktywnienie testu następuje przez wykonanie instrukcji

```
glEnable ( GL_SCISSOR_TEST );
```

a do jego wyłączenia służy procedura `glDisable` (z parametrem jak wyżej).

18.4.2. Wczesne testy fragmentu

Domyślnie szader fragmentów jest wywoływany *przed* wykonaniem testów: nożyczek, maski[11] i widoczności, które mogą spowodować odrzucenie fragmentu. Jeśli obliczenie wykonywane przez szader fragmentów zabiera znacząco dużo czasu — a to może mieć miejsce w przypadku stosowania porządnego modelu oświetlenia i będzie miało miejsce w dalszych wersjach aplikacji, nakładających tekstury na obiekty — a potem fragment okaże się niewidoczny (bo inny, przetworzony wcześniej lub później fragment odwzorowany na ten sam piksel ma mniejszą głębokość lub test maski albo nożyczek spowoduje odrzucenie fragmentu), to czas zużyty przez szader na przetwarzanie bieżącego fragmentu jest zmarnowany. Dlatego można zażądać, by wspomniane testy były wykonane przed wywołaniem szadera fragmentów, dzięki czemu szader nie będzie wywoływany dla fragmentów, które nie przeszły któregoś z testów. Żądanie to zgłasza się, podając kwalifikator wejścia szadera fragmentów postaci

```
layout(early_fragment_tests) in;
```

[10] Aby nie wykonywać interpolacji, co zabiera czas, można by tu użyć kwalifikatora `flat` dla wektora normalnego (zobacz p. 12.4.5). Ale wtedy trzeba by utworzyć osobny program szaderów do wyświetlania obrazu płaskich trójkątów, bo w celu otrzymania obrazu powierzchni gładkiej szader geometrii wyprowadza dla wierzchołków trójkąta różne wektory normalne, które trzeba interpolować. W programie rysującym tylko płaskie trójkąty szader rozdrabniania mógłby nie obliczać wektorów normalnych.

[11] Test sprawdzający, czy piksel nie jest w obszarze zabronionym do rysowania, reprezentowanym przez zawartość bufora maski (*stencil buffer*), o którym na razie nie napisałem.

Dlaczego w domyślnej kolejności szader fragmentów jest wywoływany przed wspomnianymi testami? Bo szader fragmentów może, w razie potrzeby (w celu wpłynięcia na wynik testu widoczności), zmienić głębokość fragmentu przez przypisanie wartości zmiennej wbudowanej `gl_FragDepth`[12]. Gdyby test widoczności został wykonany wcześniej, nowa głębokość byłaby fragmentowi przypisana poniewczasie.

18.4.3. Oświetlenie hemisferyczne

Jeśli obiekt jest oświetlony przez jedno punktowe źródło światła, to część obiektu odwrócona tyłem do tego źródła jest oświetlona tylko przez rozproszone w otoczeniu światło, o którym zakładamy, że jest bezkierunkowe, tj. z każdego kierunku takie samo. Efekt jest kompletnie nieplastyczny: zamiast detali kształtu obiektu na obrazie widać tylko plamę w jednolitym kolorze. Aby uzyskać lepszy obraz, można „włączyć" kilka źródeł, oświetlających przedmiot z różnych stron. Inna metoda polega na wprowadzeniu dochodzącego ze wszystkich stron światła rozproszonego, którego intensywność[13] zmienia się z kierunkiem, z jakiego światło to dochodzi.

Najprostszy model takiego światła to **oświetlenie hemisferyczne**; sferę jednostkową, której punkty reprezentują kierunki, podzielimy na dwie półsfery. Światło padające z kierunków górnej półsfery ma intensywność i kolor „nieba" (czyli jest dosyć jasne i białe lub niebieskie o małym nasyceniu), a światło dochodzące od dołu jest znacznie słabsze i ma kolor „gruntu", na którym stoi obiekt (może to być też kolor trawy). Zakładamy, że intensywność i kolor światła w każdej półsferze są stałe. Rysunek 18.3 przedstawia sferę jednostkową, w środku której znajduje się fragment powierzchni obiektu. Zaznaczone są wektory jednostkowe $\boldsymbol{z}$ i $\boldsymbol{n}$; pierwszy z nich ma kierunek zenitu, a drugi jest wektorem normalnym powierzchni. Okręgi jednostkowe H i P leżą w płaszczynach prostopadłych do wektorów $\boldsymbol{z}$ i $\boldsymbol{n}$.[14]

Załóżymy, że rozpatrywane tu światło odbija się od powierzchni przedmiotu „po lambertowsku", tj. wkład w końcowy kolor piksela na obrazie światła, które dochodzi do powierzchni z pewnego kierunku, jest proporcjonalny do kosinusa kąta padania. Przy tych założeniach możemy obliczyć współczynnik t opisujący proporcję, w jakiej światła pochodzące z górnej i dolnej półsfery są zmieszane po odbiciu od powierzchni. Do prawej strony wzoru (10.1), (18.1) lub (18.2) dodamy składnik

$$\boldsymbol{a}\big((1-t)I^{\text{sky}} + tI^{\text{ground}}\big), \tag{18.4}$$

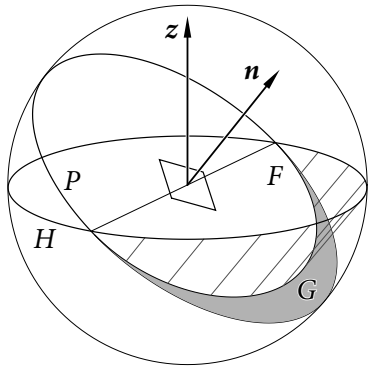
w którym wektor $\boldsymbol{a}$ opisuje kolor powierzchni, a wektory I^{sky} oraz I^{ground} reprezentują intensywności świateł dochodzących z obu półsfer. W zasadzie można też użyć tylko tego składnika, tj. pominąć punktowe źródła światła obecne we wcześniej rozpatrywanych modelach.

Całkowita intensywność światła odbitego od powierzchni jest iloczynem współczynnika $\boldsymbol{a}$ i całki po jednej z dwóch półsfer rozdzielonych okręgiem P położonym w płaszczyźnie

[12] Głębokość fragmentu obliczona w etapie rasteryzacji jest wartością zmiennej wbudowanej `gl_FragCoord.z`; zmienne `gl_FragCoord.x` i `gl_FragCoord.y` zawierają współrzędne punktu w oknie, a wartość zmiennej `gl_FragCoord.w` jest *odwrotnością* współrzędnej wagowej punktu w układzie kostki standardowej.

[13] Zobacz przypis na s. 215 i rozdział 28.

[14] Okrąg H odpowiada horyzontowi; zenit to kierunek wektorów prostopadłych do jego płaszczyzny i zorientowanych w stronę „nieba".



Rysunek 18.3. Model oświetlenia hemisferycznego

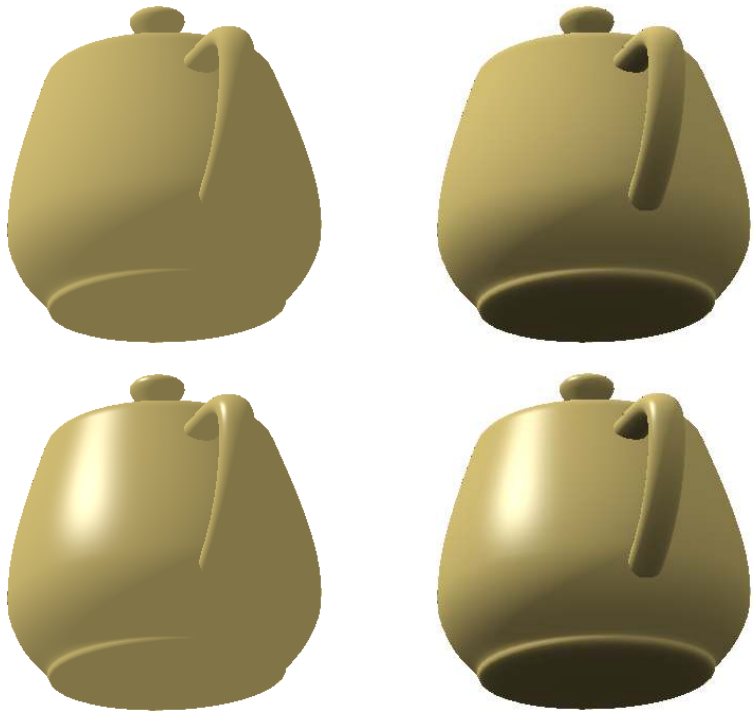
stycznej do powierzchni — wybieramy oczywiście tę półsferę, której elementem jest wektor $\boldsymbol{v}$ mający kierunek do obserwatora, czyli tę, której punkty reprezentują kierunki, z których dochodzi światło padające na stronę powierzchni widzianą przez obserwatora. Funkcja podcałkowa to intensywność światła dochodzącego z kierunku $\boldsymbol{l}$ pomnożona przez kosinus kąta między wektorami $\boldsymbol{l}$ a $\boldsymbol{n}$ (co odpowiada modelowi Lamberta odbicia światła) [15].

Rozważmy tak mały fragment odpowiedniej półsfery, aby jego punkty były wektorami o *prawie tym samym* kierunku wektora (jednostkowego) $\boldsymbol{l}$. Pole fragmentu pomnożone przez kosinus kąta między wektorami $\boldsymbol{l}$ a $\boldsymbol{n}$ jest równe polu rzutu prostopadłego tego fragmentu na płaszczyznę styczną do powierzchni. Popatrzmy na rysunek 18.3. Rzuty wszystkich fragmentów półsfery, po której całkujemy, wypełniają koło o brzegu P. Jeśli $\langle \boldsymbol{v}, \boldsymbol{n} \rangle > 0$, tj. obserwator znajduje się po wskazywanej przez wektor $\boldsymbol{n}$ stronie płaszczyzny stycznej do powierzchni, to rzutem przecięcia górnej półsfery (o brzegu H) i półsfery kierunków oświetlających tę stronę powierzchni jest figura F, której brzeg składa się z połowy okręgu P i połowy elipsy będącej rzutem prostopadłym okręgu H na płaszczyznę styczną do powierzchni. Pozostałą część koła (figurę G, narysowaną w kolorze szarym) wypełniają rzuty odpowiednich fragmentów dolnej półsfery. W ten sposób obliczenie wspomnianej całki sprowadzone zostało do znalezienia pól figur F i G. Współczynnik t we wzorze (18.4) jest ilorazem pola figury G i pola całego koła[16]. Możemy zauważyć, że jest on równy $(1 - \cos \vartheta)/2$, gdzie ϑ jest kątem między wektorami $\boldsymbol{z}$ a $\boldsymbol{n}$ (zatem $t = (1 - \langle \boldsymbol{z}, \boldsymbol{n} \rangle)/2$). Jeśli obserwator znajduje się po przeciwnej stronie powierzchni, to proporcja podziału koła na części składające się z rzutów odpowiednich fragmentów górnej i dolnej półsfery jest odwrotna. Do wzoru (18.4) należy zatem podstawić

$$t = \begin{cases} (1 - \cos \vartheta)/2 = (1 - \langle \boldsymbol{z}, \boldsymbol{n} \rangle)/2 & \text{jeśli } \langle \boldsymbol{v}, \boldsymbol{n} \rangle > 0, \\ (1 + \cos \vartheta)/2 = (1 + \langle \boldsymbol{z}, \boldsymbol{n} \rangle)/2 & \text{jeśli } \langle \boldsymbol{v}, \boldsymbol{n} \rangle \leqslant 0. \end{cases}$$

[15] „Intensywność" to radiancja światła dochodzącego z otoczenia; pomnożona przez kosinus kąta padania daje gęstość mocy światła zwaną irradiancją, zobacz rozdział 28.

[16] pole to jest oczywiście równe π



Rysunek 18.4. Obrazy otrzymane bez i z oświetleniem hemisferycznym

Stosując model oświetlenia Phonga lub Blinna-Phonga, możemy w składniku opisującym odblask uwzględnić tylko światło dochodzące bezpośrednio od punktowego źródła światła, ponieważ całka, którą trzeba by obliczyć dla odbitego w sposób zwierciadlany światła dochodzącego z odpowiedniej półsfery jest dość trudna do obliczenia (należałoby użyć kwadratury), a opisany przez nią efekt na obrazie zazwyczaj jest dość słaby.

Rysunek 18.4 umożliwia porównanie skutków użycia oświetlenia hemisferycznego (na obrazkach z prawej strony) z oświetleniem bezkierunkowym; obrazki umieszczone wyżej są wykonane przy użyciu modelu Lamberta, a te niżej przy użyciu modelu Blinna-Phonga, przy czym odblask jest otrzymany tylko dla światła padającego z jednego kierunku.

18.4.4. *Wskaźniki do procedur w GLSL-u

Zamiast wybierać wywoływaną procedurę za pomocą instrukcji przełącznika, można użyć zmiennej wskaźnikowej do procedury. Musi to być zmienna jednolita, której wartość będzie nadawać aplikacja. Na przykładzie aplikacji 2C zobaczmy, jak to się robi.

Szader fragmentów[17] zawiera deklaracje pokazane na listingu 18.12 oraz nagłówki procedur wskazywanych. W linii 3 jest deklaracja typu zmiennej wskaźnikowej, a w linii 4 zmienna ta jest zadeklarowana. Nagłówki procedur, które mają być wskazywane, muszą być poprze-

[17] opisany w pierwszym wydaniu

dzone słowem kluczowym subroutine i nazwą typu w nawiasie. Składnia wywołania procedury wskazywanej jest taka jak każdej innej procedury — po nazwie (zmiennej wskaźnikowej) podaje się listę parametrów.

Listing 18.12. Szader fragmentów z procedurami wskazywanymi

GLSL

```
1: #version 430 core
2: ....
3: subroutine vec3 LightingProc ( void );
4: layout(location=0) subroutine uniform LightingProc Lighting;
5:
6: subroutine (LightingProc) vec3 LambertLighting ( void ) { .... }
7: subroutine (LightingProc) vec3 BlinnPhongLighting ( void ) { .... }
8:
9: void main ( void )
10: {
11:   normal = normalize ( In.Normal );
12:   tnormal = In.TNormal;
13:   GetMaterial ();
14:   out_Colour = vec4 ( AGamma ( Lighting () ), 1.0 );
  } /*main*/
```

Bardziej skomplikowane instrukcje trzeba napisać w języku C. Po skompilowaniu i złączeniu programu szaderów trzeba z niego odczytać i zapamiętać w zmiennych typu GLint położenie zmiennej wskaźnikowej oraz indeksy procedur wskazywanych (listing 18.13).

Listing 18.13. Odczytywanie informacji o wskaźnikach z programu szaderów

C

```
1:   LightProcLoc = glGetSubroutineUniformLocation ( program_id,
2:                       GL_FRAGMENT_SHADER, "Lighting" );
3:   LambertProcInd = glGetSubroutineIndex ( program_id,
4:                         GL_FRAGMENT_SHADER, "LambertLighting" );
5:   BlinnPhongProcInd = glGetSubroutineIndex ( program_id,
6:                             GL_FRAGMENT_SHADER, "BlinnPhongLighting" );
```

Zmiennym wskaźnikowym trzeba nadać wartości po *każdym* wywołaniu procedury glUseProgram przed rysowaniem; służy do tego procedura glUniformSubroutinesuiv, która ma trzy parametry: identyfikator szadera w programie (np. GL_FRAGMENT_SHADER), liczbę zmiennych wskaźnikowych i tablicę indeksów procedur, które mają być wskazywane przez te zmienne. Przy tym wartości trzeba nadać *wszystkim* zmiennym wskaźnikowym; kolejne elementy tablicy określają procedury wskazywane przez zmienne o kolejnych położeniach (zaczynając od 0). Jeśli zmiennych jest więcej niż jedna, to warto jawnie nadać im położenia (przy użyciu kwalifikatora layout, jak w linii 4), aby nie musieć zgadywać, jakie położenia nadał im kompilator GLSL-a.

Inaczej niż w przypadku „zwykłych” procedur, wszystkie procedury wskazywane muszą być w tym samym szaderze, co zmienne wskazujące[18]. Jak widać, używanie wskaźników do procedur w GLSL-u jest dość kłopotliwe i chyba niewarte polecenia — jedyny zysk to uproszczenie samego wywołania procedur.

18.5. Ćwiczenia

1. Napisz procedurę realizującą model oświetlenia Phonga opisany wzorem (18.1).
2. Wykonaj eksperymenty polegające na zmienianiu parametrów materiału i oglądaniu skutków.
3. Rozbuduj szader fragmentów tak, aby można było „pomalować” różnymi „farbami” dwie strony powierzchni. Wybór „farby” (czyli materiału) powinien być dokonany na podstawie znaku iloczynu skalarnego wektora do obserwatora i wektora normalnego, $\langle \boldsymbol{v}, \boldsymbol{n} \rangle$. Iloczyn ten jest obliczany na przykład w linii 13 na listingu 18.4.
4. Rozbuduj aplikację tak, aby można było za pomocą jednego wywołania procedury `DrawBezierPatches` wykonać obraz wielu egzemplarzy czajnika, na przykład podobny do obrazu na rysunku 15.7, przy czym każdy czajnik ma być wykonany z innego materiału.
5. Rozbuduj aplikację tak, aby można było wybierać model oświetlenia (Lamberta albo Blinna-Phonga) indywidualnie dla każdego obiektu. Zmiana taka może służyć przyspieszeniu rysowania scen, w których tylko niektóre obiekty są błyszczące.
6. Zmodyfikuj aplikację tak, aby dodać oświetlenie hemisferyczne. Wymaga to m.in. dołączenia do szadera fragmentów odpowiednich zmiennych jednolitych (pola opisujące wektor $\boldsymbol{z}$ i wektory I^{sky} i I^{ground} najlepiej dodać do bloku `LSBlock`, przyda się tam też zmienna sterująca „włączaniem” i „wyłączaniem” tego oświetlenia). Kod aplikacji w C należy uzupełnić o uzyskiwanie dostępu do nowych pól w bloku oraz o procedury przypisujące tym polom odpowiednie wartości (pamiętaj, aby wektor $\boldsymbol{z}$ był jednostkowy).

[18] „Zwykłe” procedury mogą być umieszczone w szaderach kompilowanych osobno; szader, który wywołuje procedurę z innego szadera (tego samego etapu), powinien tylko przed miejscem wywołania mieć prototyp tej procedury, tj. nagłówek ze średnikiem. Ale to nie dotyczy szaderów kompilowanych do kodu SPIR-V, w szczególności wszystkich, które mają współpracować z aplikacjami standardu Vulkan. Każdy taki szader musi mieć procedurę `main` i *wszystkie* wywoływane przez nią podprogramy (z wyjątkiem tych z biblioteki GLSL-a).

19

Aplikacja druga D

Dodamy dwa ważne elementy syntezy obrazów: teksturowanie i antyaliasing. **Tekstura** jest określoną na rysowanej powierzchni funkcją, której (skalarna lub wektorowa) wartość w każdym punkcie ma wpływ na kolor tego punktu na obrazie; może to być po prostu kolor punktu na obrazie albo pewien parametr występujący w modelu oświetlenia, na przykład kolor farby, połysk, faktura (tj. chropowatość, rysy lub inne zaburzenia kształtu), przezroczystość lub wreszcie radiancja światła emitowanego przez powierzchnię. Wiele z tych czynników, określanych niezależnie, występuje jednocześnie, skąd wynika potrzeba jednoczesnego używania wielu tekstur dla jednej powierzchni, choć na początek wprowadzimy tylko jedną. Bardzo często teksturę opisuje się za pomocą jedno- dwu- lub trójwymiarowej **tablicy wartości**. Elementy takiej tablicy są przez analogię do pikseli nazywane **tekselami**. Dwuwymiarowa tekstura często jest obrazem, który można po prostu wyświetlić na ekranie (lub wydrukować) i wtedy teksele są tożsame z pikselami. Inna możliwość opisu to **tekstura proceduralna**, której wartość w danym punkcie powierzchni oblicza (na podstawie jakiegoś wzoru) odpowiedni podprogram. Tekstura trójwymiarowa może opisywać materiał, z którego jest wykonany przedmiot, na przykład kolor słojów drewna lub żyłki w marmurze. Punkty powierzchni otrzymują kolory odpowiadające przecięciu tej powierzchni z obszarem trójwymiarowym, w którym tekstura jest określona.

Antyaliasing jest to przeciwdziałanie artefaktom będącym skutkami ograniczonej rozdzielczości rastra monitora. Jeśli dla każdego piksela zostanie obliczony kolor jednego punktu (na hipotetycznym obrazie o nieskończonej rozdzielczości) i kolor ten zostanie przypisany pikselowi, to na ekranie cały obszar piksela będzie miał kolor tego jednego punktu. W efekcie na wspólnym brzegu obszarów o różnych kolorach pojawią się ząbki o wysokości lub szerokości jednego piksela, tym lepiej widoczne, im bardziej kolory tych obszarów kontrastują ze sobą. Zjawisko to można zaobserwować na obrazach wykonanych przez wszystkie wersje obu aplikacji opisanych w poprzednich rozdziałach. To zjawisko, zwane **intermodulacją** (*alias*), jeszcze bardziej może popsuć jakość obrazu obiektów pokrytych teksturą. Dlatego podstawową technikę antyaliasingu (która z grubsza polega na obliczaniu i mieszaniu kolorów wielu punktów w obszarze każdego piksela) wprowadzimy jednocześnie z teksturami.

19.1. Mipmaping

Nałożymy teksturę na część korpusu czajnika składającą się z czterech największych płatów Béziera. Będzie to tekstura dwuwymiarowa, otrzymana z czterech fotografii. W obszarze pokrytym przez teksele (dziedzinie tekstury) są określone **współrzędne tekstury**, które przyjmują w tym obszarze wartości z przedziału $[0, 1]$. Fragmenty obrazu reprezentowanego przez teksturę chcemy odpowiednio zniekształcić (porozciągać lub skurczyć) i nałożyć na poszczególne płaty. Można to zrobić w ten sposób, że dziedzinę płata Béziera (która jest kwadratem jednostkowym) utożsamiamy z pewnym prostokątem w dziedzinie tekstury; ustanawia to odwzorowanie punktów z tego prostokąta na punkty płata w przestrzeni i dalej, po rzutowaniu, na punkty na obrazie.

Ponieważ każdy piksel jest prostokątem, opisane wyżej przekształcenia wiążą nieskończenie wiele punktów w dziedzinie tekstury z tylomaż należącymi do piksela punktami na obrazie; punkty te tworzą pewien podobszar dziedziny tekstury, w którym tekstura na ogół przyjmuje różne wartości. Podobszar ten może być mały albo duży, może też mieć kształt zbliżony do kwadratu lub wydłużony. Rzecz w tym, że w obliczeniu koloru piksela może być konieczne uwzględnienie wielu tekseli — jeśli fragment dziedziny tekstury odwzorowany na piksel jest duży (bo np. cały obiekt na obrazie jest bardzo mały).

Użyjemy stosunkowo prostej i dosyć skutecznej (choć nie zawsze wystarczającej) techniki zaproponowanej w 1983 r. przez Lance'a Williamsa, który nazwał ją **mipmapingiem** (*MIP-mapping*, od łacińskiego *multum in parvo*, wiele w niewielu). Korzysta ona z dodatkowych tablic zawierających reprezentacje danej tekstury o mniejszych rozdzielczościach —

Rysunek 19.1. Tekstura i jej reprezentacje o mniejszej rozdzielczości

dwukrotnie, czterokrotnie, ośmiokrotnie itd. Tablica o największej rozdzielczości ma **poziom** 0, a każda kolejna tablica ma poziom o jeden większy. W najprostszym przypadku można te tablice otrzymać przez zwykłe uśrednianie wartości czwórek tekseli z niższego poziomu, choć lepsze efekty daje *filtrowanie sygnału*, jakim jest tekstura (tj. obliczanie splotu tekstury z odpowiednio dobraną funkcją zwaną **filtrem**, czym tu nie będziemy się zajmować). Dla tekstury dwuwymiarowej dodatkowe tablice zajmują (w pamięci GPU) jedną trzecią miejsca potrzebnego do przechowania tekstury o pełnej rozdzielczości. Aby obliczyć kolor piksela, szader fragmentów sięga do tekstury za pomocą funkcji `texture`. Funkcja ta dokonuje interpolacji wartości odpowiednich tekseli, przy czym zależnie od wielkości odpowiadającego pikselowi obszaru w teksturze wybierany jest odpowiedni poziom, na którym teksele mają wartości uśrednione w odpowiednich obszarach; poziom tym wyższy, im większy jest obszar odpowiadający pikselowi — szczegóły będą opisane dalej.

Na rysunku 19.1 jest pokazana tekstura o pełnej rozdzielczości 1024×1024 teksele oraz dodatkowe reprezentacje tej tekstury o mniejszych rozdzielczościach.

19.2. Szadery

Do szaderów używanych w poprzedniej wersji aplikacji trzeba wprowadzić przetwarzanie współrzędnych tekstury i zapewnić dostęp do samej tekstury. Wektor współrzędnych tekstury jest nowym atrybutem wierzchołka wytwarzanego przez szader rozdrabniania. Podczas rasteryzacji trójkątów będących wynikiem rozdrabniania płata, tak jak inne atrybuty wierzchołków, współrzędne tekstury są interpolowane. Otrzymany w wyniku interpolacji wektor jest przekazywany jako dana wejściowa do szadera fragmentów, który *może* (w zależności od wartości zmiennej jednolitej używanej do sterowania szaderem) użyć tekstury do otrzymania parametrów materiału występujących w modelu oświetlenia. Wektor współrzędnych tekstury dla wierzchołka jest wytwarzany przez szader rozdrabniania na podstawie współrzędnych punktu w dziedzinie płata[1].

Cała scena tworzona przez opisaną tu aplikację składa się z płatów Béziera, rysowanych jednocześnie jako poszczególne instancje jednego płata. Ponieważ szader wierzchołków ma za zadanie tylko przekazać do dalszych obliczeń numer instancji, nie ma w nim żadnych zmian w porównaniu z wersją aplikacji 2C. To samo dotyczy szadera sterowania rozdrabnianiem.

Zmiany dokonane w szaderze rozdrabniania są pokazane na listingu 19.1. Do struktury wyjściowej `GVertex` (czyli `Out`) zostało dodane pole `TexCoord` typu `vec2`. Jest też nowy blok magazynowy o nazwie `BezPatchTexCoord` (i nazwie lokalnej `txc`). Jeśli na płat ma być nałożona tekstura (co ma miejsce, gdy wartość zmiennej jednolitej `ColourSource` jest równa 2), to szader ma nadać współrzędnym pola `Out.TexCoord` wartości liczbowe.

[1] Można go też wygenerować w inny sposób, na przykład na podstawie współrzędnych wierzchołka (przez podstawienie ich do jakiegoś wzoru), lub podać jako atrybut wierzchołka w VAO (wtedy będzie on przekazany na wejście szadera wierzchołków przez etap pobierania wierzchołków), co jednak z naszym sposobem rysowania płatów Béziera jest niewykonalne.

Jeśli obie liczby należą do przedziału $[0,1]$, to reprezentują punkt należący do obszaru tekstury, która zostanie użyta (przez szader fragmentów) do obliczenia koloru piksela. W przeciwnym razie tekstura w obliczeniu koloru zostanie zignorowana.

Listing 19.1. Modyfikacje szadera rozdrabniania

GLSL

```
1: #version 450 core
2:
3: layout(quads,equal_spacing,cw) in;
4: in TCInstance { .... } In[];
5: out GVertex {
6:     vec3 Colour;
7:     vec3 Position;
8:     vec3 Normal;
9:     vec2 TexCoord;
10:  } Out;
11:
12: .... /* niezmienione wszystkie dotychczasowe zmienne jednolite */
13: uniform bool BezNormals;
14: ....
15: layout(std430,binding=3) buffer BezPatchTexCoord {
16:     vec4 txc[];
17:  } txc;
18:
19: uniform TransBlock { .... } trb;
20:
21: int inst;
22:
23: .... /* niezmienione wszystkie procedury oprócz main */
24:
25: void main ( void )
26: {
27:   vec4 pos, nv;
28:
29:   inst = instance[0];
30:   .... /* pominięty fragment bez zmian */
31:   Out.Colour = bezp.Colour;
32:   if ( ColourSource == 2 )
33:     Out.TexCoord = mix ( txc.txc[inst].xy, txc.txc[inst].zw,
34:                          gl_TessCoord.yx );
35: } /*main*/
```

Przyjrzyjmy się obliczaniu współrzędnych tekstury. Szader rozdrabniania ma dostarczyć punkt jednego z n płatów Béziera, wskazanego przez numer instancji podany w zmiennej `instance[0]` i zapamiętany dla wygody w zmiennej `inst` (linia 29). Dla każdego z tych płatów jest podana informacja o prostokącie zawartym w obszarze tekstury (tj. w kwadracie jednostkowym), w odpowiednim elemencie tablicy `txc` w bloku zmiennych jednolitych

BezPatchTexCoord. Element ten jest typu vec4; jego pierwsze dwie i ostatnie dwie współrzędne określają dwa punkty będące przeciwległymi wierzchołkami rozpatrywanego prostokąta; pierwszy z nich utożsamimy z punktem $(0,0)$, a drugi z punktem $(1,1)$ dziedziny *płata Béziera*. Etap rozdrabniania dziedziny dostarczył (w zmiennej gl_TessCoord) współrzędne punktu w dziedzinie płata. Procedury wywoływane przez pominięty fragment procedury main obliczają odpowiadający mu punkt płata i jego wektor normalny w tym punkcie. Aby otrzymać współrzędne tekstury, dokonujemy interpolacji wierzchołków podanych w zmiennej txc.txc[inst] przy użyciu współrzędnych punktu w dziedzinie płata, za pomocą funkcji mix; ponieważ wszystkie trzy parametry tej funkcji są wektorami, interpolacja odbywa się niezależnie dla każdej współrzędnej. Zwróćmy uwagę na przestawienie współrzędnych x, y punktu w dziedzinie płata (w wyrażeniu gl_TessCoord.yx). Jest ono potrzebne, aby obrazki na czajniku były właściwie zorientowane[2]. Inne, bardziej eleganckie rozwiązanie jest opisane w p. 19.8.6.

Modyfikacje szadera geometrii z listingu 15.5 są pokazane na listingu 19.2. Do bloku wejściowego GVertex i wyjściowego FVertex zostało dodane pole TexCoord; oprócz wykonywania dotychczasowych zadań szader kopiuje dane w tym polu z bloku wejściowego do wyjściowego.

Listing 19.2. Modyfikacje szadera geometrii

GLSL

```
#version 450 core
.... /* podstawowe wejście i wyjście bez zmian */

in GVertex {
    vec3 Colour;
    vec3 Position;
    vec3 Normal;
    vec2 TexCoord;
  } In[];

out FVertex {
    vec3 Colour;
    vec3 Position;
    vec3 Normal, TNormal;
    vec2 TexCoord;
  } Out;

void main ( void )
{
```

[2]Orientacja tekstury na płacie Béziera jest ściśle związana z orientacją parametryzacji płata; można ją zmieniać, odwracając kolejność wierszy siatki kontrolnej, odwracając kolejność kolumn, lub zamieniając wiersze z kolumnami — możemy w ten sposób otrzymać 8 różnych orientacji. Dla dowolnego płata w modelu czajnika należałoby w tym celu znaleźć właściwy wiersz w tablicy teapotcn streszczonej na listingu 15.12 i ustawić podane w nim 16 liczb w macierz 4×4, a następnie odwracać kolejność wierszy lub kolumn, albo transponować, a potem przepisać te liczby z powrotem wiersz po wierszu do tablicy.

```
20:   .... /* deklaracje zmiennych i początkowe instrukcje bez zmian */
21:   for ( i = 0; i < 3; i++ ) {
22:     .... /* instrukcje w pętli też bez zmian */
23:     Out.Colour = In[i].Colour;
24:     Out.TexCoord = In[i].TexCoord;
25:     EmitVertex ();
26:   }
27:   EndPrimitive ();
28: } /*main*/
```

Nowe elementy szadera fragmentów są pokazane na listingu 19.3. Do bloku wejściowego `FVertex` jest dodane nowe pole `TexCoord`, zawierające współrzędne tekstury. Ponadto w linii 19 widzimy deklarację zmiennej jednolitej `tex` typu `sampler2D`; dostęp do tekstury dostajemy przez tę zmienną, reprezentującą **ewaluator tekstury** (*sampler*) — obiekt pełniący rolę pomocniczą (ale niezbędną) podczas obliczania wartości tekstury. Parametrami funkcji `texture` są ewaluator i wektor współrzędnych punktu w dziedzinie tekstury, którego liczba współrzędnych musi się zgadzać z wymiarem tej dziedziny (np. typ `vec2` zgadza się z typem `sampler2D`). Wartość funkcji `texture` jest wektorem o czterech współrzędnych. Składową alfa tekstury odrzucamy, wybierając tylko współrzędne R, G, B za pomocą operatora kropki w wyrażeniu `texture ( ... ).rgb`. Nie jest ona potrzebna w obliczeniu koloru, a wynik końcowy obliczenia koloru fragmentu otrzymuje składową alfa w istatniej instrukcji procedury `main`.

Zatem, zmieniona procedura `GetMaterial`, gdy zmienna jednolita `ColourSource` ma wartość 2, tworzy opis materiału wykorzystywany następnie do obliczania koloru oświetlonej powierzchni na podstawie tekstury, chyba że któraś ze współrzędnych tekstury przetwarzanego fragmentu jest poza przedziałem $[0,1]$ — wtedy używany jest opis materiału podany w bloku zmiennych jednolitych `MatBlock`. Ten mechanizm umożliwia nałożenie tekstury tylko na wybrane płaty Béziera. Tekstura jest traktowana jak kolor farby w danym punkcie powierzchni i przypisywana do pól `ambref` i `dirref` zmiennej `mm`. Pola te opisują zdolność odbijania światła rozproszonego w otoczeniu i światła dochodzącego bezpośrednio od źródła światła i ulegającego odbiciu rozproszonemu. Parametry opisujące odbijanie światła w sposób zwierciadlany są brane z opisu materiału w bloku zmiennych jednolitych `mat`.

Listing 19.3. Modyfikacje szadera fragmentów

GLSL

```
1: #version 450 core
2:
3: in FVertex {
4:     vec3 Colour;
5:     vec3 Position;
6:     vec3 Normal, TNormal;
7:     vec2 TexCoord;
8:   } In;
9:
10: uniform TransBlock { .... } trb;
11: struct LSPar { .... };
```

```
12: uniform LSBlock { .... } light;
13: struct Material { .... };
14: uniform MatBlock { .... } mat;
15:
16: uniform int LightingModel = 0, ColourSource = 0;
17: Material mm;
18:
19: uniform sampler2D tex;
20:
21: void GetMaterial ( vec2 tc )
22: {
23:   switch ( ColourSource ) {
24: default: .... break;
25: case 1: ....  break;
26: case 2:
27:     mm = mat.mat;
28:     if ( tc.x >= 0.0 && tc.x <= 1.0 && tc.y >= 0.0 && tc.y <= 1.0 )
29:       mm.ambref = mm.dirref = texture ( tex, tc ).rgb;
30:     break;
31:   }
32: } /*GetMaterial*/
33:
34: .... /* wszystkie pozostałe procedury, w tym main, bez zmian */
```

Drugi program szaderów, używany do wyświetlania siatek kontrolnych płatów Béziera, został zachowany bez żadnych zmian.

19.3. Czytanie i pisanie plików TIFF

Fotografie, które mają stać się teksturami, są zapisane w plikach w formacie TIFF; na początku działania aplikacja czyta te pliki i tworzy na ich podstawie teksturę w pamięci GPU. Odkładając na chwilę na bok główny temat, tj. OpenGL, zobaczmy pomocniczą procedurę, której zadaniem jest przeczytanie takiego pliku i przekazanie aplikacji obrazu w nim zawartego. Procedura ta użyje procedur z biblioteki TIFF (`libtiff.so`), którą trzeba zainstalować z odpowiedniego pakietu, jeśli nie jest to jeszcze zrobione. Oprócz standardowych plików nagłówkowych, takich jak `string.h` (z prototypami procedur `malloc`, `free`, `memset` i `memcpy`) oraz plików nagłówkowych OpenGL-a (z definicją typu `GLubyte`), trzeba włączyć do programu plik nagłówkowy tej biblioteki `tiffio.h`.

Procedura `ReadTiffFile` na listingu 19.4 otrzymuje jako parametr wejściowy nazwę pliku i przekazuje wskaźnik do tablicy, w której są umieszczone wartości pikseli przeczytanego obrazu. Jeśli obrazu nie uda się przeczytać (bo nie ma takiego pliku, jest on nieczytelny lub zabrakło pamięci), to procedura podaje wskaźnik pusty. Jeśli czytanie pliku zakończyło się sukcesem, to zmiennym wskazywanym przez parametry `width` i `height` zostają przypisane wymiary obrazu — szerokość i wysokość w pikselach.

Listing 19.4. Czytanie pliku TIFF

C

```
1: GLubyte *ReadTiffImage ( const char *fn, int *width, int *height )
2: {
3:   TIFF    *tif;
4:   int     w, h, npix;
5:   GLubyte *image;
6:
7:   image = NULL;
8:   if ( (tif = TIFFOpen ( fn, "r" )) ) {
9:     TIFFGetField ( tif, TIFFTAG_IMAGEWIDTH, &w );
10:    TIFFGetField ( tif, TIFFTAG_IMAGELENGTH, &h );
11:    *width = w;
12:    *height = h;
13:    npix = w*h;
14:    image = malloc ( 4*npix );
15:    if ( !image )
16:      goto way_out;
17:    memset ( image, 0, npix*sizeof(uint32_t) );
18:    if ( !TIFFReadRGBAImage ( tif, w, h, (uint32_t*)image, 0 ) )
19:      { free ( image );  image = NULL; }
20: way_out:
21:    TIFFClose ( tif );
22:  }
23:  return image;
24: } /*ReadTiffImage*/
```

Wartość każdego piksela w tablicy jest zapisana w jednej liczbie 32-bitowej (składającej się z czterech bajtów, kolejno *r*, *g*, *b*, *a*); piksele w tablicy są uporządkowane wierszami. Po wykorzystaniu danych przeczytanych z pliku tablicę, której adres początku został przekazany przez procedurę `ReadTiffImage`, należy zwolnić przy użyciu procedury `free`.

Aby w książce był komplecik, na listingu 19.5 zamieściłem procedurę, która odczytuje teksturę (tj. tablicę tekseli) z pamięci GPU i zapisuje ją do pliku TIFF. Nie jest ona używana w opisanej tu aplikacji, ale może się przydać podczas uruchamiania programów. Parametr `fn` jest nazwą pliku TIFF, parametr `tex` jest identyfikatorem tekstury o szerokości `w` i wysokości `h` tekseli. Odczytania tekseli dokonuje procedura `glGetTextureImage`. Jest tu pewna niespójność procedur OpenGL-a. Otóż wiersze pikseli w plikach takich jak TIFF są przechowywane w kolejności „od góry do dołu". Procedury `glTexImage2D` i `glTexSubImage2D` otrzymują tablice o takim właśnie uporządkowaniu wierszy i przesyłając je, przestawiają wiersze, aby otrzymać wewnętrzną reprezentację tekstury zgodną z konwencją przyjętą w standardzie OpenGL (w której wiersz 0 leży na prostej $y = 0$ w układzie współrzędnych tekstury, prosta $y = 1$ zawiera wiersz `h-1`, przy czym oś y jest zorientowana do góry). Procedura `glGetTextureImage` wpisuje wiersze tekstury do tablicy przekazanej jako parametr bez

Listing 19.5. Zapisywanie tekstury w pliku TIFF

C

```
1: void SaveTiffTexture ( const char *fn, GLuint tex, int w, int h )
2: {
3:   TIFF *tif;
4:   unsigned char *image, *a, *b, *c;
5:   int  n, i;
6:
7:   image = malloc ( 3*w*(h+1)*sizeof(char) );
8:   if ( !image )
9:     return;
10:  if ( (tif = TIFFOpen ( fn, "w" )) ) {
11:    n = w*h;
12:    glGetTextureImage ( tex, 0, GL_RGB, GL_UNSIGNED_BYTE,
13:                        3*n*sizeof(char), image );
14:    for ( i = 0, a = image, b = &image[3*w*(h-1)], c = &image[3*n];
15:          i+i < h;
16:          i++, a += 3*w, b -= 3*w ) {
17:      memcpy ( c, a, 3*w*sizeof(char) );
18:      memcpy ( a, b, 3*w*sizeof(char) );
19:      memcpy ( b, c, 3*w*sizeof(char) );
20:    }
21:    TIFFSetField ( tif, TIFFTAG_IMAGEWIDTH, w );
22:    TIFFSetField ( tif, TIFFTAG_IMAGELENGTH, h );
23:    TIFFSetField ( tif, TIFFTAG_BITSPERSAMPLE, 8 );
24:    TIFFSetField ( tif, TIFFTAG_SAMPLESPERPIXEL, 3 );
25:    TIFFSetField ( tif, TIFFTAG_ROWSPERSTRIP, h );
26:    TIFFSetField ( tif, TIFFTAG_COMPRESSION, COMPRESSION_LZW );
27:    TIFFSetField ( tif, TIFFTAG_PHOTOMETRIC, PHOTOMETRIC_RGB );
28:    TIFFSetField ( tif, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG );
29:    TIFFSetField ( tif, TIFFTAG_XRESOLUTION, 300.0 );
30:    TIFFSetField ( tif, TIFFTAG_YRESOLUTION, 300.0 );
31:    TIFFSetField ( tif, TIFFTAG_RESOLUTIONUNIT, RESUNIT_INCH );
32:    TIFFWriteEncodedStrip ( tif, 0, image, 3*n );
33:    TIFFClose ( tif );
34:  }
35:  free ( image );
36: } /*SaveTiffTexture*/
```

przestawiania. Zatem, aby obraz nie był zapisany w pliku „do góry nogami", trzeba dodać instrukcję odwracającą kolejność wierszy; jest nią pętla w liniach 14–20.

Ciąg wywołań procedury `TIFFSetField` w liniach 21–31 zapamiętuje w obiekcie utworzonym przez procedurę `TIFFOpen` niezbędne informacje (pewne parametry, takie jak rodzaj stosowanej kompresji, ustaliłem arbitralnie). Procedura `TIFFWriteEncodedStrip` dokonuje kompresji obrazu i zapisuje go w pliku.

19.4. Procedury przygotowania tekstur

Na listingu 19.6 są przedstawione trzy procedury pomocnicze, których zadaniem jest utworzenie i przygotowanie do pracy reprezentacji tekstury w pamięci GPU. Parametr procedury `CreateMyTexture` jest wysokością i szerokością tekstury w tekselach; jest tu założenie, że oba te wymiary są jednakowe, choć obecnie OpenGL nie narzuca tego ograniczenia. W linii 6 jest wywoływana procedura `glGenTextures`; pierwszy jej parametr podaje liczbę obiektów tekstur do utworzenia, a drugi parametr jest tablicą (w tym przypadku jednoelementową), do której zostaną wpisane identyfikatory obiektów tekstur utworzonych przez `glGenTextures`. W tych obiektach trzeba jeszcze umieścić odpowiednie dane; wszystko po kolei.

W linii 7 nowy obiekt tekstury jest przywiązywany do celu `GL_TEXTURE_2D`, czego późniejszym skutkiem będzie określenie, że jest to tekstura dwuwymiarowa.

W pętli w liniach 8–9 obliczana jest liczba poziomów tekstury dla mipmapingu; w tym przypadku jest to największa liczba l, taka że 2^{l-1} jest dzielnikiem wysokości i szerokości `wh` tekstury. Procedura `glTexStorage2D` wywołana w linii 10 zapisuje w obiekcie informacje o liczbie poziomów i o wymiarach tablicy tekseli na każdym poziomie. Trzeci parametr, o wartości `GL_RGB8`, określa, że każdy teksel ma być zapisany w trzech bajtach, które przechowują jego składowe r, g, b. Dysponując tymi informacjami, procedura `glTexStorage2D` dokonuje rezerwacji odpowiedniego obszaru w pamięci GPU.

Parametr tekstury `GL_TEXTURE_MAX_LEVEL` nadawany w linii 11 przez procedurę `glTexParameteri` jest numerem ostatniego poziomu; można go nie podać, jeśli tablica tekseli na ostatnim poziomie ma wymiary 1×1.[3]

Zadaniem procedury `LoadMyTextureImage` jest przeczytanie pliku w formacie TIFF o nazwie podanej w parametrze `filename` i przesłanie przeczytanych danych do tablicy w pamięci GPU zarezerwowanej przez procedurę `CreateMyTexture`. Założenie jest takie, że wymiary (w pikselach) przeczytanego obrazu nie przekraczają wymiarów tekstury (w tekselach). Procedura sprawdza (w linii 24), czy przeczytany obraz, przesunięty o wektor (x, y), którego współrzędne są podane jako parametry, mieści się w obszarze tekstury i jeśli tak, to wywołuje procedurę `glTexSubImage2D`, która przesyła dane do związanej z obiektem tekstury tablicy w pamięci GPU, na poziomie `0`.[4] Tablica z pikselami zarezerwowana przez procedurę `ReadTiffImage` jest następnie zwalniana.

Procedura `SetupMyTextureMipmaps` powinna zostać wywołana po przeczytaniu i przesłaniu do tablicy na poziomie `0` wszystkich danych (tj. wszystkich obrazów, z których ma być utworzona tekstura). W linii 41 jest wywoływana procedura `glGenerateTextureMipmap`, która na podstawie zawartości tablicy tekseli na poziomie `0` oblicza i wpisuje reprezenta-

[3] Tekstura używana w tej aplikacji ma szerokość i wysokość 1024 tekseli, dzięki czemu na ostatnim poziomie powstaje tablica 1×1. Jeśli szerokość lub wysokość tekstury nie jest parzysta, to wynik dzielenia przez 2 przez procedury OpenGL-a w celu obliczenia wymiarów tekstury na kolejnym poziomie mipmapingu jest zaokrąglany (w dół). Komplikuje to obliczenia wartości tekseli na tym poziomie i ma niedobry wpływ na ich dokładność.

[4] Procedura `glTexStorage2D` zarezerwowała 3 bajty na każdy teksel. Dlatego procedura `glTexSubImage`, która otrzymała parametr — wskaźnik do tablicy, w której teksele są przechowywane w czterech bajtach, dokonuje odpowiedniej konwersji, pomijając bajty ze składową alfa. W aplikacjach, w których ta składowa jest niepotrzebna, warto oszczędzać miejse w pamięci GPU.

Listing 19.6. Procedury przygotowania tekstur

C

```
GLuint CreateMyTexture ( int wh )
{
  GLuint tex;
  int    w, l;

  glGenTextures ( 1, &tex );
  glBindTexture ( GL_TEXTURE_2D, tex );
  for ( w = wh, l = 1;  !(w & 0x01);  l++ )
    w >>= 1;
  glTexStorage2D ( GL_TEXTURE_2D, l, GL_RGB8, wh, wh );
  glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, l-1 );
  ExitIfGLError ( "CreateMyTexture" );
  return tex;
} /*CreateMyTexture*/

char LoadMyTextureImage ( GLuint tex, int txwidth, int txheight,
                          int x, int y, const char *filename )
{
  int     w, h;
  GLubyte *image;

  if ( !(image = ReadTiffImage ( filename, &w, &h )) )
    return false;
  if ( x+w <= txwidth && y+w <= txheight ) {
    glBindTexture ( GL_TEXTURE_2D, tex );
    glTexSubImage2D ( GL_TEXTURE_2D, 0, x, y, w, h,
                      GL_RGBA, GL_UNSIGNED_BYTE, image );
    free ( image );
    ExitIfGLError ( "LoadMyTextureImage" );
    return true;
  }
  else {
    free ( image );
    return false;
  }
} /*LoadMyTextureImage*/

char SetupMyTextureMipmaps ( GLuint tex )
{
  glBindTexture ( GL_TEXTURE_2D, tex );
  glGenerateTextureMipmap ( tex );
  glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR_MIPMAP_LINEAR );
  ExitIfGLError ( "SetupMyTextureMipmaps" );
```

```
46:   return true;
47: } /*SetupMyTextureMipmaps*/
```

cje o mniejszej rozdzielczości do tablic na wszystkich wyższych poziomach. Kolejne dwie instrukcje ustawiają w obiekcie tekstury (a dokładniej, w będącym jego częścią ewaluatorze) parametry określające sposób odtwarzania wartości tekstury na podstawie tablic tekseli. Pierwszy parametr procedury `glTexParameteri` jest identyfikatorem celu, do którego jest przywiązany przetwarzany obiekt tekstury. Drugi parametr procedury jest nazwą parametru w obiekcie, któremu nadajemy wartość, a trzeci określa tę wartość. Parametr `GL_TEXTURE_MAG_FILTER` określa sposób postępowania, gdy obraz teksela na ekranie jest większy niż piksel. Nadanie temu parametrowi wartości `GL_LINEAR` jest równoznaczne z żądaniem, aby wartość tekstury przekazywana szaderowi fragmentów przez funkcję `texture` (zobacz listing 19.3) była otrzymana przez liniową interpolację wartości odpowiednich tekseli w tablicy na poziomie `0`.

Parametr `GL_TEXTURE_MIN_FILTER` określa sposób postępowania, gdy obrazy tekseli są mniejsze od piksela na ekranie. Jeśli parametr ten otrzymał wartość `GL_LINEAR_MIPMAP_LINEAR`, to wyznaczane są dwa sąsiednie poziomy najlepiej pasujące do wielkości obrazu tekseli: teksele z poziomu niższego mają obrazy „trochę za małe", a teksele z poziomu wyższego mają obrazy „trochę za duże" w porównaniu z rozmiarmi piksela. Dla każdego z tych poziomów jest wykonywana interpolacja liniowa tekseli, a potem następuje interpolacja liniowa „między poziomami". Taki sposób obliczania wartości tekstury dla fragmentów jest trochę czasochłonny, ale ma dobre skutki dla końcowej jakości obrazu.

19.5. Zmiany w aplikacji

Zmiany procedury kompilacji szaderów polegają na podaniu nowych nazw plików źródłowych i zmienieniu wartości drugiego parametru procedury `GetAccessToBezPatchStorageBlocks` (zobacz listing 15.7) z `false` na `true`, aby uzyskać dostęp do bloku magazynowego `BezPatchTexCoord`, nieobecnego w programach szaderów używanych przez wcześniejsze wersje aplikacji.

Na końcu procedury `InitMyWorld` trzeba dopisać wywołanie procedury `LoadMyTextures` pokazanej na listingu 19.7. Jej zadaniem jest przeczytanie czterech plików TIFF z obrazami i utworzenie z nich tekstury, która zostanie nałożona na czajnik. Przekazaną wartość, która jest idenyfikatorem tekstury, trzeba przypisać zmiennej `mytexture`, będącej nowym polem struktury typu `AppData`.

Listing 19.7. Tworzenie tekstury przez aplikację

C

```
1: GLuint LoadMyTextures ( void )
2: {
3:   GLuint tex;
4:
5:   if ( (tex = CreateMyTexture ( 1024 )) ) {
```

```
    LoadMyTextureImage ( tex, 1024, 1024,   0,   0, "jaszczur.tif" );
    LoadMyTextureImage ( tex, 1024, 1024, 512,   0, "salamandra.tif" );
    LoadMyTextureImage ( tex, 1024, 1024,   0, 512, "lis.tif" );
    LoadMyTextureImage ( tex, 1024, 1024, 512, 512, "kwiatki.tif" );
    SetupMyTextureMipmaps ( tex );
  }
  return tex;
} /*LoadMyTextures*/
```

Na listingu 19.8 są pokazane zmienione procedury tworzenia reprezentacji i rysowania czajnika. Chcemy, aby tekstura była nałożona tylko na 4 z 32 płatów Béziera, z których składa się czajnik, a ponadto chcemy na każdy z tych czterech płatów nałożyć inny fragment tekstury, tj. zdjęcie przeczytane z innego pliku TIFF. Służy do tego tablica `txc` w bloku magazynowym `BezPatchTexCoord`; jej długość jest liczbą płatów, czyli dla czajnika 32. W strukturze `BezierPatchObjf` (listing 15.6) czwarty element tablicy `buf` jest przeznaczony do przechowywania identyfikatora bufora z blokiem `BezPatchTexCoord` dla zbioru płatów Béziera reprezentowanego przez tę strukturę.

Listing 19.8. Tworzenie reprezentacji i rysowanie czajnika z teksturą

C

```
void ConstructMyTeapot ( AppData *ad )
{
  const GLfloat MyColour[4] = { 1.0, 1.0, 1.0, 1.0 };
  const GLfloat ambr[4]  = { 0.75, 0.6, 0.2, 1.0 };
  const GLfloat diffr[4] = { 0.75, 0.6, 0.2, 1.0 };
  const GLfloat specr[4] = { 0.7, 0.7, 0.6, 1.0 };
  const GLfloat shn = 60.0, wa = 5.0, we = 5.0;
  const GLfloat txc[32][4] =
    {{-1.0,0.0,-1.0,0.0},{-1.0,0.0,-1.0,0.0},{-1.0,0.0,-1.0,0.0},
     {-1.0,0.-1,0.0,0.0},{0.5,0.5,0.0,0.0},{0.5,1.0,0.0,0.5},
     {1.0,0.5,0.5,0.0},{1.0,1.0,0.5,0.5},{-1.0,0.0,-1.0,0.0},
     {-1.0,0.0,-1.0,0.0},{-1.0,0.0,-1.0,0.0},{-1.0,0.0,-1.0,0.0},
     .... /* tu 6 kopii linii podanej wyżej */
     {-1.0,0.0,-1.0,0.0},{-1.0,0.0,-1.0,0.0}};

  ad->myteapot = ConstructTheTeapot ( MyColour );
  SetBezierPatchTessLevel ( ad->myteapot, ad->TessLevel );
  SetBezierPatchNVS ( ad->myteapot, ad->BezNormals );
  ad->matbuf[0] = NewUniformMatBlock ();
  SetupMaterial ( ad->matbuf[0], ambr, diffr, specr, shn, wa, we );
  if ( !GenBezierPatchTextureBlock ( ad->myteapot, &txc[0][0] ) )
    ExitOnError ( "ConstructMyTeapot" );
} /*ConstructMyTeapot*/

void DrawMyTeapot ( AppData *ad )
{
```

```
27:   .... /* tu instrukcje takie, jak w liniach 3-11 na listingu 18.10 */
28:   ChooseMaterial ( ad->matbuf[0] );
29:   glUniform1i ( ad->brprog.ColourSourceLoc, ad->brprog.colour_source );
30:   if ( ad->brprog.colour_source == 2 ) {
31:     BindBezPatchTextureBuffer ( ad->myteapot );
32:     glBindTexture ( GL_TEXTURE_2D, ad->mytexture );
33:     DrawBezierPatches ( ad->myteapot );
34:     glBindTexture ( GL_TEXTURE_2D, 0 );
35:   }
36:   else
37:     DrawBezierPatches ( ad->myteapot );
38: } /*DrawMyTeapot*/
39:
40: void DrawMyTorus ( AppData *ad )
41: {
42:   .... /* tu instrukcje takie, jak w liniach 18-24 na listingu 18.10 */
43:   glUniform1i ( ad->brprog.ColourSourceLoc,
44:                 ad->brprog.colour_source > 0 ? 1 : 0 );
45:   ChooseMaterial ( ad->matbuf[1] );
46:   DrawBezierPatches ( ad->mytorus );
47: } /*DrawMyTorus*/
```

Procedura `ConstructMyTeapot` po utworzeniu reprezentacji czajnika i opisu materiału wywołuje procedurę z listingu 19.9, która tworzy dodatkowy bufor przeznaczony do umieszczenia w nim bloku `BezPatchTexCoord` i przesyła do niego zawartość tablicy `txc`.

W tablicy `txc` pokazanej w liniach 8–14 mamy 32 czwórki liczb typu `GLfloat` albo 32 wektory typu `vec4`; każdy z nich odpowiada jednemu płatowi Béziera. Cztery płaty, na które chcemy nałożyć teksturę, mają numery 4, 5, 6 i 7. Przykładowo, dla płata o numerze 4 mamy podane liczby `0.5,0.5,0.0,0.0`. Wierzchołek $(0,0)$ dziedziny tego płata zostanie zatem odwzorowany na punkt $(\frac{1}{2},\frac{1}{2})$ w dziedzinie tekstury, a wierzchołek $(1,1)$ na punkt $(0,0)$. To oznacza, że na płat o numerze 4 będzie nałożona odpowiednio obrócona lewa dolna ćwiartka tekstury pokazanej na rysunku 19.1, czyli zdjęcie jaszczurki. Wszystkie płaty oprócz wymienionych czterech mają nie być pokryte teksturą. Do osiągnięcia tego efektu służą ujemne liczby podane w pozostałych elementach tablicy `txc` — procedura `GetColours` na listingu 19.3 w razie otrzymania ujemnych współrzędnych tekstury przekazuje opis materiału z bloku `MatBlock` bez uwzględnienia tekstury.

Procedura `DrawMyTeapot` przygotowuje rysowanie czajnika tak jak w aplikacji drugiej C, po czym przypisuje zmiennej jednolitej `ColourSource` wartość sterującą wyborem materiału. Jeśli zmienna `colour_source` ma wartość 2, to w linii 31 za pomocą procedury `BindBezPatchTextureBuffer` przywiązujemy bufor z tablicą `txc` czajnika, a w linii 32 wywołujemy procedurę `glBindTexture`, która przywiązuje naszą teksturę do celu `GL_TEXTURE_2D` — tekstura jest dwuwymiarowa, jej wymiar musi się zgadzać z typem zmiennej jednolitej `tex`, w naszym przypadku `sampler2D`. Po narysowaniu poteksturowanego czajnika, w linii 34, odłączamy teksturę od zmiennej jednolitej `tex`.

Procedura konstrukcji torusa pozostała niezmieniona, natomiast do procedury rysowania torusa, `DrawMyTorus`, została dodana instrukcja w liniach 43–44. Jeśli zmienna `colour_source` ma wartość 2, to zmiennej jednolitej `ColourSource` jest przypisywana wartość 1, aby torus został narysowany bez tekstury, z wykorzystaniem materiału, jaki został dla niego zdefiniowany[5].

Listing 19.9. Procedury `GenBezierPatchTextureBlock` i `BindBezPatchTextureBuffer`

C

```
1: char GenBezierPatchTextureBlock ( BezierPatchObjf *bp, const GLfloat *data )
2: {
3:   GLint size;
4:
5:   size = bp->npatches*4*sizeof(GLfloat);
6:   if ( !(bp->buf[3] = NewStorageBuffer ( size, txcbbp )) )
7:     return false;
8:   glBufferSubData ( GL_SHADER_STORAGE_BUFFER, 0, size, data );
9:   ExitIfGLError ( "GenBezierPatchTextureBlock" );
10:  return true;
11: } /*GenBezierPatchTextureBlock*/
12:
13: void BindBezPatchTextureBuffer ( BezierPatchObjf *bp )
14: {
15:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, txcbbp, bp->buf[3] );
16: } /*BindBezPatchTextureBuffer*/
```

Do procedury `ProcessCharCommand` została dodana reakcja na napisanie na klawiaturze cyfry 2, przy czym reakcje na cyfry 0, 1 i 2 polegają tylko na przypisaniu odpowiedniej wartości zmiennej `appdata.colour_source`. Przypisywanie tej wartości zmiennej jednolitej `ColourSource` ma miejsce bezpośrednio przed rysowaniem czajnika i torusa.

Niezręcznie jest mi o tym wspominać, ale do procedury `DeleteMyWorld` wypada dopisać instrukcję

```
glDeleteTextures ( 1, &appdata.mytexture );
```

19.6. Antyaliasing

Najprostsza technika antyaliasingu polega na **wielokrotnym próbkowaniu** (*multisampling*). Domyślnie dla każdego piksela obrazu oblicza się kolor jednego punktu, odpowiadającego środkowi piksela, i nadaje pikselowi ten kolor. Wielokrotne próbkowanie polega na obliczeniu kolorów pewnej liczby (kilku lub kilkunastu) punktów w obszarze piksela i zmieszaniu

[5] Torus nie jest teksturowany, więc podczas rysowania go zmienna `ColourSource` nie może mieć wartości 2 — dla torusa nie wprowadziliśmy bufora z tablicą `txc`, więc szader nie może wtedy odwoływać się do bloku `BezPatchTexCoord`.

tych kolorów w celu otrzymania wynikowego koloru piksela na obrazie. To oznacza więcej obliczeń do wykonania. W szczególności szader fragmentów może być wywołany więcej razy[6], a więc wykonanie obrazu zabierze odpowiednio więcej czasu.

Aby użyć tej techniki w aplikacji biblioteki GLFW, *tuż przed* utworzeniem okna (za pomocą procedury `glfwCreateWindow`) trzeba wykonać instrukcję

```
glfwWindowHint ( GLFW_SAMPLES, n );
```

przy czym *n* oznacza liczbę punktów (obliczanych próbek koloru) na piksel.

Po utworzeniu okna z wielopróbkowym buforem obrazu antyaliasing jest (domyślnie) włączony, zatem obrazy będą wykonywane z wielokrotnym próbkowaniem. Można to wyłączyć, wykonując instrukcję

```
glDisable ( GL_MULTISAMPLE );
```

co spowoduje (szybsze) rysowanie obrazów gorszej jakości (tj. bez antyaliasingu). Wielokrotne próbkowanie można ponownie włączyć, wywołując procedurę `glEnable` z parametrem jak wyżej.

Uwaga: Metoda antyaliasingu z wykorzystaniem bufora wielopróbkowego nie zawsze działa doskonale, bo w pewnych przypadkach nie zapewnia dokładnego „sklejenia" obrazów trójkątów o wspólnej krawędzi — piksele, przez które przechodzi krawędź mogą się wyróżniać na tle sąsiednich pikseli należących w całości do jednego lub drugiego trójkąta, gdy wynikiem cieniowania trójkątów ma być obraz powierzchni gładkiej.

Rysunek 19.2. Okno aplikacji drugiej D

[6] W rzeczywistości wielokrotne próbkowanie ma na celu oszacowanie, jaka część obszaru piksela należy do obrazu rysowanego wielokąta, jeśli jego brzeg przecina piksel. Domyślnie szader fragmentów jest wywoływany tylko dla jednego punktu w pikselu, ale można włączyć tzw. cieniowanie próbek, aby to zmienić — zobacz p. 21.5.1.

Sposoby tworzenia okna z buforem wielopróbkowym w aplikacjach korzystających z biblioteki FreeGLUT i w natywnych aplikacjach systemu Windows są opisane w p. 19.8.1, a w rozdziale 32 jest przedstawiona szczegółowo natywna aplikacja systemu X Window, która tworzy takie okno, aby wyświetlać obrazy antyaliasowane. Niezależnie od środowiska należy liczyć się z tym, że opisany tu sposób osiągnięcia antyaliasingu nie zadziała. Opisane w tej książce aplikacje, które tworzą okna z opcją wielokrotnego próbkowania, po uruchomieniu na moim laptopie działają normalnie, ale wyświetlają obrazy „ząbkowane", takie jak bez tej opcji. Polecenie utworzenia bufora obrazu z wielokrotnym próbkowaniem dla tworzonego okna aplikacji jest ignorowane, jeśli komputer nie ma wystarczających zasobów (np. pamięci GPU). Pozostaje się z tym pogodzić lub spróbować sposobu opisanego w rozdziale 25.

19.7. Ćwiczenia

1. Zapisz w formacie TIFF obrazki lub fotografie ze *swojej* kolekcji, skalując je do rozsądnej rozdzielczości, na przykład rzędu kilkuset pikseli wzdłuż i wszerz (jest wiele programów, których można do tego użyć, np. GIMP). W razie potrzeby powiększ rozdzielczość tekstury tworzonej przez aplikację i umieść w tej teksturze swoje obrazki. Następnie zmodyfikuj zawartość tablicy `txc` w procedurze `ConstructMyTeapot`, aby nałożyć teksturę na inne płaty Béziera w modelu czajnika.

2. Zmodyfikuj procedurę `ConstructMyTorus`, dodając tablicę `txc` oraz instrukcję, która tworzy UBO z zawartością tej tablicy dla torusa i wykorzystaj ją do nałożenia tekstury na 9 wymiernych płatów Béziera, z których składa się torus.

3. Nałóż teksturę na obiekt bezpośrednio, tj. przypisując zmiennej `out_Colour` wartość funkcji `texture` albo kolor materiału — użyj jakiegoś klawisza do przełączania między sposobami interpretowania tekstury jako koloru farby (podstawianego do modelu oświetlenia) lub końcowego koloru punktu na obrazie.

4. Dla eksperymentu wyłącz antyaliasing i mipmaping. W tym celu wykomentuj wywołanie procedury `glfwWindowHint` przed utworzeniem okna, w procedurze `CreateMyTexture` zmień na 1 drugi parametr wywołania procedury `glTextureStorage2D` (aby był tylko jeden poziom tekstury) i wykomentuj wywołanie procedury `SetupMyTextureMipmaps`. Możesz też użyć jakiegoś klawisza do włączania i wyłączania wielokrotnego próbkowania za pomocą procedur `glEnable` i `glDisable`. Skompiluj aplikację, uruchom i obejrzyj skutki.

5. Wypróbuj różne liczby punktów na piksel w wielokrotnym próbkowaniu, podając procedurze `glfwWindowHint` jako drugi parametr liczby od 1 do 16 i oceń (według własnego gustu) jakość obrazów.

19.8. Uzupełnienia

19.8.1. Antyaliasing w innych środowiskach

Aby włączyć wielokrotne próbkowanie w aplikacji biblioteki FreeGLUT, podczas inicjalizacji (przed utworzeniem okien) trzeba wykonać instrukcje

```
glutSetOption ( GLUT_MULTISAMPLE, n );
glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH |
                      GLUT_MULTISAMPLE );
```

W natywnej aplikacji systemu Windows, tworząc okno, w którym ma być wielokrotne próbkowanie, trzeba w tablicy `pixattr`, przekazywanej jako parametr procedury `wglChoosePixelFormatARB` (zobacz listing 3.16) podać odpowiednią listę atrybutów formatu pikseli, co działa podobnie jak w systemie X Window (porównaj tablice `vattr` na listingach 3.6 i 32.19). Oprócz atrybutów wpisanych do tablicy `pixel_format_attribs` na listingu 3.16 lista ta powinna zawierać następujące dane:

```
WGL_SAMPLE_BUFFERS_ARB, 1,
WGL_SAMPLES_ARB,        8,  /* można podać inną liczbę */
```

przy czym nazwy atrybutów powinny być wprowadzone przez makrodefinicje

```
#define WGL_SAMPLE_BUFFERS_ARB 0x2041
#define WGL_SAMPLES_ARB        0x2042
```

Zamiast samemu pisać te makrodefinicje, można użyć „nieoficjalnego” pliku nagłówkowego `wglext.h`, możliwego do znalezienia w sieci.

Wielokrotne próbkowanie można wyłączyć lub ponownie włączyć, wywołując procedurę `glDisable` lub `glEnable` z parametrem `GL_MULTISAMPLE`.

19.8.2. Rozszerzanie dziedziny tekstur

Do tworzenia tekstur dwuwymiarowych służy ciąg instrukcji podobny do podanego niżej:

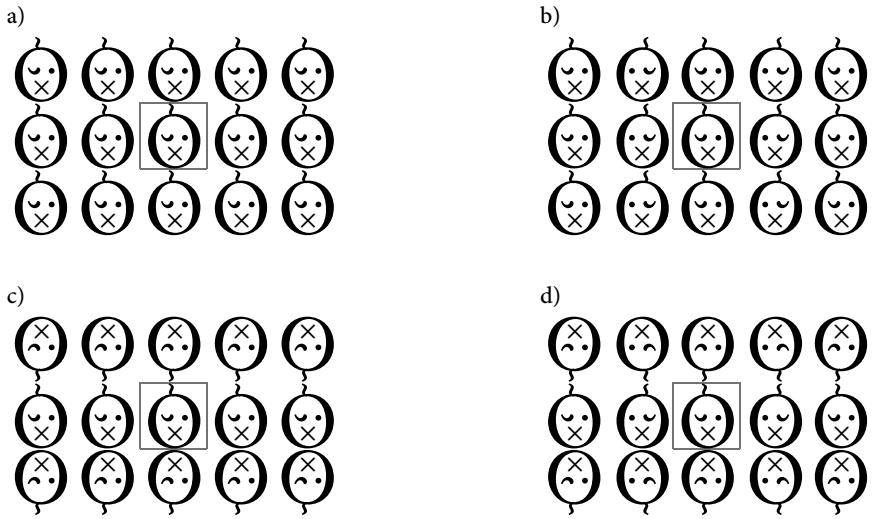
```
glGenTextures ( n, textures );
....
glBindTexture ( GL_TEXTURE_2D, textures[k] );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_LINEAR );
glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGB8, w, h, 0, GL_RGBA,
               GL_UNSIGNED_BYTE, data );
```

Procedura `glGenTextures` rezerwuje n identyfikatorów tekstur i wpisuje je do tablicy podanej jako ostatni parametr; związane z tymi identyfikatorami obiekty tekstur są początkowo puste. Procedura `glBindTexture` przywiązuje obiekt tekstury do celu `GL_TEXTURE_2D` i kolejne instrukcje związane z teksturą działają na przywiązanym do tego celu obiekcie.

Parametry `GL_TEXTURE_WRAP_S` i `GL_TEXTURE_WRAP_T` określają, co się ma dziać, jeśli współrzędne s i t reprezentują punkt poza dziedziną tekstury, tj. poza kwadratem $[0,1]^2$. Jeśli, jak w wyżej podanym przykładzie, mają one wartość `GL_REPEAT`, to odpowiednia współrzędna zostanie zastąpiona przez jej część ułamkową, co skutkuje okresowym powieleniem tekstury na całą płaszczyznę — efekt można zobaczyć na rysunku 19.3a. Inne możliwe wartości to `GL_CLAMP_TO_EDGE` i `GL_CLAMP_TO_BORDER`, `GL_MIRRORED_REPEAT`, `GL_MIRRORED_CLAMP_TO_EDGE` i `GL_MIRRORED_CLAMP_TO_BORDER`. Pierwsza z nich powoduje zastąpienie „wystającej" współrzędnej przez bliższą z liczb 0 lub 1, czyli użycie wartości teksela z pierwszej lub ostatniej kolumny albo wiersza. W drugim przypadku tekstura przyjmuje wartość określoną osobno, przez wykonanie instrukcji

```
glTexParameterfv ( GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, kolor );
```

gdzie `kolor` jest tablicą zawierającą współrzędne r, g, b, a teksela. Wartości „`MIRRORED`" powodują, że następuje odbicie wartości tekstury na odcinku $[0,1]$, dzięki czemu powstaje funkcja parzysta na odcinku $[-1,1]$, która następnie jest rozszerzana okresowo („`REPEAT`"), albo współrzędna poza przedziałem $[-1,1]$ jest zastępowana przez odpowiedni (bliższy) koniec tego przedziału („`CLAMP`").



Rysunek 19.3. Sposoby okresowego rozszerzania tekstury

Procedura `glTexImage2D` przesyła do tekstury obraz o szerokości `w` i wysokości `h`; powoduje to nadanie tablicy tekseli takich wymiarów. Drugi parametr procedury jest poziomem, na którym ma być umieszczony obraz, trzeci określa wewnętrzną reprezentację teksela.

OpenGL oferuje ogromną liczbę możliwych wewnętrznych reprezentacji tekseli, które mogą być opisane za pomocą liczb stało- i zmiennopozycyjnych[7] i mogą mieć od jednej do czterech współrzędnych. Szósty parametr określa obecność brzegu (jednej wyróżnionej „warstwy" tekseli naokoło tablicy tekseli odwzorowanej na kwadrat jednostkowy), przy czym w nowym OpenGL-u parametr ten musi mieć wartość `0`.

Po nadaniu wartości tekselom na poziomie `0` możemy wywołać procedurę `glTexImage2D` kolejno dla wszystkich poziomów, podając `NULL` jako ostatni parametr, a potem wywołać procedurę `glGenerateTextureMipmap`, która utworzy reprezentacje tekstury o mniejszej rozdzielczości. Możemy też samemu utworzyć te reprezentacje w pamięci CPU, przy użyciu dowolnych algorytmów przetwarzania obrazów, które mogą dać lepsze (albo gorsze) efekty niż algorytm zapisany w procedurze `glGenerateTextureMipmap`.

Parametrom określającym sposób filtrowania tekstury, tj. `GL_TEXTURE_MAG_FILTER` i `GL_TEXTURE_MIN_FILTER`, możemy (za pomocą procedury `glTexParameteri`) nadać wartości oznaczone przez nazwy, w których słowo „`LINEAR`" zastąpimy przez „`NEAREST`". Wtedy zamiast dokonywać interpolacji, funkcja `texture` poda wartość teksela najbliższego punktu o podanych współrzędnych tekstury. Zabiera to mniej czasu, ale daje prawie taki sam efekt jak całkowite wyłączenie antyaliasingu tekstury.

Ewaluator tekstury (*sampler*), reprezentowany przez zmienną typu `sampler2D` (lub podobnego), jest obiektem przechowującym parametry opisujące sposób obliczania wartości tekstury przez funkcję `texture`. Każda tekstura ma wbudowany ewaluator, ale można utworzyć dodatkowy obiekt (jeden lub więcej) tego rodzaju, ustawić w nim parametry i spowodować użycie go zamiast wbudowanego w teksturę ewaluatora (z parametrami o innych wartościach). Służą do tego procedury `glGenSamplers`, `glBindSampler` i rodzina procedur `glSamplerParameter*` o nazwach z różnymi przyrostkami. Przykład użycia dodatkowego ewaluatora tekstury jest w p. 28.3.2, a więcej na ten temat można przeczytać w specyfikacji [1] lub w książkach [23] i [24].

19.8.3. Tekstury skompresowane

Bardziej skomplikowane aplikacje wyświetlają bardziej skomplikowane sceny, składające się z wielu teksturowanych obiektów. Tekstury mogą łatwo przepełnić dostępną pamięć GPU, czemu można przeciwdziałać, poddając je kompresji. OpenGL daje taką możliwość; specyfikacja [1] opisuje odpowiednie procedury i wymienia pewne, zawsze dostępne, sposoby kompresji tekstur. Dodatkowe sposoby są zrealizowane w rozszerzeniach standardu, przy czym można z nich korzystać za pomocą tych samych procedur, podając tylko odpowiednie identyfikatory — ale aplikacja musi najpierw sprawdzić, czy dany sposób kompresji jest dostępny tam, gdzie ona ma działać.

[7] Reprezentacja obrazu za pomocą stałopozycyjnych liczb ośmiobitowych może być bezpośrednio wyświetlana na ekranie, ale jeśli obraz jest teksturą stanowiącą dane do dalszych obliczeń numerycznych, to taka reprezentacja może być niewystarczająco dokładna. W takich sytuacjach mogą być używane liczby zmiennopozycyjne typu `float`, tj. 32-bitowe, liczby zmiennopozycyjne „połówkowej precyzji", zajmujące 16 bitów, lub nawet liczby jedenasto- i dziesięciobitowe. Opisy tych reprezentacji liczb można znaleźć w specyfikacji [1].

Metody kompresji tekstur w OpenGL-u są zazwyczaj stratne, a zatem odkodowana podczas nakładania na obiekt tekstura nie jest identyczna z oryginalną, co jednak jest najczęściej niezauważalne. Najwydajniejsze algorytmy stratnej kompresji obrazów mogą zmniejszyć objętość danych kilkudziesięcio- lub nawet kilkusetkrotnie. Ale w przypadku tekstur musi istnieć możliwość szybkiego odkodowania dowolnego fragmentu tablicy tekseli, bez konieczności odtwarzania całych danych. Dlatego wiele z tych algorytmów dzieli tablicę na małe bloki (np. 4×4 teksele) i każdy taki blok koduje osobno. W rezultacie stopień kompresji jest rzędu kilku, co i tak jest opłacalne.

Kompresji tekstury może dokonać aplikacja, która obraz przeczytany z pliku (w postaci nieskompresowanej) przesyła do pamięci GPU, żądając skompresowania go. Inna możliwość to użycie osobnego programu, który zapisze w plikach skompresowane tekstury, które aplikacja będzie mogła bezpośrednio (i szybko) przesłać do pamięci GPU. W tym punkcie opiszę sposób otrzymania pliku ze skompresowaną teksturą i sposób umieszczenia danych z takiego pliku w pamięci GPU.

Procedury przedstawione na listingu 19.10 są przeznaczone do wbudowania do aplikacji, która ma przygotować plik ze skompresowaną teksturą. W tym celu aplikacja powinna wywołać procedurę `CompressMMTexture`, której parametrami są nazwy plików; pierwszy, w formacie TIFF, zawiera obraz, który ma stać się teksturą, a drugi jest nazwą pliku, w którym gotowa tekstura ma być zapisana. Procedura przygotowuje teksturę wielopoziomową dla mipmapingu.

W linii 33 są rezerwowane identyfikatory dwóch tekstur; pierwsza z nich, nieskompresowana, posłuży tylko do przygotowania drugiej. W linii 34 następuje czytanie obrazu przez procedurę z listingu 19.4.

W liniach 37–39, na podstawie wymiarów obrazu, jest ustalana liczba poziomów mipmapingu. Wymiary te muszą być podzielne przez 4, bo tego wymaga algorytm kompresji; ten warunek mają spełniać tablice tekseli na wszystkich poziomach, dlatego wykonywanie pętli kończy się, gdy dowolny z wymiarów przeskalowanego obrazu jest przez 4 niepodzielny.

Pierwsza tekstura jest tworzona w liniach 40–45; kolejno tekstura jest przywiązywana do celu, rezerwowane jest miejsce na tablice na wszystkich poziomach, obraz przeczytany z pliku jest przesyłany do tablicy na poziomie 0 i wreszcie procedura `glGenerateTextureMipmap` wypełnia zmniejszonymi obrazami tablice na wyższych poziomach.

Tekstura skompresowana jest tworzona w liniach 46–62. Kompresja jest wykonywana przez procedurę `glTexImage2D`, jeśli jej trzeci parametr, `internalformat`, ma wartość będącą identyfikatorem formatu skompresowanego. W procedurze na listingu jest to identyfikator `GL_COMPRESSED_RGB8_ETC2`. Obraz o maksymalnej rozdzielczości jest (po poddaniu kompresji) w liniach 49–50 przesyłany z tablicy zawierającej dane przeczytane z pliku. Kolejne obrazy są brane z pierwszej tekstury, która została utworzona po to, aby zaprząc OpenGL-a do ich wygenerowania. Wartość zmiennej `m` jest liczbą tekseli, która w każdym przebiegu pętli zmniejsza się czterokrotnie, bo szerokość i wysokość tablicy maleją dwukrotnie. Procedura `glGetTextureImage` odczytuje tablicę tekseli z kolejnego poziomu pierwszej tekstury, a procedura `glTexImage2D` wywołana w liniach 54–56 dokonuje kompresji danych i przesyła je na odpowiedni poziom drugiej tekstury.

Listing 19.10. Procedury przygotowania pliku ze skompresowaną teksturą

C

```
static char SaveCompressedMMTexture ( const char *fn,
                            GLenum internalformat, GLint levels,
                            GLint *w, GLint *h, GLint *size, GLubyte *data )
{
  FILE *f;
  int  i, s;

  if ( (f = fopen ( fn, "w" )) ) {
    fwrite ( &internalformat, sizeof(GLenum), 1, f );
    fwrite ( &levels, sizeof(GLint), 1, f );
    fwrite ( w, sizeof(GLint), levels, f );
    fwrite ( h, sizeof(GLint), levels, f );
    fwrite ( size, sizeof(GLint), levels, f );
    for ( i = s = 0; i < levels; i++ )
      s += size[i];
    fwrite ( data, s, 1, f );
    fclose ( f );
    return true;
 }
  else
    return false;
} /*SaveCompressedMMTexture*/

GLuint CompressMMTexture ( const char *ifn, const char *ofn )
{
#define MAX_LEVELS 15
  int     w, h, ww, hh, n, m, k, s;
  GLubyte *image;
  GLuint  tex[2];
  GLint   l, size[MAX_LEVELS], ww[MAX_LEVELS], hh[MAX_LEVELS];
  GLenum  internalformat;

  glGenTextures ( 2, tex );
  if ( !(image = ReadTiffImage ( ifn, &w, &h )) )
    return 0;
  n = w*h;
  for ( ww = w, hh = h, l = 1;
        !(ww & 0x03) && !(hh &0x03) && l < MAX_LEVELS;  l++ )
    ww >>= 1, hh >>=1;
  glBindTexture ( GL_TEXTURE_2D, tex[1] );
  glTexStorage2D ( GL_TEXTURE_2D, l, GL_RGB8, w, h );
  glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, l-1 );
  glTexSubImage2D ( GL_TEXTURE_2D, 0, 0, 0, w, h, GL_RGBA,
                    GL_UNSIGNED_BYTE, image );
  glGenerateTextureMipmap ( tex[1] );
```

```
46:   internalformat = GL_COMPRESSED_RGB8_ETC2;
47:   glBindTexture ( GL_TEXTURE_2D, tex[0] );
48:   glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, l-1 );
49:   glTexImage2D ( GL_TEXTURE_2D, 0, internalformat, ww[0] = w, hh[0] = h, 0,
50:                  GL_RGBA, GL_UNSIGNED_BYTE, image );
51:   for ( k = 1, m = n >> 2;  k < l;  k++, m >>= 2 ) {
52:     glGetTextureImage ( tex[1], k, GL_RGB, GL_UNSIGNED_BYTE,
53:                         3*m*sizeof(GLubyte), image );
54:     glTexImage2D ( GL_TEXTURE_2D, k, internalformat,
55:                    ww[k] = ww[k-1] >> 1, hh[k] = hh[k-1] >> 1, 0,
56:                    GL_RGB, GL_UNSIGNED_BYTE, image );
57:   }
58:   free ( image );
59:   glDeleteTextures ( 1, &tex[1] );
60:   glTextureParameteri ( tex[0], GL_TEXTURE_MAG_FILTER, GL_LINEAR );
61:   glTextureParameteri ( tex[0], GL_TEXTURE_MIN_FILTER,
62:                         GL_LINEAR_MIPMAP_LINEAR );
63:   for ( k = s = 0;  k < l;  s += size[k++] )
64:     glGetTextureLevelParameteriv ( tex[0], k,
65:                                    GL_TEXTURE_COMPRESSED_IMAGE_SIZE, &size[k] );
66:   if ( s > 0 ) {
67:     image = (GLubyte*)malloc ( s*sizeof(GLubyte) );
68:     if ( image ) {
69:       for ( k = s = 0;  k < l;  s += size[k++] )
70:         glGetCompressedTexImage ( GL_TEXTURE_2D, k, &image[s] );
71:       SaveCompressedMMTexture ( ofn, internalformat, l, ww, hh,
72:                                 size, image );
73:       free ( image );
74:     }
75:   }
76:   ExitIfGLError ( "CreateMyMMTexture" );
77:   return tex[0];
78: } /*CompressMMTexture*/
```

Instrukcje w liniach 58 i 59 zwalniają pamięć i likwidują pierwszą teksturę, już niepotrzebną. W liniach 60–62 są określane parametry wymagane, jeśli skompresowana tekstura ma być używana przez program, który wywołał procedurę `CompressMMTexture`, a dalsze instrukcje mają za zadanie odczytać skompresowane dane i zapisać je w pliku wynikowym.

Procedura `glGetTextureLevelParameteriv` wywoływana w pętli w liniach 63–65 odczytuje długości w bajtach skompresowanych danych na kolejnych poziomach, zapamiętuje je i oblicza ich sumę. W linii 67 następuje rezerwacja tablicy na dane, po czym w kolejnej pętli dane są przez procedurę `glGetCompressedTexImage` przepisywane z tekstury do tej tablicy. Zadaniem wywołanej w liniach 71–72 procedury `SaveCompressedMMTexture` jest zapisanie pliku z teksturą, po czym tablica jest zwalniana. Identyfikator skompresowanej tekstury jest przekazywany w linii 77; jest ona gotowa do użycia, a jeśli zadaniem programu jest tylko przygotowanie plików z teksturami, to wypada ją niezwłocznie zlikwidować[8].

[8] choćby po to, aby zwolnić pamięć GPU przed przygotowaniem następnego pliku z teksturą

Procedura SaveCompressedMMTexture zapisuje w pliku kolejno identyfikator formatu, tj. sposobu kompresji, liczbę poziomów tekstury, tablice z wymiarami tablic tekseli i długościami zakodowanych danych oraz same dane. W tej samej kolejności dane te są czytane przez pomocniczą procedurę ReadCompressedMMTexture wywołaną przez procedurę LoadCompressedMMTexture, której zadaniem jest utworzenie tekstury z danych w pliku, przygotowanie jej do pracy i przekazanie jej identyfikatora aplikacji (listing 19.11).

Ta procedura jest już bardzo prosta; po zarezerwowaniu identyfikatora tekstury, przywiązaniu do celu GL_TEXTURE_2D i nadaniu wartości potrzebnym parametrom tekstury procedura w pętli przesyła dane na kolejne poziomy, wywołując w tym celu procedurę glCompressedTexImage2D. Inaczej niż dotychczas, rezerwacja pamięci na te dane nie jest wykonywana przez procedurę glTexStorage2D jednocześnie dla wszystkich poziomów, tylko przez procedurę przesyłającą dane, osobno dla każdego poziomu.

Listing 19.11. Procedury przesyłania skompresowanej tekstury z pliku do pamięci GPU

C

```
static GLubyte *ReadCompressedMMTexture ( const char *fn,
                                          GLenum *internalformat, GLint *levels,
                                          GLint **w, GLint **h, GLint **size )
{
  FILE    *f;
  GLubyte *data = NULL;
  GLint   *aux, l;
  int     i, s;

  if ( (f = fopen ( fn, "r" )) ) {
    fread ( internalformat, sizeof(GLenum), 1, f );
    fread ( levels, sizeof(GLint), 1, f );
    if ( !(aux = (GLint*)malloc ( 3*(l = *levels)*sizeof(GLint) )) )
      goto way_out;
    *w = aux;  *h = &aux[l];  *size = &aux[2*l];
    fread ( *w, sizeof(GLint), *levels, f );
    fread ( *h, sizeof(GLint), *levels, f );
    fread ( *size, sizeof(GLint), *levels, f );
    for ( i = s = 0; i < *levels; i++ )
      s += (*size)[i];
    if ( s > 0 ) {
      if ( (data = (GLubyte*)malloc ( s*sizeof(GLubyte) )) )
        fread ( data, s, 1, f );
    }
way_out:
    fclose ( f );
    return data;
  }
  return NULL;
} /*ReadCompressedMMTexture*/

```

```
32: GLuint LoadCompressedMMTexture ( const char *fn )
33: {
34:   GLubyte *data;
35:   GLint   k, l, s, *w, *h, *size;
36:   GLenum  internalformat;
37:   GLuint  tex;
38:
39:   if ( (data = ReadCompressedMMTexture ( fn, &internalformat,
40:                                          &l, &w, &h, &size )) ) {
41:     if ( l > 0 && w[0] > 0 && h[0] > 0 ) {
42:       glGenTextures ( 1, &tex );
43:       glBindTexture ( GL_TEXTURE_2D, tex );
44:       glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, l-1 );
45:       glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
46:       glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
47:                         GL_LINEAR_MIPMAP_LINEAR );
48:       for ( k = s = 0;  k < l;  s += size[k++] )
49:         glCompressedTexImage2D ( GL_TEXTURE_2D, k, internalformat,
50:                                  w[k], h[k], 0, size[k], &data[s] );
51:     }
52:     else
53:       tex = 0;
54:     free ( w );
55:     free ( data );
56:     ExitIfGLError ( "LoadMyCompressedMMTexture" );
57:     return tex;
58:   }
59:   else
60:     return 0;
61: } /*LoadCompressedMMTexture*/
```

Teksele używanej w aplikacji 2D nieskompresowanej tekstury o wymiarach 1024×1024, z trzema bajtami na teksel i z 11 poziomami mipmapingu, zajmują 4914303 bajty. Po dokonaniu kompresji przy użyciu opisanych tu procedur dane opisujące tę teksturę, z ośmioma poziomami mipmapingu, zajmują 699040 bajtów, czyli ponad siedmiokrotnie mniej miejsca. Ceną za to jest niewielkie spowolnienie procesu rysowania.

Tekstury skompresowane można tylko nakładać na obiekty. W kolejnych rozdziałach tekstury będą używane jako bufory obrazów tworzonych poza ekranem. Dla takich zastosowań trzeba rezerwować miejsce na pełne, nieskompresowane tablice tekseli.

19.8.4. Jednoczesne używanie wielu tekstur

W pierwszym akapicie tego rozdziału napisałem o tym, że często występuje potrzeba używania wielu tekstur naraz. Robi się to tak: OpenGL udostępnia pewną liczbę[9] **punktów dowią-**

[9] Co najmniej 48, przy czym liczbę dostępną w konkretnym systemie aplikacja może poznać, wywołując procedurę `glGetIntegerv` z parametrem `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

zania tekstur (*texture units*), przy czym każdy z nich udostępnia dowolny z **celów** (*targets*) nazwanych GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D i kilku innych. Procedury działające na teksturach (np. wprowadzające parametry filtrowania lub przesyłające obraz do tekseli w pamięci GPU) zakładają, że tekstura jest przywiązana do odpowiedniego celu w bieżąco aktywnym punkcie dowiązania tekstury. Na początku działania aplikacji aktywny jest punkt o numerze 0, ale w każdej chwili można go zmienić: w tym celu wystarczy wykonać instrukcję

```
glActiveTexture ( GL_TEXTURE0 + i );
```

i odtąd aktywny jest punkt i (można też podać parametr o jednej z nazw GL_TEXTURE0, ..., GL_TEXTURE31, to są makra rozwijające się do kolejnych liczb całkowitych).

Aby szader fragmentów miał dostęp do tekstury przywiązanej do i-tego punktu, trzeba w nim zadeklarować zmienną jednolitą typu sampler2D w taki sposób:

```
layout(binding=i) uniform sampler2D texi;
```

przy czym wyrażenie i musi być stałe (tj. możliwe do obliczenia podczas kompilacji szadera). Można też zadeklarować tablicę ewaluatorów tekstury, na przykład w taki sposób:

```
layout(binding=i) uniform sampler2D tex[N];
```

Wtedy kolejne elementy tablicy są skojarzone z kolejnymi punktami dowiązania tekstur, od punktu i-tego do punktu $i + N - 1$.

Przed rysowaniem obiektu aplikacja dla każdej nakładanej na niego tekstury powinna wywołać procedurę glActiveTexture z numerem odpowiedniego punktu dowiązania (dodanym do stałej GL_TEXTURE0), a następnie procedurę glBindTexture, podając cel (np. GL_TEXTURE_2D dla tekstur dwuwymiarowych) i identyfikator tekstury, która w czasie rysowania tego obiektu ma być dowiązana do uaktywnionego punktu.

Opis materiału używany przez szadery można rozszerzyć tak, aby kolor każdego materiału mógł być określony przez inną teksturę. Zobaczmy przykład realizacji tej możliwości. Przedstawiona na listingu 19.12 struktura Material ma dodatkowe pole txtnum, którego wartość jest indeksem do tablicy ewaluatorów tekstury tex. Jeśli pole to ma wartość ujemną, to opis materiału nie używa tekstury. Jeśli nieujemną, to kolor materiału (czynnik odpowiedzialny za rozproszone odbicie światła), wzięty z odpowedniej tekstury, jest przypisywany polu diffref zmiennej mm używanej do obliczania oświetlenia.

Modyfikacja struktury Material w treści szadera wymaga oczywiście dostosowania w kodzie napisanym w C instrukcji odczytujących przesunięcia pól w bloku MatBlock.

Strukturę Material w C też trzeba rozszerzyć, o pola txtid i txtbp, których wartościami będą identyfikator tekstury i indeks do tablicy ewaluatorów. Do struktury MatBl (listing 18.5) trzeba dodać nowe pole, ntex, będące licznikiem tekstur tworzonych materiałów; obsługą tego licznika może się zajmować procedura AllocTextureID podobna do AllocMaterialID (listing 18.6). Dodatkowy parametr txtid procedury SetupMaterial

jest identyfikatorem tekstury, którą trzeba utworzyć wcześniej, przy czym jeśli opis materiału nie korzysta z tekstury, to parametr `txtid` powinien mieć wartość `GL_INVALID_INDEX`.

Listing 19.12. Realizacja opisu materiałów z teksturami

GLSL

```
#define MAX_TEXTURES   4
#define MAX_MATERIALS 20

in FVertex { .... vec2 TxtCoord; .... } In;

layout(binding=0) uniform sampler2D tex[MAX_TEXTURES];

struct Material {
    vec4  emission0, emission1, diffref, specref;
    float shininess, wa, we;
    int   txtnum;
  };

uniform MatBlock {
    int mtn;
    Material mat[MAX_MATERIALS];
  } mat;

Material mm;

void GetMaterial ( vec2 tc )
{
  mm = mat.mat[mat.mtn];
  if ( mm.txtnum >= 0 &&
       tc.x >= 0.0 && tc.x <= 1.0 && tc.y >= 0.0 && tc.y <= 1.0 )
    mm.diffref = texture ( tex[mm.txtnum], tc );
} /*GetMaterial*/
```

Procedura `ChooseMaterial` (listing 19.13) oprócz przypisania polu `mtn` numeru materiału dba o przywiązanie tekstury materiału do punktu dowiązania, którego numer został zarezerwowany przez procedurę `AllocTextureID`.

Listing 19.13. Procedury tworzenia i wybierania opisów materiału

C

```
int AllocTextureID ( MatBl *mat )
{
  if ( mat->ntex >= MAX_TEXTURES )
    ExitOnError ( "AllocTextureID" );
  return mat->ntex ++;
} /*AllocTextureID*/

int SetupMaterial ( MatBl *matbl, int m,
```

```
9:                      const GLfloat diffr[4], const GLfloat specr[4],
10:                     GLfloat shn, GLfloat wa, GLfloat we,
11:                     GLuint txtid )
12: {
13:   int tn;
14:
15:   if ( m < 0 )
16:     m = AllocMaterialID ( matbl );
17:   if ( m < MAX_MATERIALS ) {
18:     tn = txtid != GL_INVALID_INDEX ? AllocTextureID ( matbl ) : -1;
19:     .... /* zapamiętywanie opisu materiału w tablicy matbl->mat */
20:          /* i przesyłanie do bufora w pamięci GPU; */
21:          /* wartość zmiennej tn trzeba przypisać polu mat.mat[m].txtnum */
22:     ExitIfGLError ( "SetupMaterial" );
23:   }
24:   return m;
25: } /*SetupMaterial*/
26:
27: void ChooseMaterial ( MatBl *matbl, GLint m )
28: {
29:   Material *mat;
30:
31:   if ( m < 0 || m >= matbl->nmat )
32:     m = 0;
33:   glBindBuffer ( GL_UNIFORM_BUFFER, matbl->matbuf );
34:   glBufferSubData ( GL_UNIFORM_BUFFER, matbofs[0], sizeof(GLint), &m );
35:   mat = &matbl->mat[m];
36:   if ( mat->txtid != GL_INVALID_INDEX ) {
37:     glActiveTexture ( GL_TEXTURE0+mat->txtbp );
38:     glBindTexture ( GL_TEXTURE_2D, mat->txtid );
39:   }
40:   ExitIfGLError ( "ChooseMaterial" );
41: } /*ChooseMaterial*/
```

19.8.5. *Używanie tekstur bez dowiązania

Dostępnych punktów dowiązania tekstur jest kilkadziesiąt, a liczba tekstur używanych przez aplikację może być znacznie większa. Podczas rysowania skomplikowanej sceny należałoby wielokrotnie przywiązywać tekstury dla kolejnych obiektów, a to już może zajmować zbyt wiele czasu. Istnieje możliwość używania tekstur bez dowiązania. Jest ona rozszerzeniem standardu o nazwie `"GL_ARB_bindless_texture"` — aplikacja, która ma korzystać z tego rozszerzenia, musi podczas inicjalizacji sprawdzić jego obecność (w sposób opisany na s. 95). Ponadto aplikacja musi uzyskać dostęp do wymienionych niżej procedur realizujących to rozszerzenie, czego nie zrobi procedura `glewInit` ani `gl3wInit`.

Tekstury, które mają być używane bez dowiązania, są identyfikowane przez **uchwyty** (*handles*), będące obiektami typu zamkniętego[10]. Do ich przechowywania w pamięci CPU aplikacja powinna używać zmiennych typu `GLuint64`.

Po utworzeniu tekstury i określeniu dla niej wszystkich potrzebnych parametrów[11] aplikacja może otrzymać uchwyt tekstury, wykonując instrukcję

```
myhandle = glGetTextureHandleARB ( mytexture );
```

i przekazując jako parametr identyfikator tekstury.

Gdy tekstura nie jest przywiązana, OpenGL może ją usunąć z pamięci GPU, aby zrobić miejsce na inne obiekty. Przywiązanie tekstury sprawia, że staje się ona **rezydentna** (tj. obecna i gotowa) i wtedy szadery mogą się do niej odwoływać. Aby tekstura była rezydentna bez dowiązania, należy wywołać procedurę `glMakeTextureHandleResidentARB`, podając uchwyt jako parametr. Wywołanie procedury `glMakeTextureHandleNonResidentARB` zezwala na usunięcie tekstury niepotrzebnej w danym momencie[12].

Dostęp szadera do tekstury zapewnia zmienna odpowiedniego typu zamkniętego, na przykład `sampler2D`, przy czym nie musi to być zmienna jednolita — jej sposób i miejsce jej zadeklarowania są dość dowolne. Na przykład można umieścić tablicę zmiennych tego typu w bloku zmiennych jednolitych lub w bloku magazynowym i szader, wywołując funkcję `texture`, może podać jako pierwszy parametr odpowiedni element tej tablicy. Ale wcześniej aplikacja *musi* nadać wartości tym zmiennym — uchwyty odpowiednich tekstur, przesyłając do każdej z nich odpowiednie 8 bajtów na przykład przy użyciu procedury `glBufferSubData`. Ponadto szader powinien zawierać dyrektywę

```
#extension GL_ARB_bindless_texture : enable
```

bez której kompilator zasygnalizuje błąd.

Jeśli mamy tablicę uchwytów tekstur, to w zasadzie wszystkie instancje szadera fragmentów, działając równolegle, powinny w wywołaniach funkcji `texture` obliczać ten sam indeks do tablicy — czyli nie można używać innej tekstury dla każdego trójkąta w taśmie. Jest to spowodowane wymaganiem jednolitości obliczeń na GPU (zobacz podrozdz. 9.9). Ale jednoczesne używanie różnych tekstur jest możliwe, jeśli implementacja OpenGL-a, z którą aplikacja współpracuje, zawiera rozszerzenie `"GL_NV_gpu_shader5"` (co aplikacja też musi sprawdzić).

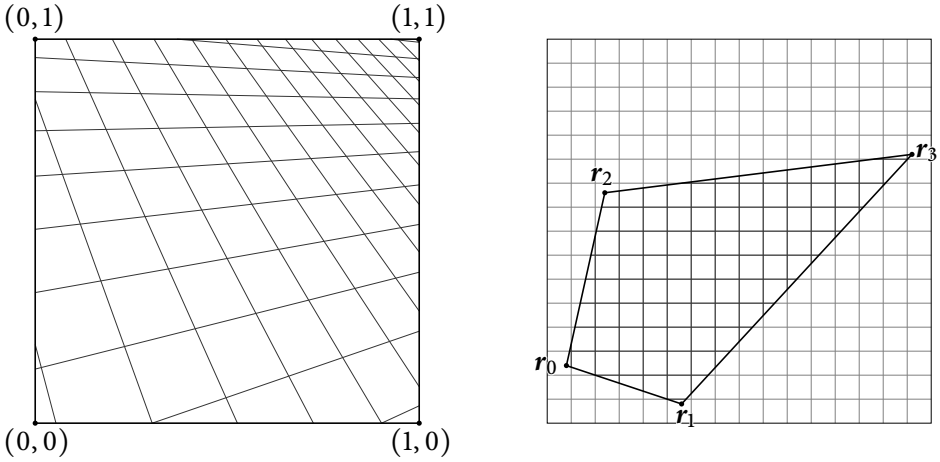
[10] Zobacz rozdział 9; obiekt typu zamkniętego jest liczbą całkowitą, która służy do identyfikowania obiektów takich jak tekstura, obraz lub licznik niepodzielny. Na takich liczbach szader nie może wykonywać żadnych działań arytmetycznych, ale wartości zmiennych typów zamkniętych mogą być przekazywane jako parametry podprogramów w GLSL-u.

[11] które zostają „zamrożone" w ewaluatorze, który będzie używany za pośrednictwem uchwytu, wobec czego nie można ich zmienić

[12] Trzeba bardzo skrupulatnie pilnować, aby potrzebne tekstury były podczas rysowania rezydentne, skutki zaniedbań mogą być nieprzyjemne.

19.8.6. Rzutowe odwzorowanie dziedziny tekstury

W geometrii rzutowej znany jest fakt, że mając dwie czwórki punktów na płaszczyźnie, takie że żadne trzy punkty z czwórki nie są współliniowe (a poza tym punkty te mogą być dowolne), możemy skonstruować (jednoznacznie określone) przekształcenie rzutowe, które pierwszą z tych czwórek przeprowadza na drugą. Dzięki temu możemy utożsamić dziedzinę płata Béziera (czyli kwadrat o wierzchołkach $(0,0)$, $(1,0)$, $(1,1)$ i $(0,1)$) z dowolnym czworokątem $\boldsymbol{r}_0\boldsymbol{r}_1\boldsymbol{r}_3\boldsymbol{r}_2$ w dziedzinie tekstury dwuwymiarowej (rys. 19.4).



Rysunek 19.4. Dziedzina płata i jej obraz w przekształceniu rzutowym w dziedzinę tekstury

Dane punkty $\boldsymbol{r}_0$, $\boldsymbol{r}_1$, $\boldsymbol{r}_2$ i $\boldsymbol{r}_3$ będziemy reprezentować za pomocą wektorów współrzędnych jednorodnych, $\boldsymbol{R}_i = (X_i, Y_i, W_i) = (w_i x_i, w_i y_i, w_i)$, przy czym przyjmiemy $w_0 = 1$ i założymy, że czworokąt $\boldsymbol{R}_0\boldsymbol{R}_1\boldsymbol{R}_3\boldsymbol{R}_2$ jest równoległobokiem. Aby skonstruować odpowiednie przekształcenie, wystarczy obliczyć współrzędne wagowe reprezentacji punktów $\boldsymbol{r}_1$ i $\boldsymbol{r}_2$.

Równanie $\boldsymbol{R}_0 + \boldsymbol{R}_3 = \boldsymbol{R}_1 + \boldsymbol{R}_2$ wyraża fakt, że przekątne równoległoboku mają wspólny środek. Równanie to możemy przepisać w postaci

$$\begin{bmatrix} x_1 & x_2 & -x_3 \\ y_1 & y_2 & -y_3 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix}.$$

Jest to układ równań liniowych, którego macierz jest nieosobliwa, jeśli punkty $\boldsymbol{r}_1$, $\boldsymbol{r}_2$, $\boldsymbol{r}_3$ nie leżą na jednej prostej. Jeśli ponadto czworokąt $\boldsymbol{r}_0\boldsymbol{r}_1\boldsymbol{r}_3\boldsymbol{r}_2$ jest wypukły, to współrzędne wagowe w_1, w_2, w_3 są dodatnie i czworokąt ten jest obrazem kwadratu jednostkowego w konstruowanym tu przekształceniu rzutowym. Jeśli czworokąt $\boldsymbol{r}_0\boldsymbol{r}_1\boldsymbol{r}_3\boldsymbol{r}_2$ jest równoległobokiem, to $w_1 = w_2 = w_3 = 1$ i otrzymamy przekształcenie afiniczne.

Pełna reprezentacja przekształcenia dziedziny płata Béziera w dziedzinę tekstury składa się z punktu $\boldsymbol{r}_0$ i wektorów $\boldsymbol{V}_1 = \boldsymbol{R}_1 - \boldsymbol{R}_0$, $\boldsymbol{V}_2 = \boldsymbol{R}_2 - \boldsymbol{R}_0$, czyli razem z ośmiu liczb, które da się upakować w dwóch zmiennych typu `vec4`. Tak więc osiem liczb (współrzędnych punktów $\boldsymbol{r}_0$, $\boldsymbol{r}_1$, $\boldsymbol{r}_2$, $\boldsymbol{r}_3$ na płaszczyźnie), które *określają* potrzebne nam przekształcenie rzutowe,

zamieniliśmy na inne osiem liczb, które umożliwiają szybkie obliczanie wartości tego przekształcenia[13]. Mając punkt $(u, v) \in [0,1]^2$ (należący do dziedziny płata, na który chcemy nałożyć teksturę), szader fragmentów powinien obliczyć wektor

$$\boldsymbol{P} = \boldsymbol{R}_0 + u\boldsymbol{V}_1 + v\boldsymbol{V}_2$$

i podać go jako parametr funkcji `textureProj` (s. 207) obliczającej wartość tekstury[14]. Do obliczenia wektora $\boldsymbol{P}$ można też użyć wzoru $\boldsymbol{P} = A[u, v, 1]^T$ z macierzą $A = [\boldsymbol{V}_1, \boldsymbol{V}_2, \boldsymbol{R}_0]$.

Przedstawiona tu konstrukcja daje znacznie więcej możliwości niż opisany wcześniej w tym rozdziale (i użyty w aplikacji 2D) sposób odwzorowania dziedziny płata w dziedzinę tekstury, za (niewielką) cenę dwukrotnie większej objętości danych reprezentujących odwzorowanie tekstury. Dlatego zaimplementowanie tej konstrukcji polecam jako ćwiczenie. Pierwszym krokiem jest napisanie i uruchomienie procedur `M3x3LUDecompf` i `M3x3LUSolvef` na podstawie procedur z listingu 5.5, a dalej powinno już być łatwo.

19.8.7. *Wierzchołki położone w punktach niewłaściwych

Pozostańmy jeszcze na pograniczu geometrii afinicznej z geometrią rzutową. Dwie różne proste położone w jednej płaszczyźnie zawsze mają coś wspólnego: punkt (jeśli się przecinają) albo kierunek (jeśli są równoległe). Jak wiemy, obrazy w rzucie perspektywicznym prostych równoległych mogą się przecinać, a obrazem prostych przecinających się mogą być proste równoległe. W geometrii afinicznej kierunki prostych bywają nazywane **punktami w nieskończoności** albo **punktami niewłaściwymi**, a geometria rzutowa w żaden sposób nie odróżnia ich od punktów „zwykłych"[15].

Rysowanie obiektów nieograniczonych, na przykład brył rozciągających się „do nieskończoności", lub widocznej po horyzont płaszczyzny[16], na której możemy ustawiać inne przedmioty, wymaga trochę więcej pracy niż rysowanie „zwykłych" prymitywów. Pierwszy (drobny) kłopot sprawia reprezentowanie punktów niewłaściwych przy użyciu wektorów współrzędnych jednorodnych, których współrzędna wagowa jest równa 0.

Patrząc „afinicznie" na prostą o kierunku wektora $\boldsymbol{v} \neq \boldsymbol{0}$ przechodzącą przez punkt $\boldsymbol{p}$, wyobrażamy sobie dwa punkty w nieskończoności na jej przeciwległych „końcach", z których jeden otrzymamy, przechodząc z wyrażeniem $\boldsymbol{p} + t\boldsymbol{v}$ do granicy dla $t \to +\infty$, a drugi dostaniemy dla $t \to -\infty$, ale w geometrii rzutowej kierunek prostej to jest *jeden* punkt. Punkt $\boldsymbol{p} + t\boldsymbol{v}$ jest reprezentowany (między innymi) przez wektor współrzędnych jednorodnych $(wx_{\boldsymbol{p}} + x_{\boldsymbol{v}}, wy_{\boldsymbol{p}} + y_{\boldsymbol{v}}, wz_{\boldsymbol{p}} + z_{\boldsymbol{v}}, w)$, gdzie $w = 1/t$. Jeśli $t \to +\infty$, to w dąży do zera, przyjmując wartości dodatnie, a jeśli $t \to -\infty$, to w przyjmuje wartości ujemne. W obu

[13] Wektory $\boldsymbol{V}_1$ i $\boldsymbol{V}_2$ wystarczy obliczyć tylko raz, a potem GPU będzie przekształcać odpowiednie punkty podczas przetwarzania setek tysięcy lub milionów fragmentów.

[14] Przekształcenia rzutowe są reprezentowane przez różnowartościowe liniowe przekształcenia przestrzeni współrzędnych jednorodnych. Podany tu rachunek, po drobnych uzupełnieniach i dolaniu niewielkiej ilości formaliny, jest dowodem twierdzenia geometrii rzutowej wspomnianego na początku tego punktu.

[15] I dlatego pojęcia równoległości prostych w geometrii rzutowej nie ma.

[16] Takie obiekty we Wszechświecie nie zostały odkryte. Ale grafika ma swoje prawa.

przypadkach granica jednorodnej reprezentacji punktów prostej jest ta sama, $(x_{\boldsymbol{v}}, y_{\boldsymbol{v}}, z_{\boldsymbol{v}}, 0)$. Zatem odcinek, którego jeden koniec jest „zwykły", a drugi jest punktem niewłaściwym reprezentowanym przez taki wektor, jest półprostą, ale nie wiadomo którą. Aby przeciąć ten dylemat, wybierzemy półprostą składającą się z punktów $\boldsymbol{p} + t\boldsymbol{v}$ dla $t \geqslant 0$.[17]

Zajmiemy się rysowaniem trójkątów, których jeden lub dwa wierzchołki są punktami niewłaściwymi; co najmniej jeden wierzchołek trójkąta powinien być „zwykły"[18]. Chcemy narysować także widoczne w oknie części trójkąta wystające poza przednią i tylną ścianę bryły widzenia. W tym celu trzeba te części wyznaczyć, co oznacza konieczność samodzielnego zaimplementowania algorytmu obcinania, tj. wyznaczania części wspólnej trójkąta i półprzestrzeni. Wcześniej trójkąty trzeba obciąć bocznymi ścianami ostrosłupa widzenia, co oznacza zdublowanie obcinania będącego stałym etapem potoku przetwarzania grafiki[19]. Algorytm obcinania wbudujemy w szader geometrii (listing 19.14). Przyjmiemy, że położenia wierzchołków trójkąta danego na wejściu są podane w układzie współrzędnych świata (o to ma zadbać szader wierzchołków) i reprezentowane przez wektory współrzędnych jednorodnych o *nieujemnych* wagach, a poza tym wierzchołki mają jeden dodatkowy atrybut, kolor.

Z jednego trójkąta szader może wytworzyć co najwyżej trzy taśmy trójkątowe, które (co można udowodnić) w sumie nie mogą mieć więcej niż 15 wierzchołków. Z każdym wierzchołkiem będzie podany wektor współrzędnych *jednorodnych* jego położenia w układzie świata oraz wektor normalny trójkąta i (interpolowany podczas obcinania) kolor.

Pierwszą czynnością procedury `main` jest przepisanie atrybutów wierzchołków (położeń i kolorów) do globalnych tablic roboczych (linie 50–53), przy czym w linii 51 są obliczane położenia wierzchołków w układzie obserwatora. Opisana dalej procedura obcinania `SHClip` pobiera dane z tych tablic i zapisuje w nich wyniki. Instrukcja w linii 54 oblicza jednostkowy wektor normalny płaszczyzny trójkąta.

Część wspólna wielokąta wypukłego i półprzestrzeni jest figurą wypukłą, która ma co najwyżej o jeden wierzchołek więcej (może też mieć ich tyle samo lub mniej). Po czterech wywołaniach procedury obcinającej (w liniach 55, 57, 59 i 61) z trójkąta otrzymamy więc co najwyżej siedmiokąt, przy czym jeśli wynik któregoś obcinania ma mniej wierzchołków niż trójkąt, to szader natychmiast kończy działanie. Dwa ostatnie parametry procedury `SHClip` określają początki miejsc w tablicach, z których odczytywane są dane i zapisywane wyniki. Wyniki czwartego obcinania znajdują się zatem w pierwszych połowach tablic.

[17] Zauważmy, że w geometrii rzutowej proste są liniami zamkniętymi, w związku z czym dowolne dwa punkty są końcami *dwóch* odcinków prostej, na której leżą. Aby określić odcinek jednoznacznie, trzeba oprócz końców podać jeszcze jego jeden (dowolny) punkt. Podobnie, trzy punkty niewspółliniowe są wierzchołkami *czterech* różnych trójkątów. Aby wybrać jeden z nich, trzeba podać jeszcze jeden punkt z jego wnętrza. Dalej będziemy wybierać trójkąty, których punkty są reprezentowane przez kombinacje wypukłe wektorów współrzędnych jednorodnych podanych jako reprezentacje wierzchołków.

[18] Zbiorem wszystkich punktów niewłaściwych płaszczyzny jest **prosta niewłaściwa**, w geometrii rzutowej taka sama jak każda inna. Trzy punkty niewłaściwe w płaszczyźnie reprezentowane przez trzy liniowo zależne wektory współrzędnych jednorodnych o zerowej wadze są więc współliniowe, a zatem nie wyznaczają trójkąta. Trzy punkty niewłaściwe reprezentowane przez wektory liniowo niezależne o zerowej współrzędnej wagowej są wierzchołkami czterech trójkątów położonych w **płaszczyźnie niewłaściwej**, składającej się z wszystkich punktów niewłaściwych przestrzeni trójwymiarowej.

[19] Proszę mi wierzyć — nie da się tego uniknąć i otrzymać dobre obrazy.

Listing 19.14. Szader geometrii do rysowania trójkątów o wierzchołkach w punktach niewłaściwych

GLSL

```
#version 430

layout(triangles) in;
layout(triangle_strip,max_vertices=15) out;

in vec3 Colour[];

out FVertex {
    vec4 WPosition;
    vec3 Normal, Colour;
  } Out;

uniform TransBlock {
    mat4  mm, mmti, vm, pm, vpm;
    vec4  eyepos;
    float left, right, bottom, top, near, far;
    vec4  viewport;
  } trb;

vec3 cross4 ( vec4 v0, vec4 v1, vec4 v2 ) { .... }  /* listing 15.1 */

vec4 pos[16], wpos[16];
vec3 colour[16];
vec3 nv;

int SHClip ( vec4 cpn, int n, int k, int l ) { .... } /* listing 19.15 */

void OutputTrStrip ( int l, int s )
{
  int i, k;

#define EMIT(I) {\
  gl_Position = pos[I]; \
  Out.WPosition = wpos[I]; \
  Out.Normal = nv; \
  Out.Colour =  colour[I]; \
  EmitVertex (); }

  for ( i = 0, k = l-1;  i <= k;  i++, k- ) {
    EMIT ( i+s )
    if ( k > i ) EMIT ( k+s );
  }
  EndPrimitive ();
} /*OutputTrStrip*/

```

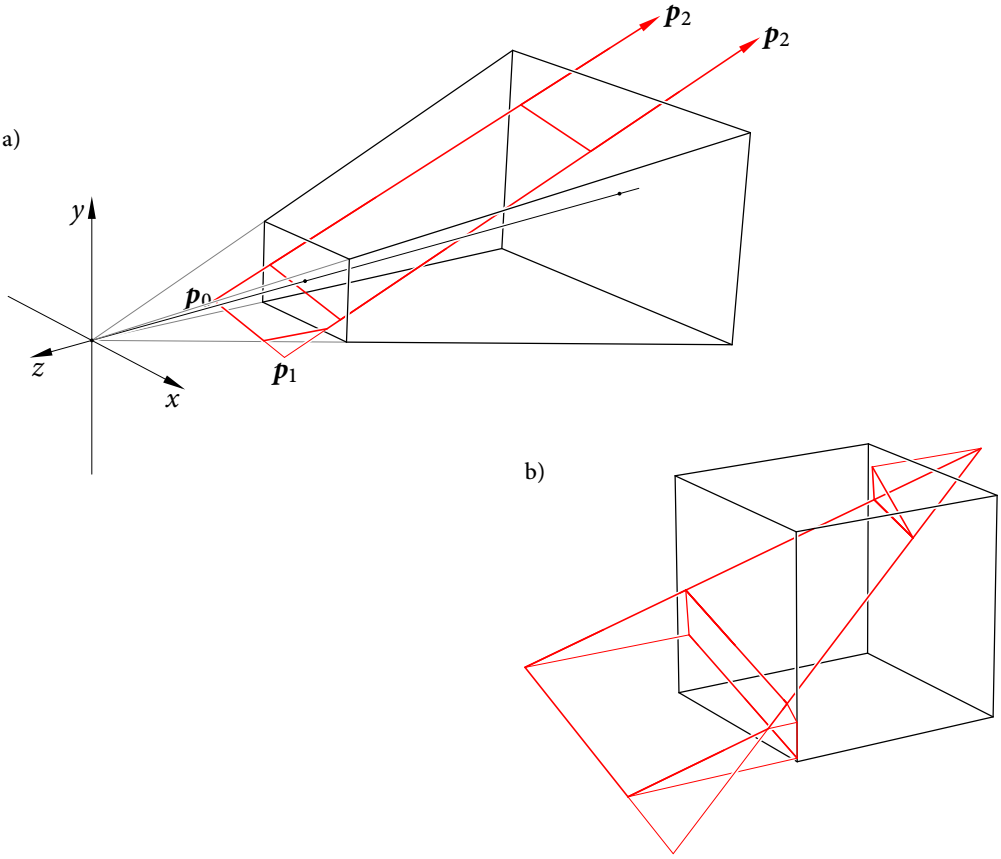
```
46: void main ( void )
47: {
48:   int i, j, l;
49:
50:   for ( i = 0; i < 3; i++ ){
51:     pos[i] = trb.vm * (wpos[i] = gl_in[i].gl_Position);
52:     colour[i] = Colour[i];
53:   }
54:   nv = normalize ( cross4 ( wpos[0], wpos[1], wpos[2] ) );
55:   if ( (j = SHClip ( vec4(trb.near,0.0,trb.left,0.0), 3, 0, 8 )) < 3 )
56:     return;
57:   if ( (j = SHClip ( vec4(-trb.near,0.0,-trb.right,0.0), j, 8, 0 )) < 3 )
58:     return;
59:   if ( (j = SHClip ( vec4(0.0,trb.near,trb.bottom,0.0), j, 0, 8 )) < 3 )
60:     return;
61:   if ( (j = SHClip ( vec4(0.0,-trb.near,-trb.top,0.0), j, 8, 0 )) < 3 )
62:     return;
63:   for ( i = 0; i < j; i++ )
64:     pos[i] = trb.pm * pos[i];
65:   OutputTrStrip ( j, 0 );
66:   if ( (l = SHClip ( vec4(0.0,0.0,1.0,-0.999), j, 0, 8 )) > 2 ) {
67:     for ( i = 0; i < l; i++ )
68:       pos[i+8].z = pos[i+8].w;
69:     OutputTrStrip ( l, 8 );
70:   }
71:   if ( (l = SHClip ( vec4(0.0,0.0,-1.0,-0.999), j, 0, 8 )) > 2 ) {
72:     for ( i = 0; i < l; i++ )
73:       pos[i+8].z = -pos[i+8].w;
74:     OutputTrStrip ( l, 8 );
75:   }
76: } /*main*/
```

Aby umożliwić obcinanie, w bloku `TransBlock` oprócz macierzy przekształceń wierzchołków muszą być podane parametry l, r, b, t, n, f bryły widzenia (zobacz podrozdz. 6.2). Niech $\boldsymbol{P}$ oznacza wektor współrzędnych jednorodnych punktu $\boldsymbol{p}$, który jeśli jest „zwykły", to ma współrzędną wagową dodatnią. Półprzestrzeń (do której mają być obcinane wielokąty) reprezentujemy za pomocą pewnego niezerowego wektora $\boldsymbol{N} \in \mathbb{R}^4$; punkt $\boldsymbol{p}$ należy do tej półprzestrzeni, jeśli iloczyn skalarny $\langle \boldsymbol{P}, \boldsymbol{N} \rangle$ jest nieujemny, przy czym jeśli jest zerem, to punkt $\boldsymbol{p}$ leży na jej płaszczyźnie brzegowej[20]. Dla półprzestrzeni, której brzegiem jest płaszczyzna przechodząca przez punkt $\boldsymbol{q}$ i mająca wektor normalny $\boldsymbol{n}$ (którego zwrot wskazuje tę półprzestrzeń), wektor $\boldsymbol{N}$ można skonstruować, dołączając do wektora $\boldsymbol{n}$ współrzędną wagową $W = \langle \boldsymbol{o} - \boldsymbol{q}, \boldsymbol{n} \rangle$. Ponieważ płaszczyzny boczne ostrosłupa widzenia przechodzą przez początek $\boldsymbol{o}$ układu obserwatora (będący środkiem rzutowania perspektywicznego),

[20] Zauważmy tu konsekwencję jednorodności reprezentacji punktów i półprzestrzeni: znak liczby $\langle \boldsymbol{P}, \boldsymbol{N} \rangle$ zależy tylko od kierunków i zwrotów wektorów $\boldsymbol{N}$ i $\boldsymbol{P}$.

w kolejnych wywołaniach procedury obcinania, jako reprezentacje półprzestrzeni (pierwszy parametr), podane są wektory $(n, 0, l, 0)$, $(-n, 0, -r, 0)$, $(0, n, b, 0)$ i $(0, -n, -t, 0)$.

W pętli w liniach 63–64 następuje obliczenie współrzędnych wierzchołków w układzie kostki standardowej, a dalej (w linii 65) wynik wcześniejszego obcinania jest przekazywany na wyjście szadera. Jest to wielokąt wypukły o co najwyżej siedmiu wierzchołkach; kolejność, w jakiej te wierzchołki są wyprowadzane przez procedurę `OutputTrStrip`, tworzy z niego taśmę trójkątową, z zachowaniem orientacji trójkąta danego na wejściu szadera. Trójkąty z tej taśmy zostaną w etapie obcinania potoku przetwarzania grafiki dodatkowo obcięte przednią i tylną płaszczyzną kostki standardowej.



Rysunek 19.5. Trójkąt z wierzchołkiem niewłaściwym w układzie obserwatora (a) i ten sam trójkąt przekształcony do układu kostki standardowej (b)

Trzeba jeszcze znaleźć, odpowiednio przekształcić i wyprowadzić części trójkąta wystające poza płaszczyzny przedniej i tylnej ściany ostrosłupa widzenia (rys. 19.5). Procedura obcinania wywołana w liniach 66 i 71 pobiera wierzchołki od początku tablic roboczych i zapisuje wynik, zaczynając od miejsca 8; wynik ten może być co najwyżej ośmiokątem. Teraz praca jest wykonywana w układzie kostki standardowej, zatem pierwszym parametrem pro-

cedury są wektory $(0, 0, 1, -0.999)$ i $(0, 0, -1, -0.999)$. Przesunięcie płaszczyzn obcinających w stronę środka kostki (zamiana liczby -1 na -0.999) ma na celu skompensowanie błędów zaokrągleń, których skutkiem (bez tego przesunięcia) byłoby „nieszczelne" sklejenie otrzymanego wielokąta (wystającego poza kostkę standardową) z wielokątem wewnątrz kostki — na wspólnym brzegu wielokątów widoczne byłyby pojedyncze niezamalowane piksele.

Instrukcje w liniach 67–68 i 71–73 rzutują wierzchołki wystających poza kostkę standardową części trójkąta odpowiednio na płaszczyzny jej tylnej ($z = 1$) i przedniej ($z = -1$) ściany. Dzięki temu wyprowadzone przez szader trójkąty przejdą etap obcinania i będą mogły być narysowane[21]. Wektor `Normal` wyprowadzany z tymi wierzchołkami jest oczywiście obliczonym w linii 54 wektorem normalnym płaszczyzny trójkąta danego na wejściu.

Pozostało przedstawienie algorytmu obcinania i realizującej go procedury (listing 19.15). Jest to klasyczny **algorytm Sutherlanda-Hodgmana**. Algorytm obcinania wielokąta o wierzchołkach $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_{n-1}$ przetwarza kolejno krawędzie $\boldsymbol{p}_{n-1}\boldsymbol{p}_0$, $\boldsymbol{p}_0\boldsymbol{p}_1$, …, $\boldsymbol{p}_{n-2}\boldsymbol{p}_{n-1}$. W zmiennych `ds` i `dt` jest przechowywana informacja, po której stronie płaszczyzny obcinającej znajduje się każdy z końców $\boldsymbol{s}$, $\boldsymbol{t}$ bieżącej krawędzi. Informacja ta, dla punktu $\boldsymbol{p}$ reprezentowanego przez wektor współrzędnych jednorodnych $\boldsymbol{P}$, jest iloczynem skalarnym $d_{\boldsymbol{P}} = \langle \boldsymbol{P}, \boldsymbol{N} \rangle$.

Parametry `k` i `l` określają miejsca w tablicach, skąd należy odczytywać wierzchołki obcinanego wielokąta i gdzie należy zapisywać wynik. Makrodefinicja `OUTPUT` wpisuje do tablic położenie i pozostałe atrybuty wyprowadzanego wierzchołka. Jeśli oba końce krawędzi $\boldsymbol{st}$ są po właściwej stronie (czyli obie liczby, $d_{\boldsymbol{S}}$ i $d_{\boldsymbol{T}}$, są nieujemne), to w linii 16 jest wyprowadzany wierzchołek $\boldsymbol{t}$ (tzn. wektor współrzędnych jednorodnych $\boldsymbol{T}$ jego położenia i pozostałe atrybuty). Jeśli punkt $\boldsymbol{s}$ jest po właściwej stronie, a punkt $\boldsymbol{t}$ po niewłaściwej, to zamiast niego jest w linii 19 wyprowadzany wierzchołek $\boldsymbol{r}$ położony na przecięciu krawędzi z płaszczyzną obcinającą. Wreszcie, jeśli punkt $\boldsymbol{s}$ jest po stronie niewłaściwej, a $\boldsymbol{t}$ po właściwej, to w liniach 25–26 są wyprowadzane dwa wierzchołki: wierzchołek $\boldsymbol{r}$ na przecięciu krawędzi z płaszczyzną i koniec $\boldsymbol{t}$ krawędzi.

Wektor współrzędnych jednorodnych punktu $\boldsymbol{r}$ jest obliczany na podstawie wzoru $\boldsymbol{R} = (1-t)\boldsymbol{S} + t\boldsymbol{T}$, do którego podstawiana jest liczba $t = d_{\boldsymbol{S}}/(d_{\boldsymbol{S}} - d_{\boldsymbol{T}})$; możemy sprawdzić, że

$$\begin{aligned}\langle \boldsymbol{R}, \boldsymbol{N} \rangle &= \Big(1 - \frac{d_{\boldsymbol{S}}}{d_{\boldsymbol{S}} - d_{\boldsymbol{T}}}\Big)\langle \boldsymbol{S}, \boldsymbol{N} \rangle + \frac{d_{\boldsymbol{S}}}{d_{\boldsymbol{S}} - d_{\boldsymbol{T}}}\langle \boldsymbol{T}, \boldsymbol{N} \rangle \\ &= \frac{-\langle \boldsymbol{T}, \boldsymbol{N} \rangle \langle \boldsymbol{S}, \boldsymbol{N} \rangle}{\langle \boldsymbol{S}, \boldsymbol{N} \rangle - \langle \boldsymbol{T}, \boldsymbol{N} \rangle} + \frac{\langle \boldsymbol{S}, \boldsymbol{N} \rangle \langle \boldsymbol{T}, \boldsymbol{N} \rangle}{\langle \boldsymbol{S}, \boldsymbol{N} \rangle - \langle \boldsymbol{T}, \boldsymbol{N} \rangle} = 0,\end{aligned}$$

a zatem istotnie, punkt $\boldsymbol{r}$ leży w płaszczyźnie obcinającej. Interpolacji położeń wierzchołków i pozostałych atrybutów (współrzędnych położenia w układzie świata oraz koloru) dokonuje funkcja `mix`. Instrukcja w linii 31 przekazuje liczbę wierzchołków wielokąta pozostałego po obcinaniu.

[21] To przekształcenie powoduje późniejszą interpolację wszystkich atrybutów wierzchołków tych części tak, jak gdyby miały kwalifikator `noperspective`, zobacz p. 12.4.4. Nie mam na to rady.

Listing 19.15. Procedura obcinania wielokąta

GLSL

```
int SHClip ( vec4 cpn, int n, int k, int l )
{
  vec4  s, t, a, b;
  vec3  cs, ct;
  float ds, dt;
  int   i, m;
#define OUTPUT(P,W,C) \
    { pos[l+m] = P;  wpos[l+m] = W;  colour[l+m] = C;  m++; }

  s = pos[k+n-1];  a = wpos[k+n-1];  cs = colour[k+n-1];
  ds = dot ( s, cpn );
  for ( i = m = 0;  i < n;  i++ ) {
    t = pos[k+i];  b = wpos[k+i];  ct = colour[k+i];
    dt = dot ( t, cpn );
    if ( ds >= 0.0 ) {
      if ( dt >= 0.0 ) OUTPUT ( t, b, ct )
      else {
        ds /= ds - dt;
        OUTPUT ( mix ( s, t, ds ), mix ( a, b, ds ), mix ( cs, ct, ds ) )
      }
    }
    else {
      if ( dt >= 0.0 ) {
        ds /= ds - dt;
        OUTPUT ( mix ( s, t, ds ), mix ( a, b, ds ), mix ( cs, ct, ds ) )
        OUTPUT ( t, b, ct )
      }
    }
    s = t;  a = b;  cs = ct;  ds = dt;
  }
  return m;
#undef OUTPUT
} /*SHClip*/
```

Zajmijmy się jeszcze teksturowaniem wielokątów, a ściślej znalezieniem, dla przetwarzanego przez szader fragmentów punktu obrazu wielokąta, współrzędnych kartezjańskich u, v określonych w płaszczyźnie tego wielokąta. Mając je, możemy na przykład pokryć wielokąt „kafelkami", czyli powielić okresowo teksturę określoną w kwadracie jednostkowym.

W punkcie 19.8.6 jest przedstawiona konstrukcja przekształcenia rzutowego, które kwadrat jednostkowy przeprowadza na dowolnie wybrany czworokąt. Problem postawiony wyżej można łatwo rozwiązać za pomocą tej konstrukcji. Niech $\boldsymbol{q}_0$, $\boldsymbol{q}_1$, $\boldsymbol{q}_2$ będą niewspółliniowymi punktami płaszczyzny rysowanego wielokąta i niech $\boldsymbol{q}_3 = \boldsymbol{q}_1 + \boldsymbol{q}_2 - \boldsymbol{q}_0$. Czworokąt $\boldsymbol{q}_0\boldsymbol{q}_1\boldsymbol{q}_3\boldsymbol{q}_2$ jest równoległobokiem, który w układzie współrzędnych o początku w punkcie $\boldsymbol{q}_0$ i wersorach osi $\boldsymbol{q}_1 - \boldsymbol{q}_0$ i $\boldsymbol{q}_2 - \boldsymbol{q}_0$ jest kwadratem jednostkowym. Należy znaleźć obrazy $\boldsymbol{r}_i$ wszystkich

punktów $\boldsymbol{q}_i$ (tzn. obliczyć współrzędne kartezjańskie punktów $\boldsymbol{r}_i$ w układzie okna), a następnie, zgodnie z opisem w p. 19.8.6, obliczyć macierz A reprezentującą przekształcenie przeprowadzające kwadrat jednostkowy na czworokąt $\boldsymbol{r}_0\boldsymbol{r}_1\boldsymbol{r}_3\boldsymbol{r}_2$. Przekształcenie potrzebne teraz (nazwiemy je **przekształceniem tekstury**) jest opisane przez macierz A^{-1}, którą trzeba udostępnić szaderowi fragmentów, na przykład w zmiennej jednolitej typu mat3.

Współrzędne ξ_i, η_i w oknie obrazu $\boldsymbol{r}_i$ punktu $\boldsymbol{q}_i$ reprezentowanego przez wektor współrzędnych jednorodnych $\boldsymbol{Q}_i$ otrzymamy, obliczając wektor $\boldsymbol{Q}'_i = (X'_i, Y'_i, Z'_i, W'_i) = PVM\boldsymbol{Q}_i$ (zobacz podrozdz. 6.6) i podstawiając jego współrzędne do wzorów (6.1); wymiary i położenie klatki w oknie, potrzebne w tym obliczeniu, są podane w dodanym do bloku `TransBlock` polu `viewport` (listing 19.14, linia 17).

Rysunek 19.6 przedstawia obrazy przykładowych figur nieograniczonych. Szader fragmentów użyty do ich narysowania powstał z szadera na listingu 10.4 przez wprowadzenie następujących zmian: po pierwsze, blok wejściowy zawiera teraz położenie odpowiadającego fragmentowi punktu w układzie świata reprezentowane przez wektor współrzędnych jednorodnych, w zmiennej `In.WPosition`. Szader zaczyna działanie od sprawdzenia, czy współrzędna wagowa tego wektora jest zerem. Jeśli tak, to kończy działanie (wykonując instrukcję discard), a jeśli nie, to szader oblicza współrzędne kartezjańskie, aby ich użyć do obliczeń oświetlenia. Po drugie, do szadera została dopisana zmienna jednolita `Ainv`, zawierająca macierz przekształcenia tekstury, i podprogram pokazany na listingu 19.16.

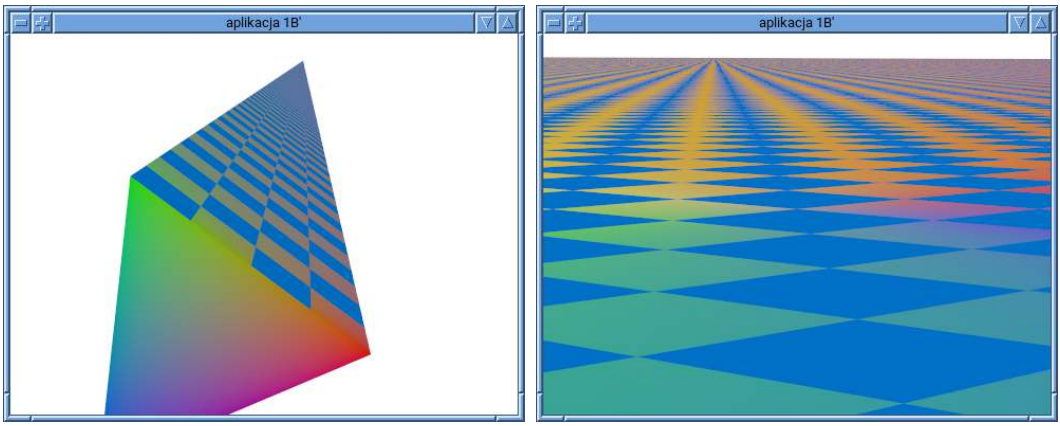
Listing 19.16. Procedura tekstury szachownicy

GLSL

```
1: uniform mat3 Ainv;
2:
3: vec3 GetMyTexture ( void )
4: {
5:   vec3 pt;
6:
7:   pt = Ainv * vec3 ( gl_FragCoord.xy, 1.0 );
8:   if ( (int ( floor ( pt.x/pt.z ) + floor ( pt.y/pt.z ) ) & 0x01) != 0 )
9:     return In.Colour;
10:  else
11:    return vec3 ( 0.0, 0.25, 0.8 );
12: } /*GetMyTexture*/
```

Po obliczeniu współrzędnych u, v punktu na płaszczyźnie podprogram bada, czy suma ich części całkowitych jest nieparzysta. Jeśli jest nieparzysta, to przekazuje kolor punktu otrzymany na wejściu (otrzymany przez interpolację kolorów wierzchołków), a jeśli parzysta, to przekazuje inny kolor; podany kolor jest używany jako kolor materiału w obliczeniach oświetlenia.

Przedstawiony na rysunku 19.6 po lewej stronie pryzmat ma cztery ściany i cztery wierzchołki, z których jeden jest położony w punkcie niewłaściwym; oto ich wektory współrzęd-



Rysunek 19.6. Obrazy nieograniczonego pryzmatu i płaszczyzny

nych jednorodnych:

$$\boldsymbol{P}_0 = (0.5, 0.5\sqrt{3}, 0, 1),\ \boldsymbol{P}_1 = (-1, 0, 0, 1),\ \boldsymbol{P}_2 = (0.5, -0.5\sqrt{3}, 0, 1),\ \boldsymbol{P}_3 = (0, 0, -1, 0).$$

Pryzmat ten jest w istocie czworościanem i w zasadzie można by go narysować, wyświetlając jedną taśmę trójkątową (zobacz rys. 7.4). Ale wtedy szader geometrii miałby dodatkowe zadanie: na podstawie numerów wierzchołków w taśmie (odczytanych ze zmiennej `gl_VertexID` i podanych mu przez szader wierzchołków) musiałby zidentyfikować przetwarzaną ścianę pryzmatu i przekazać jej numer szaderowi fragmentów, aby ten mógł wybrać macierz właściwego przekształcenia tekstury z tablicy zawierającej te macierze dla wszystkich ścian. Dlatego łatwiej jest rysować ściany osobno, umieszczając bezpośrednio przed rysowaniem każdej z nich macierz przekształcenia tekstury w zmiennej jednolitej niebędącej tablicą.

Punkty wyznaczające równoległobok (prostokąt) w płaszczyźnie ściany $\boldsymbol{p}_0\boldsymbol{p}_1\boldsymbol{p}_3$ są takie:

$$\boldsymbol{q}_0 = (0.5, 0.5\sqrt{3}, 0),\ \boldsymbol{q}_1 = (0.125, 0.375\sqrt{3}, 0),\ \boldsymbol{q}_2 = (0.5, 0.5\sqrt{3}, 1).$$

Pokazana na obrazku z prawej strony płaszczyzna $z = 0$ składa się z czterech trójkątów, $\boldsymbol{p}_0\boldsymbol{p}_1\boldsymbol{p}_2$, $\boldsymbol{p}_0\boldsymbol{p}_2\boldsymbol{p}_3$, $\boldsymbol{p}_0\boldsymbol{p}_3\boldsymbol{p}_4$ i $\boldsymbol{p}_0\boldsymbol{p}_4\boldsymbol{p}_1$, których wierzchołki są reprezentowane przez wektory

$$\boldsymbol{P}_0 = (0, 0, 0, 1),\ \boldsymbol{P}_1 = (1, 0, 0, 0),\ \boldsymbol{P}_2 = (0, 1, 0, 0),\ \boldsymbol{P}_3 = (-1, 0, 0, 0),\ \boldsymbol{P}_4 = (0, -1, 0, 0).$$

Użyty do określenia przekształcenia tekstury czworokąt (będący kwadratem) ma wierzchołki w punktach

$$\boldsymbol{q}_0 = (0, 0, 0),\ \boldsymbol{q}_1 = (1, 0, 0),\ \boldsymbol{q}_2 = (0, 1, 0).$$

Przekształcenie to jest wspólne dla wszystkich trójkątów, dzięki czemu zbudowaną z nich płaszczyznę można narysować za pomocą jednego wywołania procedury `glDrawElements` w trybie `GL_TRIANGLE_FAN`.

Na koniec podam jeszcze trzy uwagi. Aby część trójkąta wystająca za tylną ścianę bryły widzenia została narysowana, przedtem trzeba wywołać procedurę `glDepthFunc` z parametrem `GL_LEQUAL`, bo głębokość wszystkich fragmentów tej części po rasteryzacji jest równa 1, a więc *nie jest* mniejsza niż początkowa zawartość bufora głębokości. Domyślny sposób wykonywania testu widoczności można przywrócić, wywołując tę procedurę z parametrem `GL_LESS`.

Dodatkowy kłopot podczas rysowania części trójkąta wystających poza przednią i tylną ścianę bryły widzenia sprawia fakt, że algorytm widoczności z buforem głębokości nie działa dla tych części, przez co końcowy obraz zależy tylko od kolejności ich wyświetlania (i dlatego często nie jest poprawny). Jeśli rysujemy ściany nieograniczonej wypukłej bryły (obiektu z zamkniętą objętością), to problem rozwiązuje użycie mechanizmu odrzucania ścian (zobacz p. 7.6.2). Przed rysowaniem trzeba określić orientację ścian odwróconych przodem, wybrać, które ściany (odwrócone przodem, czy tyłem) mają być odrzucane i włączyć odrzucanie. Wybór orientacji odrzucanych ścian zależy od tego, czy obserwator znajduje się wewnątrz, czy na zewnątrz bryły, co też trzeba zbadać przed rysowaniem.

Nakładając tekstury proceduralne, takie jak opisana wyżej, koniecznie trzeba dokonać antyaliasingu w sposób opisany w p. 21.5.1. Bez tego obrazy mogą wyglądać brzydko.

Ćwiczenie: Narysuj czworościan, którego dwa lub trzy wierzchołki są położone w punktach niewłaściwych.

20

Aplikacja druga E

Dodamy do sceny lustro, w którym czajnik z torusem będzie się odbijał. Możemy taki efekt osiągnąć na dwa sposoby. Pierwszy sposób polega na odbiciu sceny (tj. obu obiektów i wszystkich źródeł światła) w płaszczyźnie lustra i narysowaniu odbitej sceny w obszarze zajmowanym przez obraz lustra. Drugi sposób, zrealizowany w opisanej tu aplikacji, polega na odbiciu w płaszczyźnie lustra położenia obserwatora i narysowaniu obrazu sceny „widzianej zza lustra" (przy czym lustro i wszelkie przedmioty położone za nim pominiemy). Płaszczyzna lustra jest wtedy rzutnią, a samo lustro (lub prostokąt otaczający lustro, które może nie być prostokątne) jest klatką, w której określimy raster o odpowiedniej rozdzielczości. Otrzymany obraz rastrowy nałożymy jako teksturę na lustro podczas rysowania sceny widzianej z oryginalnego położenia obserwatora[1].

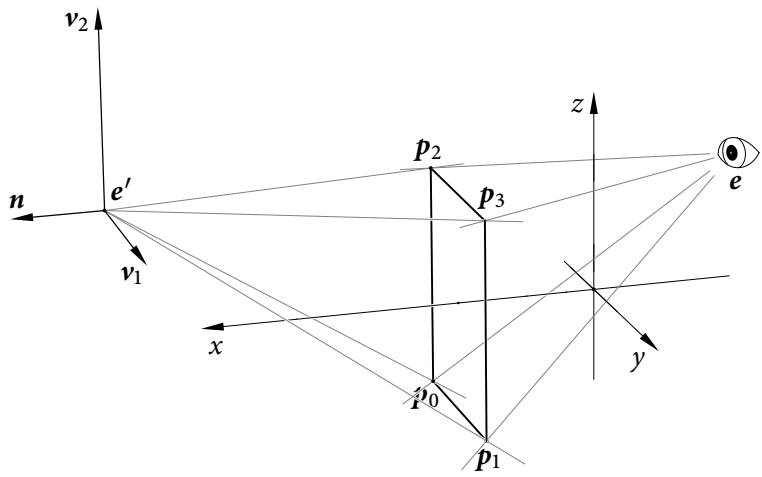
Realizacja opisanego sposobu wymaga użycia odrobiny algebry z geometrią oraz nowego elementu OpenGL-a: tworzenia obrazu poza oknem na ekranie. Nie życzymy sobie oglądać obrazu zza lustra inaczej niż jako odbicie w lustrze odpowiednich obiektów w gotowym obrazie całej sceny. Ale po kolei.

20.1. Algebra z geometrią

Dowolną płaszczyznę, w tym płaszczyznę lustra, możemy reprezentować za pomocą jej dowolnego punktu, $\boldsymbol{p}_0$, i wektora normalnego, $\boldsymbol{m}$ (ewentualnie jednostkowego, $\boldsymbol{n} = \pm\boldsymbol{m}/\|\boldsymbol{m}\|$). Obraz $\boldsymbol{e}'$ położenia obserwatora $\boldsymbol{e}$ w odbiciu symetrycznym względem tej płaszczyzny otrzymamy ze wzorów (5.13) i (5.15), z których po przekształceniach dostaniemy

$$\boldsymbol{e}' = \boldsymbol{e} - \frac{2\langle \boldsymbol{m}, \boldsymbol{e} - \boldsymbol{p}_0\rangle}{\langle \boldsymbol{m}, \boldsymbol{m}\rangle}\boldsymbol{m} = \boldsymbol{e} - 2\langle \boldsymbol{n}, \boldsymbol{e} - \boldsymbol{p}_0\rangle \boldsymbol{n}.$$

[1] Zauważmy, że stosując każdy z tych sposobów, musimy scenę narysować dwukrotnie. Czyżby ktoś był zaskoczony?



Rysunek 20.1. Położenie obserwatora, lustro i układ odbitego obserwatora

Dla położonego w przestrzeni trójwymiarowej prostokąta[2] (lustra) o wierzchołkach $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, $\boldsymbol{p}_3$, $\boldsymbol{p}_2$ (podanych w układzie współrzędnych świata) obliczymy wektory

$$\boldsymbol{v}_1 = \boldsymbol{p}_1 - \boldsymbol{p}_0, \quad \boldsymbol{v}_2 = \boldsymbol{p}_2 - \boldsymbol{p}_0, \quad \boldsymbol{m} = \boldsymbol{v}_1 \wedge \boldsymbol{v}_2, \quad \boldsymbol{n} = \operatorname{sgn}\langle \boldsymbol{m}, \boldsymbol{p}_0 - \boldsymbol{e} \rangle \frac{\boldsymbol{m}}{\|\boldsymbol{m}\|}.$$

Punkt $\boldsymbol{e}'$ i wektory $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{n}$ stanowią układ odniesienia dla układu współrzędnych kartezjańskich w przestrzeni. Będzie to **układ obserwatora odbitego w lustrze**; jego położenie, będące początkiem układu, to punkt $\boldsymbol{e}'$. Wektory $\boldsymbol{v}_1$ i $\boldsymbol{v}_2$ mają kierunki i długości boków lustra, a wektor jednostkowy $\boldsymbol{n}$ jest do nich (czyli do płaszczyzny lustra) prostopadły; w nowym układzie te trzy wektory są wersorami osi (rys. 20.1). Wybór zwrotu wektora $\boldsymbol{n}$ wynika z potrzeby otrzymania układu współrzędnych, w którym punkty płaszczyzny lustra mają współrzędną z ujemną. Macierz przejścia od nowego układu do układu świata jest taka[3]:

$$V^{-1} = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \boldsymbol{n} & \boldsymbol{e}' \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{20.1}$$

Odwrotność tej macierzy, V, opisuje przejście od układu świata do układu odbitego obserwatora; wykorzystamy ją podczas rysowania obrazu na teksturze, która później będzie nałożona na lustro. Potrzebujemy jeszcze macierzy P opisującej przekształcenie bryły widoczności na kostkę standardową.

W układzie obserwatora odbitego w lustrze punkty $\boldsymbol{p}_0$ i $\boldsymbol{p}_3$ mają odpowiednio współrzędne $(l, b, -n)$ i $(r, t, -n)$ (zobacz opis rzutowania perspektywicznego w rozdz. 6). Liczby l, r, b, t, n podamy jako parametry procedury `M4x4Frustumf`, która utworzy macierz P.

[2] To może być równoległobok, gdyby ktoś tego potrzebował.

[3] To jest macierz 4×4, mnożymy przez nią wektory współrzędnych jednorodnych.

Ponieważ wektor $\boldsymbol{n}$ jest jednostkowy, liczba n (`near`) jest odległością obserwatora (i jego obrazu odbitego w lustrze) od płaszczyzny lustra. Potrzebujemy jeszcze parametru f (`far`) określającego położenie tylnej ściany bryły widzenia. Sensownym wyborem wydaje się przyjęcie tego parametru o takiej samej wartości jak podczas tworzenia macierzy dla końcowego obrazu. Jeszcze jedno: rysowane obiekty zostaną obcięte do kostki standardowej. To w szczególności oznacza, że gdyby obiekt wystawał poza płaszczyznę lustra, to jego wystająca część nie zostałaby narysowana. Właśnie tak powinno być.

Konsekwencją dokonanego wyboru wektorów $\boldsymbol{v}_1$ i $\boldsymbol{v}_2$ jest odwzorowanie prostokąta — lustra — na całą ścianę $[-1,1] \times [-1,1] \times \{-1\}$ kostki standardowej, dzięki czemu wykonany obraz będzie dokładnie odpowiadał obszarowi lustra.

20.2. Tworzenie obrazów poza oknem

Termin *framebuffer* jest na język polski powszechnie tłumaczony prawie dosłownie (i prawie sensownie) jako **bufor ramki**[4]; niech już będzie. Termin ten oznacza strukturę danych, w ramach której mogą być tworzone obrazy. Struktura taka jest tworzona przez system okien (z uwzględnieniem *jego* specyfiki) dla każdego okna, z którym OpenGL ma współdziałać. Aplikacja może utworzyć dodatkowe bufory ramki i wykorzystać je do tworzenia obrazów pomocniczych (np. obrazów sceny odbitej w lustrze) lub do tworzenia obrazów, które nie mają być natychmiast wyświetlane, tylko na przykład zapisywane w plikach jako klatki filmu animowanego (i mają np. znacznie większą rozdzielczość niż ekran monitora).

Sam bufor ramki po utworzeniu *nie jest* gotowy do pracy; trzeba w nim zainstalować **załączniki** (*attachments*), czyli konkretne obiekty, w których powstaje obraz[5]. Załącznik może być teksturą lub **buforem roboczym** (*renderbuffer*). Tekstury są bardziej uniwersalne (bo można je potem nałożyć na obiekty na innym obrazie, dokonując m.in. filtrowania i interpolacji przez ewaluatory), za to bufory robocze mogą być lepiej zoptymalizowane do współpracy z buforem ramki w charakterze jego załączników (zobacz p. 20.6.2). Aby utworzyć obraz, zawsze potrzebny jest załącznik będący buforem obrazu. Zazwyczaj potrzebny jest też załącznik używany jako bufor głębokości i rzadziej załącznik — bufor maski (*stencil bufer*), który tu jest nam niepotrzebny.

20.3. Szadery

Do rysowania czajnika i torusa poza ekranem i na ekranie wykorzystamy szadery wypróbowane w aplikacji drugiej D. Pierwszy program służy do rysowania płatów Béziera z uwzględnieniem oświetlenia, a drugi do rysowania siatek kontrolnych. Oprócz nich będzie potrzebny program rysujący lustro; jeśli jest widoczna strona lustra, w której obiekty się odbijają, to zadaniem tego programu będzie nałożenie na lustro tekstury przedstawiającej wykonany wcześniej obraz tych obiektów.

[4] Niech mi ktoś na przykład wytłumaczy, skąd i po co jest to zdrobnienie?

[5] Sposób tworzenia i przykład zastosowania bufora ramki bez załączników jest przedstawiony w p. 29.2.5.

Program rysujący lustro składa się z dwóch szaderów, wierzchołków i fragmentów, pokazanych na listingach 20.1 i 20.2.

Na początku szadera wierzchołków jest zadeklarowany blok zmiennych jednolitych `TransBlock`, identyczny jak we wszystkich innych szaderach tej aplikacji, w których ten blok występuje. Oprócz struktury `gl_PerVertex` (z polem `gl_Position`, do którego szader przypisuje współrzędne wierzchołka w układzie kostki standardowej) wyjście zawiera zmienną `TexCoord`. Na podstawie numeru wierzchołka (te numery odpowiadają indeksom wierzchołków lustra, zobacz rys. 20.1) szader wierzchołków generuje współrzędne tekstury. Prostokąt lustra jest w układzie współrzędnych tekstury kwadratem jednostkowym, pokrywającym się z dziedziną tekstury, którą nałożymy na lustro.

Listing 20.1. Szader wierzchołków lustra

GLSL

```
1: #version 450 core
2:
3: layout(location=0) in vec4 in_Position;
4:
5: uniform TransBlock {
6:     mat4 mm, mmti, vm, pm, vpm;
7:     vec4 eyepos;
8:   } trb;
9:
10: layout(location=0) out vec2 TexCoord;
11:
12: void main ( void )
13: {
14:   switch ( gl_VertexID ) {
15: default: TexCoord = vec2 ( 0.0, 0.0 );  break;
16:  case 1: TexCoord = vec2 ( 1.0, 0.0 );  break;
17:  case 2: TexCoord = vec2 ( 1.0, 1.0 );  break;
18:  case 3: TexCoord = vec2 ( 0.0, 1.0 );  break;
19:   }
20:   gl_Position = trb.vpm * (trb.mm * in_Position);
21: } /*main*/
```

Szader fragmentów na listingu 20.2 jest bardzo prosty, ponieważ jego jedynym zadaniem jest przekazanie na wyjście koloru pobranego z tekstury, bez żadnych zmian; wszelkie obliczenia oświetlenia zostały wykonane wcześniej, podczas tworzenia tej tekstury. Wyrażenie warunkowe w linii 11 dla lustra obróconego drugą stroną do obserwatora podaje na wyjście kolor ciemnoszary; „z tyłu" lustro jest zwykłym nieprzezroczystym prostokątem. Widzimy tu zastosowanie wbudowanej w GLSL zmiennej interfejsu `gl_FrontFacing` typu `bool`, która umożliwia szaderowi fragmentów rozróżnienie stron płaszczyzny rysowanego trójkąta.

Warto zwrócić uwagę na nieobecność korekcji gamma po obliczeniu koloru przez ten szader fragmentów. Korekcji gamma po obliczeniu koloru na podstawie wybranego modelu oświetlenia dokonuje pierwszy program, w związku z czym obraz otrzymany w buforze pozaekranowym jest gotów do nałożenia na lustro bez dodatkowej korekty — gdyby tu została dokonana jeszcze raz, to obraz odbity w lustrze miałby zniekształcone, zbyt jasne kolory.

Listing 20.2. Szader fragmentów lustra

GLSL

```
#version 450 core

layout(location=0) in vec2 TexCoord;

out vec4 out_Colour;

uniform sampler2D tex;

void main ( void )
{
  out_Colour = gl_FrontFacing ? vec4(0.3) : texture ( tex, TexCoord );
} /*main*/
```

Listing 20.3 przedstawia procedurę, która kompiluje opisane wyżej szadery i łączy je w program, którego identyfikator zostaje przypisany zmiennej wskazywanej przez parametr. W linii 12 następuje przywiązanie bloku zmiennych jednolitych `TransBlock` w tym programie do punktu dowiązania ustalonego po przygotowaniu programów rysowania płatów Béziera i ich siatek kontrolnych, co należało zrobić wcześniej.

Listing 20.3. Procedura przygotowania programu do rysowania lustra

C

```
void LoadMirrorShaders ( GLuint *program_id )
{
  static const char *filename[] =
    { "app2e2.vert.glsl", "app2e2.frag.glsl" };
  static const GLuint shtype[] = { GL_VERTEX_SHADER, GL_FRAGMENT_SHADER };
  GLuint shader_id[2];
  int    i;

  for ( i = 0; i < 2; i++ )
    shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
  *program_id = LinkShaderProgram ( 2, shader_id, "2" );
  AttachUniformTransBlockToBP ( *program_id );
  for ( i = 0; i < 2; i++ )
    glDeleteShader ( shader_id[i] );
  ExitIfGLError ( "LoadMirrorShaders" );
} /*LoadMirrorShaders*/
```

20.4. Procedury obsługi lustra

Listing 20.4 przedstawia definicję struktury danych opisujących lustro; pole tego typu jest dodane do struktury danych części graficznej aplikacji. Pola struktury typu `Mirror` zawierają identyfikatory bufora ramki do rysowania poza ekranem oraz jego załączników, identyfikatory obiektu tablicy wierzchołków (VAO) i bufora ze współrzędnymi wierzchołków prostokąta — lustra, oraz macierz przekształcenia modelu.

Listing 20.4. Struktura opisu lustra

C

```
#define MIRRORTXT_W 1024
#define MIRRORTXT_H 1024

typedef struct Mirror {
    GLuint  mirror_fbo, mirror_txt[2];
    GLuint  mirror_vao, mirror_vbo;
    GLfloat mirror_matrix[16];
  } Mirror;
```

Na listingu 20.5 są pokazane procedury, z których pierwsza tworzy obiekty potrzebne do wykonania obrazu odbitego w lustrze i narysowania lustra, a druga sprząta. Konstruowanie lustra zaczyna się od utworzenia dwóch tekstur, które będą załącznikami pozaekranowego bufora ramki. W pierwszej z nich, o własnościach nadawanych w liniach 9–13, ma powstać obraz, a druga (linie 14–16) będzie buforem głębokości używanym podczas wykonywania tego obrazu. Przywiązanie tekstur do celu `GL_TEXTURE_2D` czyni z nich obu tekstury dwuwymiarowe. Szerokość i wysokość są określone przez makrodefinicje na listingu 20.4. Ostatni parametr procedury `glTexImage2D`, wywołanej w liniach 12–13 i 15–16 jest wskaźnikiem pustym, bo zadaniem tej procedury jest rezerwacja pamięci na teksele, bez przesyłania danych z pamięci CPU[6]. Pozostałe parametry określają wewnętrzny format tekseli: mają one trzy składowe (r, g, b — nie potrzebujemy składowej alfa, więc nie zużywamy na nią miejsca w pamięci GPU) reprezentowane za pomocą liczb ośmiobitowych bez znaku (`GL_UNSIGNED_BYTE`). Drugi parametr określa tworzony poziom mipmapingu; w tym przypadku poziom jest tylko jeden (ma numer `0`).

Każdy teksel tekstury używanej jako bufor głębokości przechowuje jedną 32-bitową liczbę zmiennopozycyjną (`GL_FLOAT`). Parametr określający format tekseli w tym przypadku musi mieć wartość `GL_DEPTH_COMPONENT`.

Listing 20.5. Procedury tworzenia i likwidacji lustra

C

```
static const GLfloat mvertpos[4][3] =
  {{1.5,-1.2,-1.0},{1.5,1.2,-1.0},{1.5,1.2,1.0},{1.5,-1.2,1.0}};

```

[6] I w ogóle aplikacja nie będzie przesyłać żadnych danych z pamięci CPU do tych tekstur z pamięci GPU — tylko GPU będzie coś do tych tekstur pisać i je czytać.

```
4: void ConstructMirror ( Mirror *mirror )
5: {
6:   GLenum status;
7:
8:   glGenTextures ( 2, mirror->mirror_txt );
9:   glBindTexture ( GL_TEXTURE_2D, mirror->mirror_txt[0] );
10:  glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
11:  glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
12:  glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGB, MIRRORTXT_W, MIRRORTXT_H,
13:                 0, GL_RGB, GL_UNSIGNED_BYTE, NULL );
14:  glBindTexture ( GL_TEXTURE_2D, mirror->mirror_txt[1] );
15:  glTexImage2D ( GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, MIRRORTXT_W,
16:                 MIRRORTXT_H, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL );
17:  glGenFramebuffers ( 1, &mirror->mirror_fbo );
18:  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, mirror->mirror_fbo );
19:  glFramebufferTexture ( GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
20:                         mirror->mirror_txt[0], 0 );
21:  glFramebufferTexture ( GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
22:                         mirror->mirror_txt[1], 0 );
23:  if ( (status = glCheckFramebufferStatus ( GL_DRAW_FRAMEBUFFER )) !=
24:       GL_FRAMEBUFFER_COMPLETE )
25:    ExitOnError ( "ConstructMirror" );
26:  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
27:  glBindTexture ( GL_TEXTURE_2D, 0 );
28:
29:  glGenVertexArrays ( 1, &mirror->mirror_vao );
30:  glBindVertexArray ( mirror->mirror_vao );
31:  glGenBuffers ( 1, &mirror->mirror_vbo );
32:  glBindBuffer ( GL_ARRAY_BUFFER, mirror->mirror_vbo );
33:  glBufferData ( GL_ARRAY_BUFFER,
34:                 4*3*sizeof(GLfloat), mvertpos, GL_STATIC_DRAW );
35:  glEnableVertexAttribArray ( 0 );
36:  glVertexAttribPointer ( 0, 3, GL_FLOAT, GL_FALSE,
37:                          3*sizeof(GLfloat), (GLvoid*)0 );
38:  glBindVertexArray ( 0 );
39:  M4x4Identf ( mirror->mirror_matrix );
40:  ExitIfGLError ( "ConstructMirror" );
41: } /*ConstructMirror*/
42:
43: void DeleteMirror ( Mirror *mirror )
44: {
45:  glDeleteFramebuffers ( 1, &mirror->mirror_fbo );
46:  glDeleteTextures ( 2, mirror->mirror_txt );
47:  glDeleteBuffers ( 1, &mirror->mirror_vbo );
48:  glDeleteVertexArrays ( 1, &mirror->mirror_vao );
49:  ExitIfGLError ( "DeleteMirror" );
50: } /*DeleteMirrorVAO*/
```

W linii 17 jest tworzony bufor ramki. Przywiązanie go (w linii 18) do celu `GL_DRAW_FRAMEBUFFER` uaktywnia go i umożliwia określenie jego załączników. Wykonuje to procedura `glFramebufferTexture`, która przyczepia do niego utworzone wcześniej tekstury. Drugi parametr określa rolę załącznika — pierwsza tekstura staje się buforem obrazu, a druga buforem głębokości. W liniach 23–24 następuje sprawdzenie, czy bufor ramki jest gotowy do pracy; procedura `glCheckFramebufferStatus` bada, czy bufor ramki ma wszystkie konieczne załączniki[7]. Wartość podawana przez tę procedurę wskazuje, że wszystko jest OK, albo podaje informację o wykrytym błędzie. W opisanej tu aplikacji w razie błędu następuje rezygnacja z dalszego działania.

Uwaga: Fakt, że na jednym komputerze został utworzony poprawny i kompletny bufor ramki *nie jest* gwarancją sukcesu na innym komputerze — może na nim zabraknąć pamięci GPU lub mogą wystąpić inne, lokalne przyczyny niepowodzenia.

Instrukcja w linii 26 przywiązuje do celu `GL_DRAW_FRAMEBUFFER` domyślny bufor ramki związany z oknem, co umożliwia rysowanie w oknie. W linii 27 tekstura jest odwiązywana od celu, co zabezpiecza ją przed przypadkową zmianą parametrów.

W liniach 29–37 jest tworzony obiekt tablicy wierzchołków i bufor ze współrzędnymi tych wierzchołków, branymi z tablicy `mvertpos`. Działanie wywoływanych w tym celu procedur jest opisane w p. 7.2.2. Obiekt tablicy wierzchołków jest w linii 38 odaktywniany; będzie ponownie uaktywniany tylko na czas rysowania lustra i jego zawartość się nie zmieni.

W linii 39 do tablicy `mirror_matrix` są wpisywane współczynniki macierzy jednostkowej — jest to macierz przejścia od układu modelu do układu świata dla lustra.

Sprzątanie lustra, wykonywane przez procedurę `DeleteMirror`, polega na zwolnieniu bufora ramki i jego załączników oraz obiektu tablicy wierzchołków i bufora z ich współrzędnymi.

Na listingu 20.6 jest pokazana procedura `SetupMirrorVPMatrices`, której zadaniem jest obliczenie położenia odbitego w lustrze obserwatora oraz macierzy przejścia od układu współrzędnych świata do układu odbitego obserwatora i od tego ostatniego układu do kostki standardowej. Procedura ta wykonuje tylko obliczenie współczynników tych macierzy, bez przesyłania czegokolwiek do pamięci GPU. Parametrem wejściowym jest tablica `eyepos` ze współrzędnymi położenia obserwatora (w układzie świata). W tablicy `reyepos` procedura ma umieścić współrzędne położenia odbitego obserwatora, a w tablicach `mvm` i `mpm` współczynniki wspomnianych macierzy przekształceń.

Ponieważ macierze są przechowywane w porządku *kolumnowym*, można zrobić to, co zostało zrobione w linii 8: przypisać dodatkowym wskaźnikom adresy początków poszczególnych kolumn roboczej tablicy `mfm`, w której utworzymy macierz V^{-1} daną wzorem (20.1). W pętli w liniach 9–13 są obliczane współrzędne wektorów $\boldsymbol{v}_1$ i $\boldsymbol{v}_2$ oraz współrzędne wektora $\boldsymbol{p}_0 - \boldsymbol{e}$ potrzebnego do ustalenia właściwego zwrotu wektora $\boldsymbol{n}$. Wektor ten jest obliczany w liniach 14–18. W linii 19 następuje obliczenie współrzędnych położenia obserwatora, które następnie jest kopiowane do tablicy `mfm` jako początek ostatniej kolumny macierzy V^{-1}.

[7] Bufor ramki powinien mieć co najmniej jeden załącznik, np. bufor obrazu lub bufor głębokości, ale do celów specjalnych można utworzyć bufor ramki bez załączników. Opis, jak go użyć, będzie w rozdziale 29.

W linii 21 tworzony jest ostatni wiersz tej macierzy, po czym następuje wywołanie procedury `M4x4Invertf`, którego skutkiem jest obliczenie macierzy V opisującej przejście od układu świata do układu odbitego obserwatora. W liniach 23, 24 obliczamy współrzędne wierzchołków lustra $\boldsymbol{p}_0$ i $\boldsymbol{p}_3$ w układzie odbitego obserwatora, aby następnie podstawić je jako parametry wywołania procedury `M4x4Frustumf`, która konstruuje macierz przekształcenia do układu kostki standardowej.

Listing 20.6. Tworzenie macierzy przekształceń i rysowanie lustra

C

```
1: void SetupMirrorVPMatrices ( GLfloat eyepos[4], GLfloat reyepos[4],
2:                              GLfloat mvm[16], GLfloat mpm[16] )
3: {
4:   GLfloat mfm[16], *v1, *v2, *nv, *p;
5:   GLfloat s, a[3], b[3];
6:   int     i;
7:
8:   v1 = &mfm[0], v2 = &mfm[4], nv = &mfm[8], p = &mfm[12];
9:   for ( i = 0; i < 3; i++ ) {
10:    v1[i] = mvertpos[1][i]-mvertpos[0][i];
11:    v2[i] = mvertpos[3][i]-mvertpos[0][i];
12:    a[i] = mvertpos[0][i]-eyepos[i];
13:  }
14:  V3CrossProductf ( nv, v2, v1 );
15:  s = sqrt ( V3DotProductf ( nv, nv ) );
16:  if ( V3DotProductf ( nv, a ) < 0.0 )
17:    s = -s;
18:  nv[0] /= s;  nv[1] /= s;  nv[2] /= s;
19:  V3ReflectPointf ( reyepos, mvertpos[0], nv, eyepos );
20:  memcpy ( p, reyepos, 3*sizeof(GLfloat) );
21:  mfm[3] = mfm[7] = mfm[11] = 0.0;  mfm[15] = 1.0;
22:  M4x4Invertf ( mvm, mfm );
23:  M4x4MultMP3f ( a, mvm, mvertpos[0] );
24:  M4x4MultMP3f ( b, mvm, mvertpos[2] );
25:  M4x4Frustumf ( mpm, NULL, a[0], b[0], a[1], b[1], -a[2], 20.0 );
26: } /*SetupMirrorVPMatrices*/
27:
28: void DrawMirror ( Mirror *mirror, GLuint program_id )
29: {
30:  glUseProgram ( program_id );
31:  glBindVertexArray ( mirror->mirror_vao );
32:  glBindTexture ( GL_TEXTURE_2D, mirror->mirror_txt[0] );
33:  glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
34:  glDrawArrays ( GL_TRIANGLE_FAN, 0, 4 );
35:  glBindVertexArray ( 0 );
36:  ExitIfGLError ( "DrawMirror" );
37: } /*DrawMirror*/
```

Procedura rysowania lustra jest bardzo prosta, ponieważ wszelkie skomplikowane przygotowania do rysowania lustra, w tym przesłanie do pamięci GPU (do bufora z blokiem zmiennych jednolitych `TransBlock`) macierzy przekształcenia modelu, są wykonywane przed jej wywołaniem. Należy wybrać program szaderów (złożony z szaderów na listingach 20.1 i 20.2), którego identyfikator jest podany jako parametr, uczynić bieżącym VAO z wierzchołkami lustra, przywiązać do celu `GL_TEXTURE_2D` teksturę z widokiem sceny w lustrze, ustawić wypełnianie wielokątów i narysować prostokąt (jako wachlarz złożony z dwóch trójkątów).

Uwaga: Wywołanie procedury `glPolygonMode` ma na celu zapewnienie, że obraz lustra składa się z wypełnionych trójkątów, a nie tylko z ich krawędzi. W domyślnym stanie kontekstu OpenGL-a trójkąty są wypełniane, co mogło być zmienione podczas rysowania czajnika i torusa. Jeśli narysowanie jakiegoś obiektu wymaga zmiany stanu kontekstu, to autor aplikacji ma do wyboru przywracanie stanu domyślnego natychmiast po narysowaniu obiektu lub założenie, że stan (zmienianych przez aplikację) danych w kontekście OpenGL-a jest nieokreślony, co wymaga nadania im wartości przed rysowaniem obiektu.

20.5. Zmiany w aplikacji

Zmiany, których trzeba dokonać, aby z aplikacji drugiej D powstała aplikacja druga E, nie są wielkie, ale są dosyć liczne. Listing 20.7 przedstawia strukturę danych aplikacji z wyróżnionymi nowymi polami: `mirror`, zawierającym opisaną w poprzednim podrozdziale reprezentację lustra, i `miprog`, przechowującym identyfikator programu szaderów do rysowania lustra.

Listing 20.7. Zmieniona struktura `AppData`

C

```
typedef struct {
    Camera           camera;
    BezierPatchObjf  *myteapot, *mytorus;
    Mirror           mirror;
    TransBl          trans;
    LightBl          light;
    MatBl            mat;
    GLuint           mytexture;
    GLint            BezNormals, TessLevel;
    char             cnet, skeleton, animate;
    float            model_rot_axis[3];
    double           teapot_rot_angle, torus_rot_angle;
    GLfloat          teapot_mmatrix[16], torus_mmatrix[16];
    BPRenderPrograms brprog;
    GLuint           miprog;
} AppData;
```

Oprócz instrukcji wykonywanych w poprzedniej wersji procedura inicjalizacji ma wywołać procedury `LoadMirrorShaders` i `ConstructMirror` z listingów 20.3 i 20.5. Ponieważ numer punktu dowiązania dla bloku zmiennych jednolitych `TransBlock`, używanego przez wszystkie programy szaderów, jest ustalany przez procedurę przygotowującą programy do rysowania płatów Béziera, wywołanie procedury `LoadBPShaders` musi *poprzedzać* wywołanie procedury przygotowującej program do rysowania lustra.

Listing 20.8. Zmiany w procedurze inicjalizacji

C

```
void InitMyWorld ( int argc, char *argv[], int width, int height )
{
  float axis[4] = {0.0,0.0,1.0};

  memset ( &appdata, 0, sizeof(AppData) );
  LoadBPShaders ( &appdata.brprog );
  LoadMirrorShaders ( &appdata.miprog );
  appdata.trans.trbuf = NewUniformTransBlock ();
  .... /* tu instrukcje pozostawione bez zmian */
  ConstructMyTorus ( &appdata );
  ConstructMirror ( &appdata.mirror );
  InitLights ( &appdata );
  appdata.mytexture = LoadMyTextures ();
} /*InitMyWorld*/
```

Strukturę `TransBl` (która jest typem pola `trans`) zdefiniowaną na listingu 10.6 zmienimy w sposób pokazany na listingu 20.9. W polach `mm` i `mmti` będzie przechowywana macierz przekształcenia modelu i transpozycja jej odwrotności. W polach `wvm` i `wpm` będą zapamiętane macierze przejścia do układu obserwatora i kostki dla rzutu podczas rysowania sceny w oknie, a pola `mvm` i `mpm` są przeznaczone dla odpowiednich macierzy do rysowania na teksturze lustra. W polu `eyepos` jest przechowywane położenie obserwatora, a w polu `reyepos` położenie obserwatora odbitego w lustrze.

Listing 20.9. Zmieniona struktura `TransBl`

C

```
typedef struct TransBl {
    GLfloat mm[16], mmti[16],
            wvm[16], wpm[16],
            mvm[16], mpm[16];
    GLfloat eyepos[4], reyepos[4];
  } TransBl;
```

W procedurze `_ResizeMyWorld` (listing 15.16) ostatnią instrukcję trzeba zastąpić wywołaniem procedury `SetupMirrorVPMatrices`, która na podstawie położenia obserwatora, pamiętanego w polu `trans.eyepos`, obliczy położenie obserwatora odbitego w lustrze i macierze przejścia do układu kostki standardowej dla obu rzutów perspektywicznych. Przesłanie tych macierzy do pamięci GPU, tak jak i ustawienie wielkości klatki, będzie wykonywane

bezpośrednio przed rysowaniem, zatem nie trzeba też wywoływać w tym miejscu procedury `glViewport`, ale trzeba zapamiętać nowe wymiary okna.

Listing 20.10. Procedura `_ResizeMyWorld`

C

```
static void _ResizeMyWorld ( AppData *ad, int width, int height )
{
  float lr;

  ad->camera.win_width = width;  ad->camera.win_height = height;
  lr = 0.5533*(float)width/(float)height;
  M4x4Frustumf ( ad->trans.wpm, NULL,
                 ad->camera.left = -lr, ad->camera.right = lr,
                 ad->camera.bottom = -0.5533, ad->camera.top = 0.5533,
                 ad->camera.near = 5.0, ad->camera.far = 15.0 );
  ad->camera.rl = 2.0*lr;  ad->camera.tb = 2.0*0.5533;
  SetupMirrorVPMatrices ( ad->trans.eyepos, ad->trans.reyepos,
                          ad->trans.mvm, ad->trans.mpm );
} /*_ResizeMyWorld*/
```

Zmiana struktury `TransBl` wymaga dostosowania procedury `LoadVPMatrix` (listing 12.9), której nowa wersja, pokazana na listingu 20.11, ma dodatkowy parametr. Zależnie od jego wartości do bufora zawierającego blok zmiennych jednolitych `TransBlock` są przesyłane macierze opisujące odpowiednie rzutowanie z oryginalnego lub odbitego punktu położenia obserwatora.

Listing 20.11. Zmieniona procedura `LoadVPMatrix`

C

```
void LoadVPMatrix ( TransBl *trans, char mirror )
{
  GLfloat *vm, *pm, *ep, vpm[16];

  if ( mirror )
    { vm = trans->mvm;  pm = trans->mpm;  ep = trans->reyepos; }
  else
    { vm = trans->wvm;  pm = trans->wpm;  ep = trans->eyepos; }
  M4x4Multf ( vpm, pm, vm );
  glBindBuffer ( GL_UNIFORM_BUFFER, trans->trbuf );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[2], 16*sizeof(GLfloat), vm );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[3], 16*sizeof(GLfloat), pm );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[4], 16*sizeof(GLfloat), vpm );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[5], 4*sizeof(GLfloat), ep );
  ExitIfGLError ( "LoadVPMatrix" );
} /*LoadPMatrix*/
```

Na listingu 20.12 jest pokazana nowa procedura zmiany położenia obserwatora. Dokonane w niej modyfikacje polegają na użyciu nowych nazw pól struktury `TransBl` i na zastąpieniu wywołania procedury `LoadVPMatrix` przez `SetupMirrorVPMatrices`.

Listing 20.12. Procedura zmiany położenia obserwatora

C

```
1: static void _RotateViewer ( AppData *ad, double delta_xi, double delta_eta )
2: {
3:   float   vi[3], lgt, vk[3];
4:   double  angi, angk;
5:   GLfloat tm[16];
6:
7:   .... /* początkowe instrukcje bez zmian, zobacz listing 15.17 */
8:   M4x4RotateVfv ( ad->trans.wvm, ad->camera.viewer_rvec,
9:                   -ad->camera.viewer_rangle );
10:  M4x4MultMTVf ( ad->trans.eyepos, ad->trans.wvm, ad->camera.viewer_pos0 );
11:  M4x4InvTranslateMfv ( ad->trans.wvm, ad->camera.viewer_pos0 );
12:  SetupMirrorVPMatrices ( ad->trans.eyepos, ad->trans.reyepos,
13:                           ad->trans.mvm, ad->trans.mpm );
14: } /*_RotateViewer*/
```

Zobaczmy teraz procedury rysowania (listing 20.13); wywoływana przez procedury z części okienkowej procedura `RedrawMyorld` włącza test głębokości, a następnie wywołuje kolejno dwie procedury, których celem jest utworzenie obrazu sceny widzianej w lustrze i końcowego obrazu w oknie. Każda z tych procedur uaktywnia odpowiedni bufor ramki (`DrawSceneToMirror` pozaekranowy, `DrawSceneToWindow` związany z oknem), ustawia klatkę, przesyła do bloku zmiennych jednolitych `TransBlock` macierze odpowiedniego rzutowania, kasuje tło i bufor głębokości, a następnie wywołuje pomocniczą procedurę `DrawScene`, której zadaniem jest narysowanie wszystkich obiektów *oprócz* lustra. Procedura `DrawSceneToWindow` rysuje lustro, a potem czajnik i torus. Obie procedury kończą działanie wywołaniem procedury `glFlush`, która informuje, że wszystkie dane do narysowania zostały przekazane i należy obraz jak najszybciej dokończyć.

Kolor nadawany pikselom przed rysowaniem odbicia obiektów w lustrze (linia 18) nie jest całkiem biały, tylko nieco ciemniejszy szary, aby lustro było widoczne na końcowym obrazie.

Listing 20.13. Procedury rysowania sceny

C

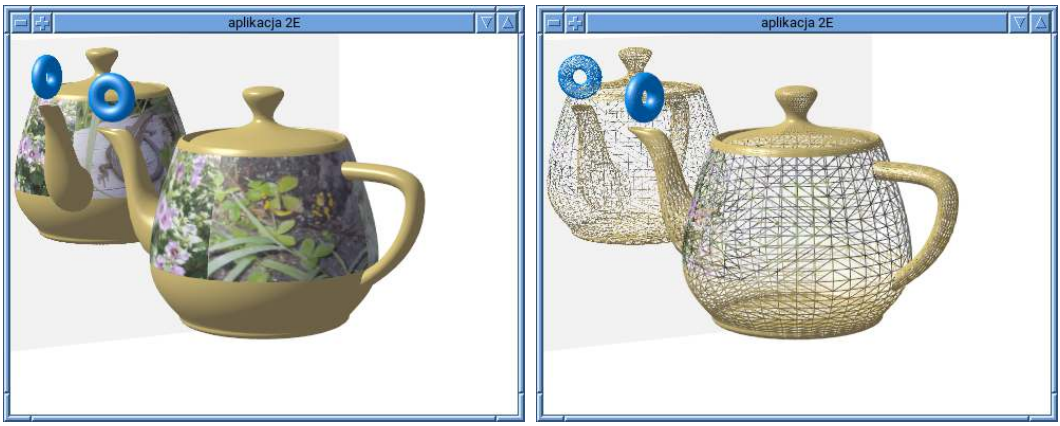
```
1: void DrawScene ( AppData *ad )
2: {
3:   LoadMMatrix ( &ad->trans, ad->teapot_mmatrix );
4:   DrawMyTeapot ( ad );
5:   if ( ad->cnet )
6:     DrawMyCNet ( ad->myteapot, ad->brprog.program_id[1] );
7:   LoadMMatrix ( &ad->trans, ad->torus_mmatrix );
8:   DrawMyTorus ( ad );
```

```
9:   if ( ad->cnet )
10:    DrawMyCNet ( ad->mytorus, ad->brprog.program_id[1] );
11: } /*DrawScene*/
12:
13: void DrawSceneToMirror ( AppData *ad )
14: {
15:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, ad->mirror.mirror_fbo );
16:   glViewport ( 0, 0, MIRRORTXT_W, MIRRORTXT_H );
17:   LoadVPMatrix ( &ad->trans, true );
18:   glClearColor ( 0.95, 0.95, 0.95, 1.0 );
19:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
20:   DrawScene ( ad );
21:   glFlush ();
22: } /*DrawSceneToMirror*/
23:
24: void DrawSceneToWindow ( AppData *ad )
25: {
26:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
27:   glViewport ( 0, 0, ad->camera.win_width, ad->camera.win_height );
28:   LoadVPMatrix ( &ad->trans, false );
29:   glClearColor ( 1.0, 1.0, 1.0, 1.0 );
30:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
31:   LoadMMatrix ( &ad->trans, ad->mirror.mirror_matrix );
32:   DrawMirror ( ad, ad->miprog );
33:   DrawScene ( ad );
34:   glFlush ();
35: } /*DrawSceneToWindow*/
36:
37: void RedrawMyWorld ( void )
38: {
39:   glEnable ( GL_DEPTH_TEST );
40:   DrawSceneToMirror ( &appdata );
41:   DrawSceneToWindow ( &appdata );
42: } /*RedrawMyWorld*/
```

20.6. Uzupełnienia

20.6.1. Skróty w OpenGL-u 4.5

„Tradycyjna" w OpenGL-u metoda tworzenia buforów i tekstur polega na zarezerwowaniu identyfikatorów (za pomocą procedury `glGenBuffers` albo `glGenTextures`). Następnie identyfikator trzeba przywiązać do odpowiedniego celu, na przykład `GL_UNIFORM_BUFFER` lub `GL_TEXTURE_2D`, a następnie wywołać odpowiednie procedury, których pierwszy parametr jest identyfikatorem *celu*, do którego jest przywiązany identyfikator, i które określają własności bufora lub tekstury lub przesyłają do nich dane. W szczególności rodzaj (tj. wy-



Rysunek 20.2. Okno aplikacji drugiej E

miar) tekstury jest określony przez cel, do którego identyfikator został przywiązany po jego zarezerwowaniu.

W specyfikacji OpenGL 4.5 pojawiły się nowe procedury, wygodniejsze w użyciu, bo umożliwiające działania z buforami i teksturami bez pośrednictwa celu — pierwszym parametrem tych procedur jest identyfikator bufora albo tekstury, a nie celu. Przy użyciu tych procedur jest możliwe przesłanie danych lub określenie parametrów bez przywiązywania obiektu (bufora lub tekstury) do celu, co jest określane skrótem DSA — *direct state access*.

Nowe procedury obsługi buforów mają słowo `Named` w nazwach; na przykład odpowiednikami „tradycyjnych" procedur `glBufferData`, `glBufferSubData`, `glCopyBufferSubData` są procedury `glNamedBufferData`, `glNamedBufferSubData` i `glCopyNamedBufferSubData`; dokładniejszy ich opis jest na stronie [7]. Trzeba jednak pamiętać, że pewne procedury wymagają, aby odpowiednie bufory były przywiązane do właściwych celów. Na przykład przed wywołaniem procedury `glDrawElements` należy bufor z indeksami wierzchołków przywiązać do celu `GL_ELEMENT_ARRAY_BUFFER`[8].

Z kolei tradycyjne procedury obsługi tekstur mają w nazwach skrót `Tex`, podczas gdy „nowe" procedury mają pełne słowo `Texture`. Do zarezerwowania identyfikatorów tekstur z jednoczesnym określeniem ich wymiarów zamiast `glGenTextures` można wywołać procedurę `glCreateTextures`; pierwszy jej parametr jest identyfikatorem celu (np. `GL_TEXTURE_2D`), drugi jest liczbą identyfikatorów, a trzeci jest tablicą, do której te identyfikatory mają być wpisane. Procedury z rodziny `glTexParameter*` (np. `glTexParameteri`, `glTexParameterfv`) mają odpowiedniki o nazwach `glTextureParameter*` (np. `glTextureParameteri`, `glTextureParameterfv`). Pierwszy parametr każdej z tych procedur jest identyfikatorem *tekstury* (a nie celu, do którego jest przywiązana tekstura), a pozostałe parametry są takie jak w „starych" procedurach.

Niestety, wśród „nowych" procedur nie ma procedur przesyłających dane z pamięci CPU, a zatem aby umieścić w teksturze potrzebne obrazy, trzeba przywiązać obiekt tekstury do

[8] To najlepiej jest zrobić za pośrednictwem procedury `glBindVertexArray`.

odpowiedniego celu (np. GL_TEXTURE_2D) i wywołać „starą” procedurę (np. glTexImage2D albo glTexStorage2D i glTexSubImage2D).

20.6.2. Bufory robocze

Alternatywnym dla tekstur rodzajem załączników pozaekranowego bufora ramki są bufory robocze (*renderbuffers*). Są one lepiej dostosowane do tworzenia obrazów (bo są zoptymalizowane ze względu na szybkość zapisu), ale nie mogą być użyte zamiast tekstur do nakładania na rysowane później obiekty. Obraz z bufora roboczego może być szybko przesłany do innych buforów ramki — na przykład do bufora związanego z oknem aplikacji. Aplikacja może też odczytać dane (tj. obraz lub jego część) z bufora roboczego, aby zapisać je w pliku lub przetworzyć w dowolny inny sposób.

Bufor ramki, którego załącznikami są bufory robocze o wymiarach $w \times h$ pikseli tworzy taki przykładowy ciąg instrukcji:

```
GLuint rbuf[2], fb;

glGenRenderbuffers ( 2, rbuf );
glBindRenderbuffer ( GL_RENDERBUFFER, (GLuint)rbuf[0] );
glRenderbufferStorage ( GL_RENDERBUFFER, GL_RGBA8, w, h );
glBindRenderbuffer ( GL_RENDERBUFFER, (GLuint)rbuf[1] );
glRenderbufferStorage ( GL_RENDERBUFFER, GL_DEPTH_COMPONENT, w, h );
glBindRenderbuffer ( GL_RENDERBUFFER, 0 );
glGenFramebuffers ( 1, &fb );
glBindFramebuffer ( GL_FRAMEBUFFER, (GLuint)fb );
glFramebufferRenderbuffer ( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                            GL_RENDERBUFFER, rbuf[0] );
glFramebufferRenderbuffer ( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                            GL_RENDERBUFFER, rbuf[1] );
if ( glCheckFramebufferStatus ( GL_FRAMEBUFFER ) !=
     GL_FRAMEBUFFER_COMPLETE ) ExitOnError ( "" );
glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
```

W tym przykładzie pierwszy z dwóch buforów roboczych utworzonych przez procedurę glGenRenderbuffers staje się buforem obrazu, a drugi buforem głębokości. Wielkość i rodzaj oraz format danych bufora roboczego są określane po przywiązaniu (za pomocą procedury glBindRenderbuffer) bufora roboczego do do celu GL_RENDERBUFFER przez wywołanie procedury glRenderbufferStorage. Utworzone bufory robocze stają się załącznikami bufora ramki wskutek wywołań procedury glFramebufferRenderbuffer, co działa analogicznie do przywiązywania załącznika — tekstury. Do likwidacji buforów roboczych (podczas sprzątania) służy procedura glDeleteRenderbuffers.

W OpenGL-u 4.5 i 4.6 jest dostępna procedura glNamedRenderbufferStorage, której pierwszy parametr to identyfikator bufora roboczego, a pozostałe parametry są takie jak

w procedurze `glRenderbufferStorage`. Dzięki niej przestaje być konieczne używanie procedury `glBindRenderbuffer`, co umożliwia skrócenie kodu aplikacji.

Niezależnie od tego, czy załącznikami są tekstury, czy bufory robocze, dane (tj. zawartość wskazanego prostokąta) między buforami ramki można szybko przesyłać za pomocą procedury `glBlitFramebuffer` lub `glBlitNamedFramebuffer`. Wywołanie pierwszej z tych procedur musi być poprzedzone przywiązaniem buforów ramki odpowiednio do celów `GL_READ_FRAMEBUFFER` i `GL_DRAW_FRAMEBUFFER`; druga procedura wymaga podania identyfikatorów buforów ramki w pierwszych dwóch parametrach. Kolejne parametry to dwie czwórki liczb, opisujące położenie i wymiary (w pikselach) prostokątów, z których dane należy odczytać i zapisać. Prostokąty te nie muszą mieć jednakowych wymiarów; jeśli nie mają, to dokonywane jest odpowiednie skalowanie i filtrowanie obrazu — wyborem steruje ostatni parametr, który może mieć wartość `GL_LINEAR` albo `GL_NEAREST`. Parametr przedostatni określa załączniki, których zawartość ma być przesłana. Może to być bufor obrazu, bufor głębokości lub bufor maski.

Jeśli utworzony w buforze ramki obraz ma być zapisany w pliku, bo jest na przykład klatką filmu animowanego, to można go odczytać (tj. przesłać do pamięci CPU) przy użyciu procedury `glReadPixels`. Dokładne opisy wspomnianych tu procedur można znaleźć w specyfikacji [1] lub na stronie [7], która jest znacznie wygodniejsza do takich poszukiwań.

20.7. Ćwiczenia

1. Wykonaj eksperymenty z lustrem „odwracającym", tzn. zamieniającym prawą stronę z lewą[9]. W tym celu zmodyfikuj szader wierzchołków na listingu 20.1, zamieniając współrzędne tekstury przyporządkowane poszczególnym wierzchołkom.
2. Rysowanie rzutu sceny na teksturze jest nieoszczędne, ponieważ ustawiamy klatkę o pełnej rozdzielczości tekstury, 1024×1024, nawet jeśli klatka z gotowym obrazem w oknie ma znacznie mniejsze wymiary. Zabiera to sporo czasu, a ponadto odcinki widoczne w lustrze na prawym rysunku 20.2 są znacznie cieńsze niż odcinki bezpośredniego obrazu czajnika. Zmodyfikuj aplikację (szadery i kod w C) tak, aby ustawiać wielkość klatki odpowiednio do wielkości obrazu w lustrze. Wymaga to m.in. zmienienia współrzędnych w układzie tekstury przyporządkowanych wierzchołkom lustra. Jeśli na przykład klatka miałaby wielkość połowy tekstury, to przedział zmienności współrzędnych tekstury powinien mieć długość $\frac{1}{2}$.
3. Spróbuj uruchomić antyaliasing podczas nakładania tekstury na lustro. W tym celu dla obrazu (nie dla bufora głębokości) utwórz teksturę wielopoziomową, a następnie, po wykonaniu obrazu na poziomie `0` tekstury (na końcu procedury `DrawSceneToMirror`) wywołaj procedurę `glGenerateTextureMipmap`.
4. Zmień szadery programu do rysowania lustra tak, aby tylna strona lustra była „wykonana z określonego materiału" i odpowiednio oświetlona.

[9] To nieprawda, że zwykłe lustro zamienia stronę prawą z lewą. Lustro, jeśli można tak powiedzieć, zamienia przód z tyłem.

5. Wymodeluj lustro niedoskonałe, na przykład zabrudzone. W tym celu zmieszaj obraz obiektów odbitych w lustrze ze światłem odbitym od zabrudzeń lustra opisanych przez dodatkową teksturę.
6. Jeśli obserwator znajduje się „za lustrem", tj. po jego nie-lustrzanej stronie, to tworzenie odbitego w lustrze obrazu sceny jest niepotrzebne, bo i tak nie będzie on widoczny. Dodaj odpowiednią instrukcję warunkową, aby nie wywoływać procedury `DrawSceneToMirror` w takiej sytuacji.
7. *Oprogramuj poruszanie lustrem (przez zmienianie macierzy modelu przechowywanej w zmiennej `mirror_matrix`); pamiętaj o uwzględnieniu przekształcenia lustra w obliczaniu macierzy przekształceń do rysowania na teksturze lustra.

21

Aplikacja druga F

Nałożymy na obiekty proceduralną **teksturę odkształceń**; zaburzanie wektora normalnego powierzchni przekazywanego do obliczeń z ustalonym modelem oświetlenia umożliwia otrzymanie obrazu powierzchni chropowatej, porysowanej, pofalowanej albo na przykład pokrytej łuskami lub dachówkami. Osiągnięcie takich efektów przy użyciu dokładnego opisu kształtu (przez wygenerowanie tablicy z wierzchołkami wszystkich wypukłości i wnęk powierzchni) byłoby skrajnie niepraktyczne. Studiowanie lepszej metody rozwiązania problemu zaczniemy od zaaplikowania sobie homeopatycznej dawki geometrii różniczkowej.

21.1. Wektor normalny zaburzonej powierzchni

Równoległobok, oraz role
Przezeń na naszym łez padole
Pełnione — to jest temat-rzeka
Znany z pism Euklidesa Greka.

A. E. Housman: *Równoległobok, albo: Dziecięcy optymizm*[1]

Rozważamy płat powierzchni parametrycznej, o parametryzacji $\boldsymbol{p}\colon A \to \mathbb{R}^3$ (w naszej aplikacji będzie to każdy z płatów Béziera, jego dziedzina A jest kwadratem jednostkowym $[0,1]^2$). **Odwzorowanie Gaussa**[2] jest to funkcja wektorowa, która każdemu punktowi $(u,v) \in A$ przyporządkowuje jednostkowy wektor normalny $\boldsymbol{n}$ płata $\boldsymbol{p}$ w punkcie $\boldsymbol{p}(u,v)$. Jest ono dane wzorem

$$\boldsymbol{n}(u,v) = \frac{\boldsymbol{m}(u,v)}{\|\boldsymbol{m}(u,v)\|}, \qquad \boldsymbol{m}(u,v) = \boldsymbol{p}_u(u,v) \wedge \boldsymbol{p}_v(u,v),$$

przy czym zakładamy, że wektory pochodnych cząstkowych $\boldsymbol{p}_u(u,v)$ i $\boldsymbol{p}_v(u,v)$ są liniowo niezależne. Wprowadzimy przekształcenie $\boldsymbol{q}$ dziedziny A w pewien obszar w $\mathbb{R}^2$ i określoną w tym obszarze funkcję d, która przyjmuje wartości rzeczywiste o *małych* wartościach

[1] Przekład Stanisława Barańczaka, Znak, 1998.

[2] Nie ma się czego bać: już od dawna korzystamy z tego odwzorowania, rysując oświetlony czajnik.

bezwzględnych. Zakładamy, że przekształcenie $\boldsymbol{q}$ i funkcja d są ciągłe i mają (kawałkami) ciągłe pochodne.

Określimy teraz parametryzację *nowej powierzchni* wzorem

$$\hat{\boldsymbol{p}}(u,v) = \boldsymbol{p}(u,v) + d\big(\boldsymbol{q}(u,v)\big)\boldsymbol{n}(u,v),$$

który możemy zapisać krócej w postaci $\hat{\boldsymbol{p}} = \boldsymbol{p} + (d \circ \boldsymbol{q})\boldsymbol{n}$. Nowa powierzchnia powstaje z powierzchni oryginalnej przez przesunięcie każdego punktu w kierunku wektora normalnego, przy czym wielkość tego przesunięcia jest wartością funkcji d. Użycie tekstury odkształceń polega na narysowaniu oryginalnej powierzchni, ale do modelu oświetlenia podstawia się wektor normalny $\hat{\boldsymbol{n}}$ płata $\hat{\boldsymbol{p}}$. Jeśli przemieszczenia (czyli wartości funkcji d) są małe (w porównaniu z rozmiarami powierzchni), to otrzymany obraz bardzo trudno jest odróżnić od obrazu powierzchni $\hat{\boldsymbol{p}}$.

Głębokości (tj. współrzędne z w układzie kostki standardowej) punktów odkształconej powierzchni są oczywiście inne niż głębokości odpowiadających im punktów gładkiej powierzchni oryginalnej. Choć różnice głębokości są zazwyczaj małe, można (i, choć to nie jest konieczne, warto) uwzględnić je w testach widoczności. Przecięcie gładkich powierzchni jest krzywą gładką, a krzywa wspólna powierzchni z nałożoną teksturą odkształceń jest odpowiednio pofalowana. W dużym powiększeniu powinno to być na obrazach widoczne.

Aby to osiągnąć, szader fragmentów musi przypisać zmiennej `gl_FragDepth` sumę głębokości fragmentu powierzchni oryginalnej i poprawki głębokości, którą musi obliczyć. Niech C oznacza macierz przejścia do układu współrzędnych kostki standardowej od układu, w którym określamy odkształcenie[3]. Poprawkę otrzymamy, mnożąc przesunięcie d punktu $\boldsymbol{p}$ w tym układzie przez liczbę $Z/2W$; W oznacza tu współrzędną wagową wektora $C\boldsymbol{P}$, a Z jest trzecią współrzędną wektora $C\boldsymbol{N}$, gdzie $\boldsymbol{P}$ to wektor współrzędnych jednorodnych punktu $\boldsymbol{p}$ (o współrzędnej wagowej 1), a wektor $\boldsymbol{N}$ otrzymamy przez dołączenie wagi 0 do jednostkowego wektora normalnego $\boldsymbol{n}$ powierzchni w tym punkcie — w układzie, w którym określamy odkształcenia[4].

Znajdziemy wzory umożliwiające obliczenie wektora normalnego $\hat{\boldsymbol{n}}$. W tym celu potrzebujemy pochodnych cząstkowych parametryzacji $\hat{\boldsymbol{p}}$, które są kolumnami jej macierzy różniczki $\mathrm{D}\hat{\boldsymbol{p}}$. Obliczanie pochodnych sumy, iloczynu i złożenia funkcji wektorowych daje się wygodnie zapisać w postaci macierzowej, w której mamy

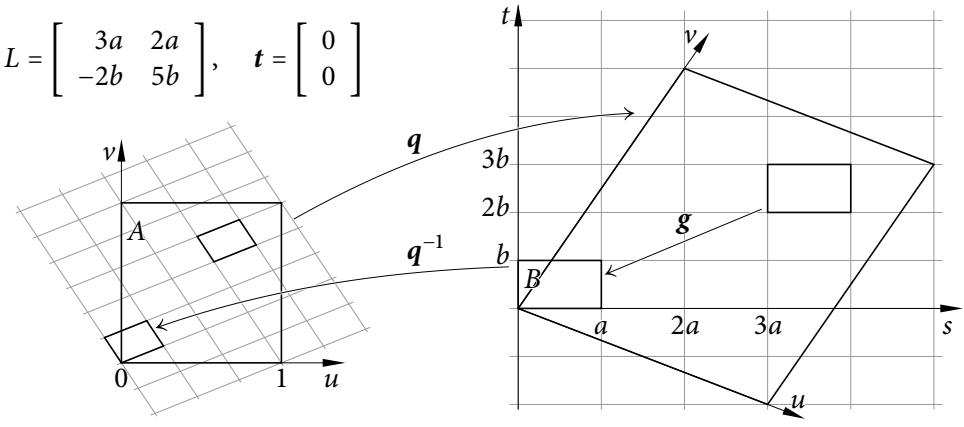
$$\mathrm{D}\hat{\boldsymbol{p}} = \mathrm{D}\boldsymbol{p} + \mathrm{D}(d \circ \boldsymbol{q}) \cdot \boldsymbol{n} + (d \circ \boldsymbol{q})\mathrm{D}\boldsymbol{n} \approx \mathrm{D}\boldsymbol{p} + (\mathrm{D}d \cdot \mathrm{D}\boldsymbol{q})\boldsymbol{n}.$$

[3] Odkształcenia powierzchni możemy określić w układzie współrzędnych modelu lub w układzie świata; w opisanej tu aplikacji jest zrealizowana ta druga możliwość, wtedy macierz $C = PV$ jest iloczynem macierzy widoku V i macierzy rzutowania perspektywicznego P. Jeśli odkształcenia są określone w układzie modelu, to trzeba przyjąć $C = PVM$, gdzie M jest macierzą przejścia od układu modelu do układu świata.

[4] Wektor $\boldsymbol{n}$, który w układzie modelu albo świata jest jednostkowym wektorem normalnym powierzchni w punkcie $\boldsymbol{p}$ (i pomnożony przez d określa jego przemieszczenie), w tym zastosowaniu trzeba przekształcać jak wektor swobodny — zatem przekształcenie go do układu kostki standardowej realizuje macierz C. Ponieważ przekształcenie perspektywiczne, realizowane przez macierz P, nie jest przekształceniem afinicznym, dokładność tej konstrukcji maleje ze wzrostem wartości bezwzględnej d. Dokładniejsza, ale bardziej kosztowna konstrukcja polega na obliczeniu punktu $\hat{\boldsymbol{p}} = \boldsymbol{p} + d\boldsymbol{n}$ i przypisaniu zmiennej `gl_FragDepth` liczby $\zeta = z/2$, gdzie z oznacza trzecią współrzędną punktu $\hat{\boldsymbol{p}}$ w układzie kostki standardowej.

W ostatnim kroku powyższego rachunku pominęliśmy składnik $(d \circ \boldsymbol{q})\mathrm{D}\boldsymbol{n}$, ponieważ jest w nim czynnik d; dzięki założeniu, że funkcja d ma małe wartości bezwzględne, ten składnik istotnie jest pomijalny[5]. Do końcowego wzoru potrzebujemy zatem podstawić macierz $\mathrm{D}\boldsymbol{p}$, której kolumny są pochodnymi cząstkowymi parametryzacji $\boldsymbol{p}$, macierz (wierszową) $\mathrm{D}d$, która jest gradientem funkcji d, macierz różniczki $\mathrm{D}\boldsymbol{q}$ przekształcenia $\boldsymbol{q}$ i wektor normalny $\boldsymbol{n}$ płata $\boldsymbol{p}$. Po obliczeniu macierzy $\mathrm{D}\hat{\boldsymbol{p}}$ wystarczy obliczyć iloczyn wektorowy jej kolumn i po unormowaniu tego iloczynu mamy wektor $\hat{\boldsymbol{n}}$.

Przyjmiemy, że przekształcenie $\boldsymbol{q}$ jest afiniczne; niech $\boldsymbol{s} = \boldsymbol{q}(\boldsymbol{u}) = L\boldsymbol{u} + \boldsymbol{t}$. Macierzą różniczki takiego przekształcenia jest macierz L. Niech $d(\boldsymbol{s}) = \tilde{d}\big(\boldsymbol{g}(\boldsymbol{s})\big)$, gdzie $\tilde{d}$ jest funkcją określoną w prostokącie $B = [0, a) \times [0, b)$, a funkcja $\boldsymbol{g}$ sprowadza punkt $\boldsymbol{s} = (s, t) \in \mathbb{R}^2$ do prostokąta B, co polega na obliczeniu wektora $\boldsymbol{s}' = (s', t') = (s + ka, t + lb)$ z odpowiednio dobranymi liczbami *całkowitymi* k, l. Prostokąt B jest „podstawową komórką" tekstury; jego obraz w przekształceniu $\boldsymbol{q}^{-1}$ jest równoległobokiem, którego kopie tworzą parkietaż (czyli podział na jednakowe kafelki) całej płaszczyzny zawierającej dziedzinę A rysowanego płata. W rezultacie złożenie funkcji $d \circ \boldsymbol{q} = \tilde{d} \circ \boldsymbol{g} \circ \boldsymbol{q}$ jest funkcją podwójnie okresową. Wypada zadbać o to, aby była to funkcja ciągła; w tym celu trzeba wybrać taką funkcję $\tilde{d}$, której obcięcia do przeciwległych boków prostokąta B są identyczne.



Rysunek 21.1. Konstrukcja przekształcenia współrzędnych tekstury

Przykład konstrukcji przekształcenia $\boldsymbol{q}$ jest pokazany na rysunku 21.1. Kolumny macierzy L muszą być wektorami liniowo niezależnymi, przy czym jeśli dziedzina A płata jest kwadratem jednostkowym, to w pierwszym i drugim wierszu współczynniki powinny być całkowitymi wielokrotnościami długości boków a i b prostokąta B (a wektor $\boldsymbol{t}$ może być dowolny). Wtedy wektory przesunięć odpowiadające bokom kwadratu A też będą okresami funkcji $d \circ \boldsymbol{q}$. Dzięki temu odkształcenia płatów mających wspólny brzeg (jakich w czajniku jest wiele) będą na tym brzegu identyczne i odkształcone płaty będą sklejone w sposób ciągły.

[5] Poza tym do obliczenia różniczki odwzorowania Gaussa, $\mathrm{D}\boldsymbol{n}$, potrzebne są pochodne drugiego rzędu parametryzacji $\boldsymbol{p}$. Nie chcemy się z nimi kłopotać.

Do obliczenia wartości funkcji $\boldsymbol{g}$ najwygodniej jest użyć funkcji `fract`, która znajduje części ułamkowe współrzędnych wektora podanego jej jako parametr. Część ułamkowa liczby x jest równa $x - \lfloor x \rfloor$. Współrzędne wektora $\boldsymbol{s}' = \boldsymbol{g}(\boldsymbol{s})$, które podamy jako argumenty funkcji $\tilde{d}$, są zatem równe

$$s' = a(s'' - \lfloor s'' \rfloor), \quad t' = b(t'' - \lfloor t'' \rfloor),$$

gdzie $s'' = s/a$, $t'' = t/b$.

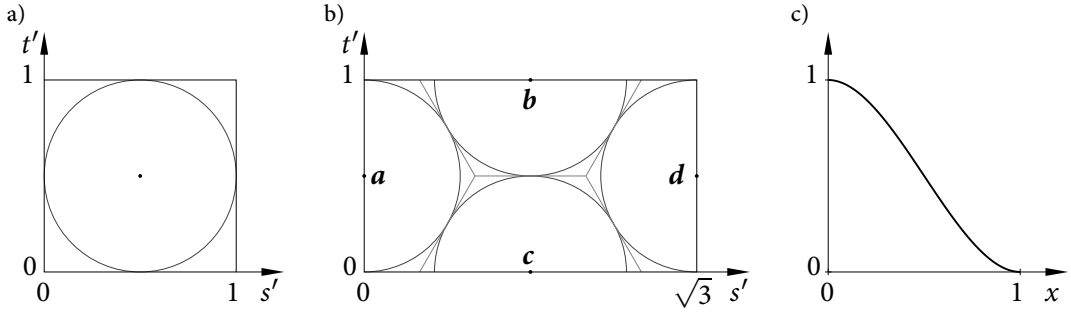
Pozostało wybrać konkretną funkcję $\tilde{d}$ i znaleźć sposób obliczania jej pochodnych cząstkowych. Na początek eksperymentów proponuję dwie funkcje; potem Czytelnik może im wymyślać swoje zamienniki. Moje propozycje są dane za pomocą wzorów

$$r(s', t') = \sqrt{(s' - s_0)^2 + (t' - t_0)^2},$$
$$h(x) = 1 - 3x^2 + 2x^3,$$
$$\tilde{d}(s', t') = \begin{cases} ch\big(2r(s', t')\big) & \text{dla } r(s', t') < 0.5, \\ 0 & \text{w przeciwnym razie.} \end{cases}$$

Liczby s_0, t_0 są współrzędnymi środka koła o promieniu 0.5, w którym funkcja $\tilde{d}$ przyjmuje wartości między zerem a ustaloną liczbą c. Gradient funkcji $\tilde{d}$ w punkcie (s', t') możemy obliczyć na podstawie wzorów

$$\mathrm{D}\tilde{d} = \begin{cases} 2c\mathrm{D}h\big(2r\big)\mathrm{D}r = \\ \quad 2c\big(-6(2r) + 6(2r)^2\big)\left[\dfrac{s' - s_0}{r}, \dfrac{t' - t_0}{r}\right] & \text{dla } r < 0.5, \\ [0, 0] & \text{w przeciwnym razie.} \end{cases}$$

Pierwsza z proponowanych funkcji, $\tilde{d}_1$, jest określona w kwadracie $B = [0, 1)^2$, którego punkt $(s_0, t_0) = (0.5, 0.5)$ jest środkiem (rys. 21.2a).



Rysunek 21.2. Dziedziny funkcji $\tilde{d}_1$ i $\tilde{d}_2$ i wykres funkcji h

Dla drugiej funkcji, $\tilde{d}_2$, przyjąłem, że obszar B jest prostokątem $[0, \sqrt{3}) \times [0, 1)$. Można w niego wpisać cztery półkola o promieniu 0.5, których środkami są punkty $\boldsymbol{a} = (0, 0.5)$, $\boldsymbol{b} = (0.5\sqrt{3}, 1)$, $\boldsymbol{c} = (0.5\sqrt{3}, 0)$ i $\boldsymbol{d} = (\sqrt{3}, 0.5)$. Mając punkt $(s', t') \in B$, należy wyznaczyć

najbliższy mu punkt spośród tych czterech (oznaczmy go (s_0, t_0)) i obliczyć odległość r tych punktów. Aby znaleźć środek półkola najbliższy danego punktu, wystarczy zbadać znaki następujących wyrażeń, które przyjmują wartość 0 na prostych, na których leżą zaznaczone na rysunku 21.2b odcinki oddzielające poszczególne półkola:

$$t' - 0.5, \quad \sqrt{3}s' + t' - 1.5, \quad \sqrt{3}s' - t' - 1.5, \quad \sqrt{3}s' - t' - 0.5, \quad \sqrt{3}s' + t' - 2.5.$$

Rozszerzenie okresowe funkcji $\tilde{d}_2$ na całą płaszczyznę daje parkietaż z sześciokątów foremnych z wpisanymi kołami o średnicy 1, w których ta funkcja ma wartości niezerowe. Wybór funkcji h (rys 21.2c) zapewnia, że okresowe rozszerzenia obu funkcji, $\tilde{d}_1$ i $\tilde{d}_2$, mają ciągłe pochodne pierwszego rzędu.

21.2. Szadery

Szadery wierzchołków i sterowania rozdrabnianiem w programie używanym do rysowania oświetlonych obiektów nie wymagają żadnych zmian w porównaniu z poprzednim wariantem aplikacji. Szader rozdrabniania musi dodatkowo przekazać na swoje wyjście współrzędne punktu w dziedzinie płata i pochodne cząstkowe parametryzacji $\boldsymbol{p}$. Główna zmiana szadera geometrii wiąże się ze zmienionym interfejsem szaderów rozdrabniania i fragmentów — trzeba przekazać z wejścia na wyjście wektory pochodnych cząstkowych. Jeśli to szader geometrii ma wyprodukować wektor normalny, to musi też dokonać ortogonalizacji pochodnych cząstkowych do tego wektora[6]. Do szadera fragmentów zostały dopisane procedury realizujące teksturę odkształceń.

Listing 21.1. Zmiany szadera rozdrabniania — wyjście

GLSL

```
layout(quads,equal_spacing,ccw) in;

out GVertex {
    int  instance;
    vec4 Colour;
    vec3 Position;
    vec3 pu, pv, Normal;
    vec2 PatchCoord, TexCoord;
  } Out;
```

Listing 21.1 przedstawia zmiany bloku wyjściowego szadera rozdrabniania; dodane pole `instance` służy do przekazania dalej numeru instancji, tj. numeru płata Béziera, aby na jego podstawie można było wybrać różne tekstury odkształceń dla poszczególnych płatów rysowanych jednocześnie. To pole zostało dodane „na zaś”, aby ułatwić późniejsze przeróbki aplikacji. Ponadto na wyjście przekazywane są wektory pochodnych cząstkowych płata,

[6] Czyli dokonać rzutowania wektorów pochodnych cząstkowych na podprzestrzeń prostopadłą do wektora normalnego — zobacz podrozdział 5.5.

w polach pu i pv, oraz punkt w dziedzinie płata (skopiowany ze zmiennej wejściowej `gl_TessCoord`) w polu `PatchCoord`.

Procedury `BPHorner2f`, `BPHorner3f` i `BPHorner4f`, których zadaniem jest obliczenie punktu i wektora normalnego płata Béziera odpowiednio wielomianowego płaskiego, wielomianowego trójwymiarowego i wymiernego trójwymiarowego reprezentowanego przez czterowymiarowy płat jednorodny, zostały zmienione tak, aby dodatkowo obliczać wektory pochodnych cząstkowych; w związku z tym mają one dwa nowe parametry wyjściowe, pu i pv. Zmienione fragmenty procedur `BPHorner3f` i `BPHorner4f` są podane na listingu 21.2. Wektory te są reprezentowane w postaci jednorodnej, tj. z dołączoną współrzędną wagową 0. W liniach 8 i 9 obliczone wcześniej sumy są mnożone przez liczby n i m, tj. stopnie płata (zobacz wzory (15.7) i (15.8)). W przypadku płata wymiernego wektory pochodnych parametryzacji wymiernej obliczamy na podstawie wzorów otrzymanych z następującego rachunku (parametry płata dla skrótu pomijamy):

$$\boldsymbol{p} = \frac{\boldsymbol{r}}{w} \quad \Rightarrow \quad \boldsymbol{r} = \boldsymbol{p}w,$$

$$\text{stąd} \qquad \boldsymbol{r}_u = \boldsymbol{p}_u w + \boldsymbol{p} w_u \quad \Rightarrow \quad \boldsymbol{p}_u = \frac{1}{w}(\boldsymbol{r}_u - \boldsymbol{p} w_u)$$

$$\text{oraz} \qquad \boldsymbol{r}_v = \boldsymbol{p}_v w + \boldsymbol{p} w_v \quad \Rightarrow \quad \boldsymbol{p}_v = \frac{1}{w}(\boldsymbol{r}_v - \boldsymbol{p} w_v),$$

przy czym $\boldsymbol{r}$ oznacza składający się z pierwszych trzech współrzędnych płata jednorodnego $\boldsymbol{P}$ licznik ułamka definiującego płat wymierny $\boldsymbol{p}$, a funkcja w, czyli mianownik, opisuje czwartą współrzędną (wagową) płata $\boldsymbol{P}$. Obliczenie na podstawie powyższych wzorów jest realizowane przez instrukcje w liniach 19–24.

Listing 21.2. Zmiany szadera rozdrabniania — wyznaczanie punktów płata

GLSL

```
1: void BPHorner3f ( float u, float v,
2:                   out vec4 pos, out vec3 pu, out vec3 pv, out vec3 nv )
3: {
4:   .... /*początek bez zmian */
5:   BCHorner3f ( bezp.udeg, q0, u, r, ru );
6:   BCHorner3f ( bezp.udeg, q1, u, rv, ruv );
7:   pos = vec4 ( r, 1.0 );
8:   pu = bezp.udeg * ru;
9:   pv = bezp.vdeg * rv;
10:  nv = cross ( ru, rv );
11: } /*BPHorner3f*/
12: ....
13: void BPHorner4f ( float u, float v,
14:                   out vec4 pos, out vec3 pu, out vec3 pv, out vec3 nv )
15: {
16:   .... /*początek bez zmian */
17:   BCHorner4f ( bezp.udeg, q0, u, pos, ru );
18:   BCHorner4f ( bezp.udeg, q1, u, rv, ruv );
```

```
19:   nv = cross4 ( pos, rv, ru );
20:   w = pos.w;
21:   pos /= w;
22:   pu = bezp.udeg*(ru.xyz-pos.xyz*ru.w)/w;
23:   pv = bezp.vdeg*(rv.xyz-pos.xyz*rv.w)/w;
24: } /*BPHorner4f*/
```

Nie zamieściłem w książce zmian procedury `BPHorner2f`, aby dać Czytelnikowi okazję do ćwiczenia polegającego na samodzielnym jej zmodyfikowaniu, co wymaga dokonania zmian także w procedurze `BCHorner2f`.

Zmiany w procedurze `main` szadera rozdrabniania są pokazane na listingu 21.3. W linii 7 jest przekazywany na wyjście numer instancji (czyli płata Béziera). W liniach 21 i 22 są przekazywane na wyjście wektory pochodnych cząstkowych płata przekształcone do układu współrzędnych świata (w którym szader podaje także punkt płata, w polu `Position`). Wyrażenie `mat3(trb.mm)` „wycina" z macierzy 4×4 jej lewy górny blok 3×3, który reprezentuje część liniową przejścia od układu współrzędnych modelu do układu świata. Wektory pochodnych parametryzacji są wektorami swobodnymi w przestrzeni, w której znajdują się nasze obiekty, dlatego mnożymy je przez tę macierz, a nie przez jej transpozycję odwrotności.

Listing 21.3. Zmiany szadera rozdrabniania — procedura `main`

GLSL

```
1: void main ( void )
2: {
3:   vec4 pos;
4:   vec3 pu, pv, nv;
5:   int  i;
6:
7:   Out.instance = inst = In[0].instance;
8:   pos = vec4 ( 0.0 );
9:   nv = pu = pv = vec3 ( 0.0 );
10:  switch ( bezp.dim ) {
11: case 2:
12:     BPHorner2f ( gl_TessCoord.x, gl_TessCoord.y, pos, pu, pv, nv );  break;
13: case 3:
14:     BPHorner3f ( gl_TessCoord.x, gl_TessCoord.y, pos, pu, pv, nv );  break;
15: case 4:
16:     BPHorner4f ( gl_TessCoord.x, gl_TessCoord.y, pos, pu, pv, nv );  break;
17:  }
18:  Out.PatchCoord = gl_TessCoord.xy;
19:  gl_Position = trb.vpm * (trb.mm * pos);
20:  Out.Position = (trb.mm * pos).xyz;
21:  Out.pu = mat3(trb.mm) * pu;
22:  Out.pv = mat3(trb.mm) * pv;
23:  if ( !BezNormals || dot ( nv, nv ) < 1.0e-10 )
24:  .... /* dalej bez zmian */
25: } /*main*/
```

Na listingu 21.4 jest pokazany szader geometrii. W liniach 6–12 i 14–20 są pokazane zmienione bloki interfejsu, wejściowy (który ma identyczne pola jak blok wyjściowy na listingu 21.1) i wyjściowy. Budowa tego bloku jest taka sama jak bloku wejściowego, ale pole typu `int`, którego wartości, choć jednakowe dla wszystkich wierzchołków trójkąta, nie mogą być interpolowane, jest opatrzone kwalifikatorem `flat` (zobacz p. 12.4.5). Ten sam kwalifikator trzeba podać w opisie bloku wejściowego szadera fragmentów.

Listing 21.4. Szader geometrii

GLSL

```
1: #version 450
2:
3: layout(triangles) in;
4: layout(triangle_strip,max_vertices=3) out;
5:
6: in GVertex {
7:     int  instance;
8:     vec4 Colour;
9:     vec3 Position;
10:    vec3 pu, pv, Normal;
11:    vec2 PatchCoord, TexCoord;
12:  } In[];
13:
14: out FVertex {
15:    flat int instance;
16:    vec4      Colour;
17:    vec3      Position;
18:    vec3      pu, pv, Normal, TNormal;
19:    vec2      PatchCoord, TexCoord;
20:  } Out;
21:
22: uniform bool BezNormals;
23:
24: void main ( void )
25: {
26:   int i;
27:   vec3 v1, v2, nv;
28:
29:   v1 = In[1].Position - In[0].Position;
30:   v2 = In[2].Position - In[0].Position;
31:   nv = normalize ( cross ( v2, v1 ) );
32:   for ( i = 0; i < 3; i++ ) {
33:     gl_Position = gl_in[i].gl_Position;
34:     Out.Position = In[i].Position;
35:     if ( !BezNormals || dot ( In[i].Normal, In[i].Normal ) < 1.0e-10 ) {
36:       Out.pu = In[i].pu - dot ( In[i].pu, nv )*nv;
37:       Out.pv = In[i].pv - dot ( In[i].pv, nv )*nv;
38:       Out.Normal = nv;
```

```
39:     }
40:     else {
41:       Out.pu = In[i].pu;
42:       Out.pv = In[i].pv;
43:       Out.Normal = In[i].Normal;
44:     }
45:     Out.TNormal = nv;
46:     Out.Colour = In[i].Colour;
47:     Out.PatchCoord = In[i].PatchCoord;
48:     Out.TexCoord = In[i].TexCoord;
49:     Out.instance = In[i].instance;
50:     EmitVertex ();
51:   }
52:   EndPrimitive ();
53: } /*main*/
```

Szader geometrii przekazuje na wyjście współrzędne punktu w dziedzinie płata oraz pochodne cząstkowe i wektor normalny, chyba że (z powodu osobliwości) wektor normalny jest wektorem zerowym lub zmiennej `BezNormals` została nadana wartość `false` — w tym przypadku szader geometrii ma za zadanie obliczyć i przekazać na wyjście wektor normalny płaszczyzny przetwarzanego trójkąta. Ale wtedy szader musi dokonać rzutowania wektorów pochodnych cząstkowych na dwuwymiarową podprzestrzeń równoległą do tej płaszczyzny (czyli przeprowadzić ich ortogonalizację do wektora $\boldsymbol{n}$), co jest robione w liniach 36–37 na podstawie wzoru (5.9).

Listing 21.5 przedstawia zmieniony blok wejściowy i zmienne globalne oraz procedurę `main` szadera fragmentów. Blok wejściowy jest dopasowany do bloku wyjściowego szadera geometrii. Mamy też nową zmienną jednolitą `NormalSource`, która służy do wybierania tekstury odkształceń, o czym będzie mowa dalej, i zmienną `ModifyDepth`, która umożliwia włączanie i wyłączanie dodawania poprawki głębokości fragmentu. Procedura `main` w linii 21 przypisuje zmiennej globalnej `pnormal` podany na wejściu szadera wektor normalny $\boldsymbol{n}$ oryginalnej powierzchni (dokonując normalizacji, potrzebnej po etapie rasteryzacji), po czym w linii 22 wywołuje procedurę odpowiedzialną za podanie wektora normalnego powierzchni odkształconej i poprawki głębokości. Poprawka jest dodawana do głębokości fragmentu, jeśli zmienna jednolita `ModifyDepth` ma wartość `true`. Po przypisaniu (w linii 24) zmiennej `mm` własności materiału (które mogą być określone na podstawie tekstury) jest wywoływana procedura, która realizuje obliczenie koloru fragmentu przy użyciu wybranego modelu oświetlenia.

Listing 21.5. Zmiany szadera fragmentów — wejście i procedura `main`

GLSL

```
1: in FVertex {
2:     flat int instance;
3:     vec4     Colour;
4:     vec3     Position;
5:     vec3     pu, pv, Normal, TNormal;
```

```
6:     vec2     PatchCoord, TexCoord;
7:   } In;
8:
9: uniform int ColourSource, NormalSource;
10: uniform bool ModifyDepth = false;
11:
12: Material mm;
13: vec3     normal, tnormal, pnormal;
14:
15: void main ( void )
16: {
17:   vec3  Colour;
18:   float dz;
19:
20:   tnormal = In.TNormal;
21:   pnormal = normalize ( In.Normal );
22:   normal = GetNormalVector ( In.PatchCoord, dz );
23:   gl_FragDepth = ModifyDepth ? gl_FragCoord.z + dz : gl_FragCoord.z;
24:   GetMaterial ( In.TexCoord );
25:   switch ( LightingModel ) {
26: default: Colour = LambertLighting ();     break;
27:  case 1: Colour = BlinnPhongLighting ();  break;
28:   }
29:   out_Colour = vec4 ( AGamma ( Colour ), 1.0 );
30: } /*main*/
```

Na listingu 21.6 są pokazane procedury realizujące tekstury odkształceń, tj. obliczające wektor normalny $\hat{\boldsymbol{n}}$ w odpowiednim punkcie. Rozpatrywane w podrozdziale 21.1 przekształcenia afiniczne $\boldsymbol{q}$ zostały tu zakodowane „na twardo” — wartościami zmiennych `dtxm1` i `dtxm2` są macierze 2×3; dwie pierwsze kolumny każdej z nich tworzą blok L, a trzecia kolumna jest wektorem $\boldsymbol{t}$:

$$L_1 = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}, \quad L_2 = \begin{bmatrix} 6\sqrt{3} & 0 \\ 0 & 10 \end{bmatrix}, \quad \boldsymbol{t}_1 = \boldsymbol{t}_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Ponadto w linii 2 mamy „na twardo” zakodowaną stałą $c = -0.003$, która określa wysokość „nierówności” na powierzchni.

Aby poprawić czytelność kodu szadera, każda z opisanych tu tekstur proceduralnych jest realizowana przez trzy podprogramy, z których pierwszy, `NTex12` oblicza wartość funkcji $d = \tilde{d} \circ \boldsymbol{g}$ i jej pochodne cząstkowe (zobacz s. 515–516). Te trzy liczby są przekazywane w polach `x`, `y` i `z` wektora podanego w instrukcji `return`.

Pierwszym zadaniem drugiego podprogramu, `NormalTex12`, jest obliczenie pochodnych funkcji złożonej $d \circ \boldsymbol{q}$. Wartościami parametrów `dx` i `dy` są liczby $s' - s_0$ i $t' - t_0$ (zobacz s. 516). Parametr `L` to macierz L opisująca część liniową przekształcenia $\boldsymbol{q}$. Parametr wyjściowy `d` służy do przekazania wartości funkcji d. W linii 27 następuje obliczenie gradientu

Listing 21.6. Procedury realizujące tekstury odkształceń

GLSL

```
#define SQRT3 1.7320508
#define C     (-0.005)

const mat3x2 dtxm1 = mat3x2 ( 10.0, 0.0,  0.0, 10.0,  0.0, 0.0 );
const mat3x2 dtxm2 = mat3x2 ( 6.0*SQRT3, 0.0,  0.0, 10.0,  0.0, 0.0 );

vec3 NTex12 ( float dx, float dy )
{
  float r;

  r = 2.0*sqrt ( dx*dx + dy*dy );
  if ( r >= 1.0 )
    return vec3 ( 0.0, 0.0, 0.0 );
  else if ( r > 0.0 )
    return C*vec3 ( (r+r-3.0)*r*r+1.0, (24.0*r-24.0)*vec2 ( dx, dy ) );
  else
    return vec3 ( C, 0.0, 0.0 );
} /*NTex12*/

vec3 NormalTex12 ( float dx, float dy, mat2 L, out float d )
{
  vec2 Ddq;
  vec3 dd, dpu, dpv;

  dd = NTex12 ( dx, dy );
  d = dd.x;
  Ddq = L * d.yz;
  dpu = In.pu + Ddq.x*pnormal;
  dpv = In.pv + Ddq.y*pnormal;
  return normalize ( cross ( dpu, dpv ) );
} /*NormalTex12*/

vec3 NormalTexture1 ( vec2 pc, out float d )
{
  pc = fract ( dtxm1*vec3 ( pc, 1.0 ) );
  return NormalTex12 ( pc.x-0.5, pc.y-0.5, mat2(dtxm1), d );
} /*NormalTexture1*/

vec3 NormalTexture2 ( vec2 pc, out float d )
{
  int   c;
  float dx, dy;

  pc = dtxm2*vec3 ( pc, 1.0 );
  pc = fract ( vec2 ( pc.x/SQRT3, pc.y ) );
```

```
46:   switch ( (pc.x > 0.5 ? 1 : 0) + (pc.y > 0.5 ? 2 : 0) ) {
47: case 0: c = 3.0*pc.x-pc.y > 0.5 ? 1 : 0;  break;
48: case 1: c = 3.0*pc.x+pc.y > 2.5 ? 3 : 1;  break;
49: case 2: c = 3.0*pc.x+pc.y > 1.5 ? 2 : 0;  break;
50: case 3: c = 3.0*pc.x-pc.y > 1.5 ? 3 : 2;  break;
51:   }
52:   switch ( c ) {
53: case 0: dx = pc.x;      dy = pc.y-0.5;  break;  /* a */
54: case 1: dx = pc.x-0.5;  dy = pc.y;      break;  /* c */
55: case 2: dx = pc.x-0.5;  dy = pc.y-1.0;  break;  /* b */
56: case 3: dx = pc.x-1.0;  dy = pc.y-0.5;  break;  /* d */
57:   }
58:   return NormalTex12 ( dx*SQRT3, dy, mat2(dtxm2), d );
59: } /*NormalTexture2*/
60:
61: #define M3ROW(M,I) vec3 ( M[0][I], M[1][I], M[2][I] )
62:
63: vec3 GetNormalVector ( vec2 pc, out float dz )
64: {
65:   vec3 nv;
66:
67:   switch ( NormalSource ) {
68: default: dz = 0.0;  return pnormal;
69:  case 1: nv = NormalTexture1 ( pc, dz );  break;
70:  case 2: nv = NormalTexture2 ( pc, dz );  break;
71:   }
72:   dz *= 0.5*dot ( M3ROW ( trb.vpm, 2 ), pnormal ) * gl_FragCoord.w;
73:   return nv;
74: } /*GetNormalVector*/
```

funkcji $d \circ \boldsymbol{q}$. W liniach 28 i 29 do wektorów pochodnych cząstkowych płata $\boldsymbol{p}$ są dodawane zaburzenia, a w linii 30 obliczone w ten sposób pochodne płata $\hat{\boldsymbol{p}}$ są mnożone; ich unormowany iloczyn wektorowy jest jednostkowym wektorem normalnym płata $\hat{\boldsymbol{p}}$.

Każda z dwóch opisanych tu tekstur ma inny trzeci podprogram. Parametr `pc` tych procedur przekazuje współrzędne (u, v) w dziedzinie płata. Procedura `NormalTexture1` oblicza w linii 35 parę (s', t') współrzędnych punktu w kwadracie jednostkowym, tj. przedstawionym na rysunku 21.2a obszarze B. Jest to wartość funkcji $\boldsymbol{g}$ (zobacz s. 515) złożonej z przekształceniem afinicznym $\boldsymbol{q}$, zakodowanym jako wartość zmiennej `dtxm1`. Przekształcenie to odwzorowuje kwadrat $[0,1]^2$ — dziedzinę płata Béziera — na kwadrat $[0,10]^2$, a liczby s' i t' są częściami ułamkowymi współrzędnych punktu w tym kwadracie. Trzeci parametr procedury `NormalTex12` jest macierzą części liniowej przekształcenia $\boldsymbol{q}$.

Procedura `NormalTexture2`, realizująca drugą teksturę, ma bardziej skomplikowane zadanie, bo podstawowa komórka tekstury (prostokąt B, rys. 21.2b) nie jest kwadratem. Przekształcenie reprezentowane przez wartość zmiennej `dtxm2` przekształca dziedzinę płata Béziera na prostokąt $[0, 6\sqrt{3}] \times [0, 10]$; mieści się w nim dokładnie 60 podstawowych komórek. Funkcja `fract` w linii 45 oblicza parę liczb $(s'/\sqrt{3}, t')$. Instrukcje w liniach 46–57

wyznaczają najbliższy punktu (s', t') środek (s_0, t_0) półkola wpisanego w prostokąt B i obliczają parę liczb $\big((s' - s_0)/\sqrt{3}, t' - t_0\big)$. W linii 58 jest wywoływana procedura `NormalTex12` z parametrami $s' - s_0$ i $t' - t_0$, po czym następuje przekazanie wyników jej obliczeń.

Wektor normalny płata $\tilde{\boldsymbol{p}}$ i wielkość przesunięcia obliczone przez wybrany (za pomocą zmiennej jednolitej `NormalSource`) podprogram otrzymuje procedura `GetNormalVector`. Jej ostatnim zadaniem jest obliczenie poprawki głębokości fragmentu, czyli przyrostu współrzędnej z fragmentu na podstawie wielkości przesunięcia (w układzie świata) punktu $\boldsymbol{p}$ w kierunku jednostkowego wektora normalnego $\boldsymbol{n}$ na odległość d. Mamy zatem wektor $\boldsymbol{n}$, przypisany zmiennej `nv`, oraz liczbę d, zapamiętaną w zmiennej `dz`. Należy ją pomnożyć przez iloraz współrzędnej z wektora $\boldsymbol{n}$ przekształconego do układu kostki standardowej i współrzędnej wagowej W punktu $\boldsymbol{p}$ w tym układzie. Dodatkowy czynnik 0.5 jest związany przekształceniem współrzędnej z w układzie kostki standardowej (w którym $z \in [-1, 1]$) do układu, w którym głębokość jest liczbą z przedziału $[0, 1]$. Szader fragmentów w zmiennej wbudowanej `gl_FragCoord.w` otrzymuje liczbę $1/W$. Aby obliczyć współrzędną z wektora $\boldsymbol{n}$ w układzie kostki standardowej, można pomnożyć ten wektor przez macierz przechowywaną w zmiennej `trb.vpm`, ale wtedy byłyby obliczane również niepotrzebne pozostałe współrzędne. Do obliczenia tylko współrzędnej z służy makrodefinicja `M3ROW` (linia 61), która tworzy wektor z pierwszych trzech współczynników wskazanego wiersza macierzy. Wystarczy obliczyć iloczyn skalarny tego wektora i wektora $\boldsymbol{n}$ (linia 72).

21.3. Zmiany w aplikacji

W kodzie aplikacji napisanym w C są potrzebne bardzo niewielkie zmiany. Procedura `LoadBPShaders` w dodatku do dotąd używanych zmiennych jednolitych musi (przy użyciu procedury `glGetUniformLocation`) uzyskać dostęp do zmiennych `NormalSource` i `ModifyDepth`. Do zapamiętania ich położeń służą nowe pola `NormalSourceLoc` i `ModifyDepthLoc` struktury `BPRenderPrograms`, a ich wartości są pamiętane w polach `normal_source` i `modify_depth` (wszystkie te pola są typu `GLint`). Przed rysowaniem płatów Béziera zmiennej `NormalSource` będzie nadawana wartość `0`, `1` lub `2`. Zmiennej `ModifyDepth` będą nadawane wartości `false` albo `true` (reprezentowane przez liczby `0` i `1`), co umożliwi obejrzenie skutków modyfikowania głębokości fragmentów. W obu przypadkach użyjemy do tego procedury `glUniform1i`. Licząc na wyobraźnię, zaradność i niemałą już wiedzę Czytelników, pomijam listingi ze zmodyfikowaną struktrą `BPRenderPrograms` i instrukcjami odczytywania położeń zmiennych jednolitych oraz nadawania im wartości.

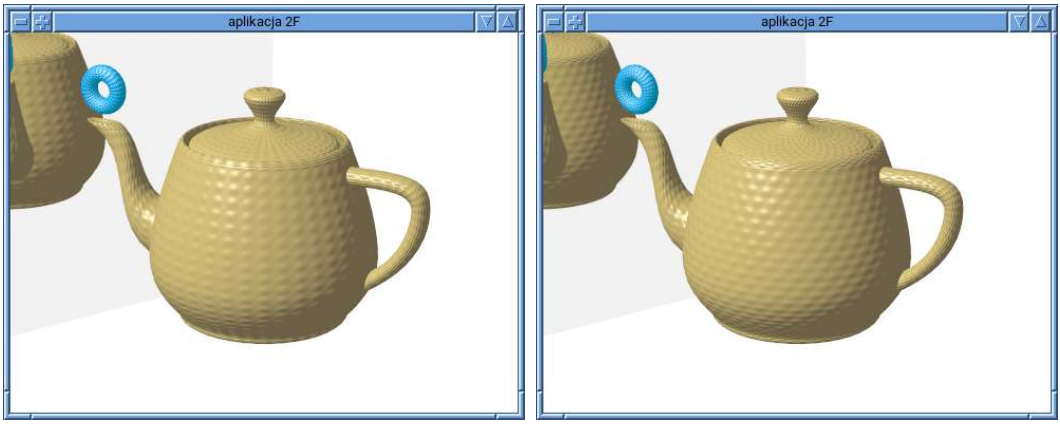
Polecenia przełączenia tekstury odkształceń oraz włączania i wyłączania modyfikacji głębokości są wydawane przez naciskanie klawiszy z literami `D` i `M`. Aby to działało, do procedury `ProcessCharCommand` trzeba dopisać reakcje na te polecenia, pokazane na listingu 21.7.

Listing 21.7. Dodatkowe instrukcje w procedurze `ProcessCharCommand`

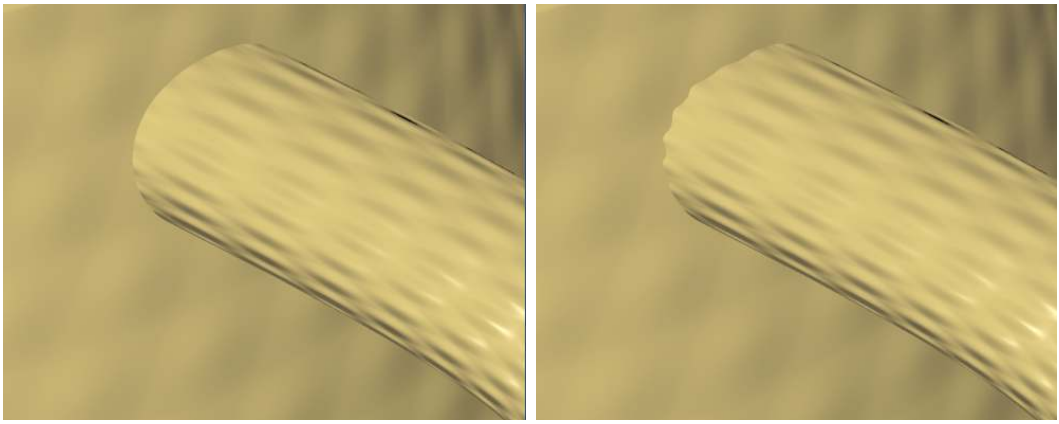
C

```
1: char ProcessCharCommand ( char charcode )
2: {
3:   switch ( toupper ( charcode ) ) {
```

```
4: ....
5: case 'D':
6:     appdata.brprog.normal_source = (appdata.brprog.normal_source+1) % 3;
7:     return true;
8: case 'M':
9:     appdata.brprog.modify_depth = !appdata.brprog.modify_depth;
10:    return true;
11: ....
12:   }
13:   return false;
14: } /*ProcessCharCommand*/
```



Rysunek 21.3. Okno aplikacji drugiej F



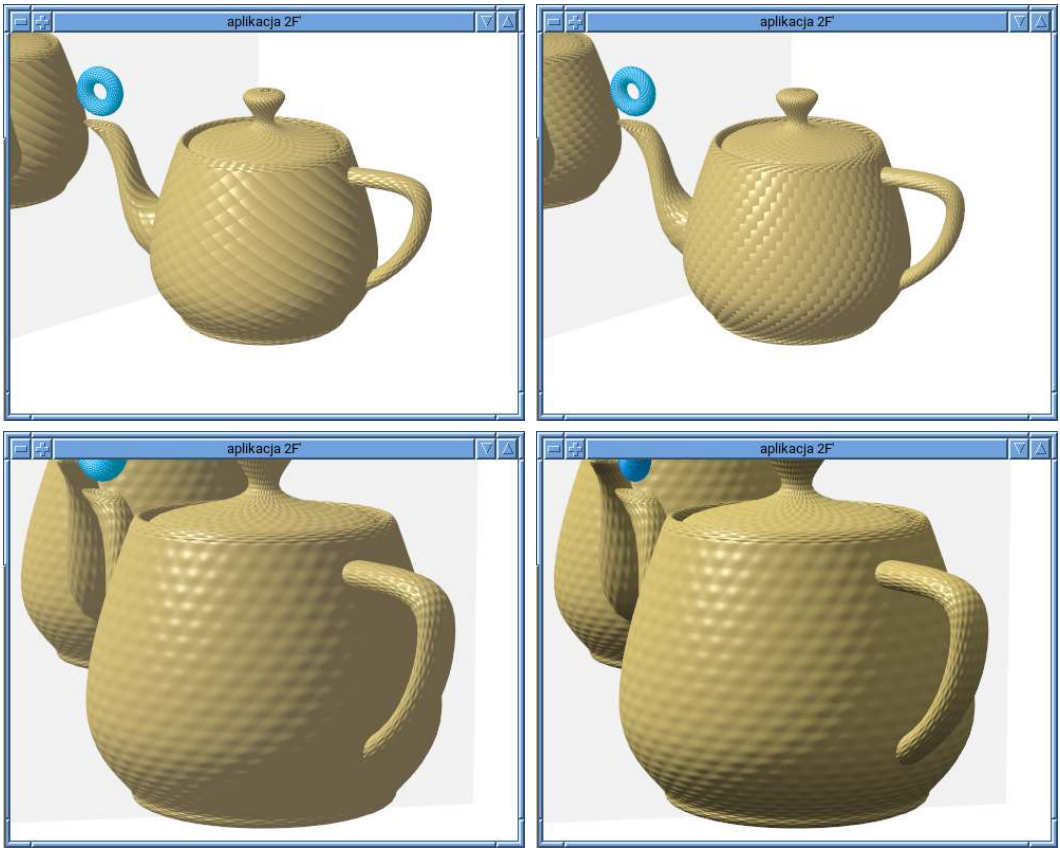
Rysunek 21.4. Skutek modyfikowania głębokości fragmentów

Rysunki 21.3 i 21.4 przedstawiają czajnik z teksturami odkształceń opisanymi w tym rozdziale. Na obrazie po prawej stronie rysunku 21.4 możemy zobaczyć skutek modyfikowania

głębokości fragmentów, porównując go z obrazem z lewej strony wykonanym bez tej modyfikacji.

21.4. Ćwiczenia

1. Wykonaj eksperymenty polegające na zmienianiu macierzy L w kodzie szadera fragmentów. Zmieniaj też stałą c i oglądaj skutki tych zmian na obrazach.
2. Przerób zmienną szadera przechowującą macierz L o nadanej stałej wartości na zmienną jednolitą, aby umożliwić zmienianie wartości współczynników tej macierzy podczas działania aplikacji. Oprogramuj to i poeksperymentuj.
3. Zmień (według własnego uznania) funkcję h użytą do zdefiniowania funkcji $\tilde{d}_1$ i $\tilde{d}_2$; tak zmodyfikuj procedurę `NTex12`, aby realizowała obliczenie tekstury odkształceń z nową funkcją.



Rysunek 21.5. Rozwiązania ćwiczeń 3 i 4

4. Narysuj obiekty z nałożoną teksturą odkształceń przy użyciu oświetlenia hemisferycznego (p. 18.4.3) i porównaj otrzymane obrazy z obrazami obiektów oświetlonych jednakowo ze wszystkich stron.
5. Dokonaj takich modyfikacji szaderów i aplikacji, aby dla każdego płata było możliwe nałożenie innej tekstury odkształceń (w tym indywidualne określanie przekształcenia $\boldsymbol{q}$ i stałej c). Wykorzystaj do tego pole `instance` struktury `FVertex` z listingu 21.5.
6. *Zmodyfikuj szadery tak, aby nakładać teksturę odkształceń określoną w układzie współrzędnych modelu. Dzięki temu wielkość odkształceń będzie się zmieniać razem z obiektem, gdy ten zostanie poddany innemu skalowaniu.
7. Zastanów się, jak użyć tablicy tekseli do reprezentowania tekstury odkształceń i nie poprzestań na zastanawianiu się.

21.5. Uzupełnienia

21.5.1. Antyaliasing tekstur proceduralnych

Jeśli jest włączone wielokrotne próbkowanie, to OpenGL działa tak, że wybrane punkty w obszarze piksela są albo nie są zaliczane do obrazu rasteryzowanego punktu, odcinka lub trójkąta. Po ustaleniu, że co najmniej jeden z tych punktów należy do obrazu tej figury, domyślnie szader fragmentów jest wywoływany tylko dla jednego z nich, po czym ten sam kolor jest przypisywany wszystkim punktom należącym do obrazu tej figury. Ma to na celu oszczędność czasu obliczeń. Wyniki są zadowalające dla pikseli na brzegach figur i (dzięki skutecznej interpolacji tekseli) także w obszarach pokrytych „zwykłymi" teksturami. Znacznie gorzej wyglądają skutki użycia tekstur proceduralnych, zwłaszcza w tych obszarach, w których tekstura „szybko oscyluje"[7].

Można spowodować wykonywanie szadera fragmentów dla większej liczby punktów, co poprawia jakość obrazu (kosztem spowolnienia obliczeń). Nazywa się to **cieniowaniem próbek** (ang. *sample shading*). W tym celu przed rysowaniem trzeba wykonać instrukcje

```
glEnable ( GL_SAMPLE_SHADING );
glMinSampleShading ( 1.0 );
```

Pierwsza instrukcja włącza cieniowanie próbek, a druga podaje informację, dla jakiej części punktów należących do obrazu danej figury ma być wywoływany szader fragmentów; parametr `1.0` oznacza, że dla wszystkich. Kolory poszczególnych fragmentów są następnie uśredniane. Wyniki zastosowania opcji domyślnej i cieniowania próbek (przy 12 punktach na piksel) można porównać na rysunku 21.6. Choć efekt cieniowania próbek jest lepszy, nie jest idealny, a to dlatego, że na środku pokrywki czajnika szybkość oscylacji nałożonej tekstury jest nieograniczona.

[7] Czyli tam, gdzie obraz tekstury ma składowe o wysokich częstotliwościach.

Rysunek 21.6. Obrazy otrzymane z wyłączonym i włączonym cieniowaniem próbek

Cieniowania próbek warto używać *tylko* podczas rysowania obiektów pokrytych teksturą i *tylko* wtedy, gdy jakość obrazów otrzymanych bez tego jest niezadowalająca. Cieniowanie próbek podczas rysowania obiektów bez tekstury, a także podczas znajdowania reprezentacji obszarów cienia (czym zajmiemy się w następnym rozdziale), jest marnowaniem czasu. Zatem należy też wywoływać, tam gdzie trzeba, procedurę `glDisable`.

21.5.2. *Modyfikowanie współrzędnych tekstury odkształceń

Choć odkształcenia zrealizowane w sposób opisany w tym rozdziale są małe, podczas oglądania z różnych kierunków *powinny* odpowiednio zmieniać kształt na obrazie (to zjawisko jest nazywane **paralaksą**), a tego nie robią. Jeśli ponadto na powierzchnię jest nałożona tekstura „zwykła", opisująca kolory poszczególnych punktów powierzchni, której wektor normalny nie jest równoległy do wektora kierunku do obserwatora, to zwykła tekstura w danym punkcie na obrazie opisuje kolor innego punktu, położonego nieopodal. Najczęściej to jest niezauważalne, no ale czasami chcielibyśmy się zbliżyć do perfekcji.

Mamy zatem oryginalny płat powierzchni parametrycznej $\boldsymbol{p}(u,v)$ i płat $\hat{\boldsymbol{p}}(u,v)$ zdefiniowany w sposób przedstawiony w podrozdziale 21.1. Oba płaty opiszemy w układzie współrzędnych kostki standardowej. Niech C oznacza macierz 4×4 opisującą (w reprezentacji jednorodnej) przejście do tego układu od układu, w którym płaty są określone (układu modelu lub układu świata). Przekształcenie opisane przez tę macierz na ogół nie jest afiniczne, bo jednym z jego etapów jest przekształcenie perspektywiczne (zobacz podrozdz. 6.2, 6.6).

Niech $\boldsymbol{Q}$, $\overline{\boldsymbol{Q}}$ i W oznaczają funkcje, z których pierwsza „wybiera" pierwsze trzy współrzędne wektora w $\mathbb{R}^4$ (a więc jest przekształceniem $\mathbb{R}^4 \to \mathbb{R}^3$), druga wybiera pierwsze dwie współrzędne, a wartością trzeciej jest czwarta współrzędna takiego wektora. Oczywiście, iloraz $\boldsymbol{r}(\boldsymbol{P}) = \boldsymbol{Q}(\boldsymbol{P})/W(\boldsymbol{P})$ opisuje przejście od współrzędnych jednorodnych do kartezjańskich, a $\overline{\boldsymbol{r}}(\boldsymbol{P}) = \overline{\boldsymbol{Q}}(\boldsymbol{P})/W(\boldsymbol{P})$ jest funkcją wektorową składającą się z pierwszych dwóch

współrzędnych funkcji $\boldsymbol{r}$. Niech

$$\boldsymbol{P}(u,v) = \begin{bmatrix} \boldsymbol{p}(u,v) \\ 1 \end{bmatrix}, \quad \boldsymbol{N}(u,v) = \begin{bmatrix} \boldsymbol{n}(u,v) \\ 0 \end{bmatrix}, \quad \hat{\boldsymbol{P}}(u,v) = \begin{bmatrix} \hat{\boldsymbol{p}}(u,v) \\ 1 \end{bmatrix}.$$

Płaty są więc opisane w układzie kostki standardowej przez parametryzacje

$$\boldsymbol{r}\big(C\boldsymbol{P}(u,v)\big) = \boldsymbol{Q}\big(C\boldsymbol{P}(u,v)\big)/W\big(C\boldsymbol{P}(u,v)\big),$$
$$\boldsymbol{r}\big(C\hat{\boldsymbol{P}}(u,v)\big) = \boldsymbol{Q}\big(C\hat{\boldsymbol{P}}(u,v)\big)/W\big(C\hat{\boldsymbol{P}}(u,v)\big),$$

natomiast funkcje $\overline{\boldsymbol{r}}\big(C\boldsymbol{P}(u,v)\big)$ i $\overline{\boldsymbol{r}}\big(C\hat{\boldsymbol{P}}(u,v)\big)$ są parametryzacjami obrazów tych płatów na ścianie kostki standardowej, odwzorowanej dalej na klatkę w oknie.

Ustalmy punkt (u_0, v_0) w dziedzinie i rozważmy odpowiadający mu punkt powierzchni przetwarzany przez szader fragmentów. Obraz punktu $\hat{\boldsymbol{p}}(u_0, v_0)$ na ścianie kostki standardowej ma współrzędne $(x, y) = \overline{\boldsymbol{r}}\big(C\hat{\boldsymbol{P}}(u_0, v_0)\big)$. Chcemy znaleźć taki punkt $(u^\star, v^\star)$ w dziedzinie płata $\boldsymbol{p}$, aby było $(x, y) = \overline{\boldsymbol{r}}\big(C\boldsymbol{P}(u^\star, v^\star)\big)$; innymi słowy, chcemy, aby obraz punktu $\hat{\boldsymbol{p}}(u_0, v_0)$ był identyczny z obrazem punktu $\boldsymbol{p}(u^\star, v^\star)$. Aby ulepszyć obraz płata $\hat{\boldsymbol{p}}$, współrzędne tekstur obliczymy na podstawie punktu $(u^\star, v^\star)$ zamiast (u_0, v_0). Tym samym uznamy*, że przetwarzany fragment odpowiada punktowi $\hat{\boldsymbol{p}}(u^\star, v^\star)$.

Uwaga: Taki punkt $(u^\star, v^\star)$ *może nie istnieć*, ale to nas nie powstrzyma od próby znalezienia go, a ściślej biorąc, próby znalezienia jego przybliżenia lepszego niż (u_0, v_0). W wielu przypadkach to się nam uda.

Mamy do rozwiązania układ dwóch równań nieliniowych

$$\overline{\boldsymbol{r}}\big(C\hat{\boldsymbol{P}}(u_0, v_0)\big) = \overline{\boldsymbol{r}}\big(C\boldsymbol{P}(u^\star, v^\star)\big)$$

z niewiadomymi $u^\star, v^\star$. Możemy założyć*, że $W\big(C\hat{\boldsymbol{P}}(u_0, v_0)\big) \approx W\big(C\boldsymbol{P}(u^\star, v^\star)\big)$, co umożliwia zastąpienie napisanego wyżej układu równań przez $\overline{\boldsymbol{Q}}\big(C\hat{\boldsymbol{P}}(u_0, v_0)\big) = \overline{\boldsymbol{Q}}\big(C\boldsymbol{P}(u^\star, v^\star)\big)$. Mamy zatem znaleźć miejsce zerowe funkcji

$$\boldsymbol{f}(u,v) = \overline{\boldsymbol{Q}}\Big(C\big(\hat{\boldsymbol{P}}(u_0, v_0) - \boldsymbol{P}(u,v)\big)\Big).$$

Użyjemy do tego metody Newtona, ale ograniczymy się* do wykonania tylko jednego kroku tej metody. Po pierwsze, nie mamy gwarancji, że kolejne kroki dałyby nam ciąg punktów zbieżny do $(u^\star, v^\star)$, a po drugie, jak się przekonamy, szader fragmentów, który będzie wykonywać to obliczenie, może na podstawie danych otrzymanych na wejściu wykonać ten jeden krok kosztem znacznie mniejszym od tego, który musiałby ponieść w krokach kolejnych.

Przekształćmy jeszcze wzór opisujący funkcję $\boldsymbol{f}$. Na podstawie definicji płata $\hat{\boldsymbol{p}}$ mamy

$$\boldsymbol{f}(u,v) = \overline{\boldsymbol{Q}}\Big(C\big(\boldsymbol{P}(u_0, v_0) + d\big(\boldsymbol{q}(u_0, v_0)\big)\boldsymbol{N}(u_0, v_0) - \boldsymbol{P}(u,v)\big)\Big).$$

Zgodnie z definicją metody Newtona mamy obliczyć punkt

$$(u_1, v_1) = (u_0, v_0) - \big(\mathrm{D}\boldsymbol{f}(u_0, v_0)\big)^{-1} \boldsymbol{f}(u_0, v_0). \tag{21.1}$$

Kolumny macierzy $\mathrm{D}\boldsymbol{f}(u, v)$ (macierzy różniczki funkcji $\boldsymbol{f}$) są pochodnymi funkcji $\boldsymbol{f}$ ze względu na zmienne u, v; korzystając z tego, że funkcja $\overline{\boldsymbol{Q}}$ jest przekształceniem liniowym, otrzymujemy wzory

$$\boldsymbol{f}_u(u, v) = \overline{\boldsymbol{Q}}\big(-C\boldsymbol{P}_u(u, v)\big), \qquad \boldsymbol{f}_v(u, v) = \overline{\boldsymbol{Q}}\big(-C\boldsymbol{P}_v(u, v)\big).$$

Tak więc do wzoru (21.1) możemy podstawić wektor i macierz

$$\begin{aligned} \boldsymbol{f}(u_0, v_0) &= d\big(\boldsymbol{q}(u_0, v_0)\big)\overline{\boldsymbol{Q}}\big(C\boldsymbol{N}(u_0, v_0)\big), \\ \mathrm{D}\boldsymbol{f}(u_0, v_0) &= \Big[-\overline{\boldsymbol{Q}}\big(C\boldsymbol{P}_u(u_0, v_0)\big), -\overline{\boldsymbol{Q}}\big(C\boldsymbol{P}_v(u_0, v_0)\big)\Big]. \end{aligned}$$

Zauważmy, że kolejne kroki metody Newtona wymagałyby obliczania punktów i pochodnych płatów Béziera, a w tym celu szader fragmentów musiałby zawierać wszystkie procedury szadera rozdrabniania. Ale do wykonania pierwszego kroku wystarczy zadbać o to, aby szader fragmentów otrzymał na wejściu punkt $\boldsymbol{p}(u_0, v_0)$ i wektory pochodnych $\boldsymbol{p}_u(u_0, v_0)$ i $\boldsymbol{p}_v(u_0, v_0)$ oraz wektor normalny $\boldsymbol{n}(u_0, v_0)$ (w układzie współrzędnych modelu lub świata). W aplikacji 2F takie właśnie dane (w układzie świata), obliczone przez szader rozdrabniania dla wierzchołków trójkątów przybliżających płaty Béziera i przekazane dalej przez szader geometrii, etap rasteryzacji interpoluje i dostarcza szaderowi fragmentów, który musi tylko unormować wektor normalny.

Jeśli obliczony punkt (u_1, v_1) leży poza kwadratem jednostkowym (tj. poza dziedziną płata Béziera), to prawdopodobnie poszukiwany w tym kwadracie punkt $(u^\star, v^\star)$ nie istnieje. Najlepsze, co można wtedy zrobić, to liczbę u_1 lub v_1 „wystającą" poza przedział $[0, 1]$ zastąpić przez bliższy koniec tego przedziału, albo (jeśli funkcja $d\big(\boldsymbol{q}(u, v)\big)$ ma okresowe rozszerzenie na całą płaszczyznę) dodać taką liczbę całkowitą, aby suma należała do tego przedziału.

Listing 21.8 przedstawia sposób realizacji opisanego w tym punkcie sposobu modyfikowania współrzędnych tekstury odkształceń. Procedura `NTex56`, podobnie jak procedura `NTex12` na listingu 21.6, oblicza wartość funkcji $d = \tilde{d} \circ \boldsymbol{f}$ określającej kształt odkształcenia w podstawowej komórce tekstury i pochodne częstkowe tej funkcji. Procedura `NormalTex5`, podobnie jak `NormalTex12`, oblicza wektor normalny zaburzonej powierzchni. Jest ona wywoływana przez procedurę `NormalTexture5`, podobną do procedury `NormalTexture1` (dodana instrukcja w liniach `49–50` wybiera płaty rysowane bez odkształceń).

Modyfikacji współrzędnych tekstury dokonuje procedura `NormalTexture6`. Procedura `NTex56` jest przez nią wywoływana dwukrotnie. Przed każdym wywołaniem następuje obliczenie wartości funkcji $\boldsymbol{f} \circ \boldsymbol{q}$, przez przekształcenie punktu (u_0, v_0) albo (u_1, v_1) za pomocą macierzy będącej wartością zmiennej `dtxm5` i użycie funkcji `fract`. W liniach `65–69` procedura oblicza wektor $\boldsymbol{f}(u_0, v_0)$ i macierz $\mathrm{D}\boldsymbol{f}(u_0, v_0)$, a w linii `70` jest wykonywany krok metody Newtona, którego wynikiem jest punkt (u_1, v_1).

Listing 21.8. Procedura modyfikująca współrzędne tekstury odkształceń

GLSL

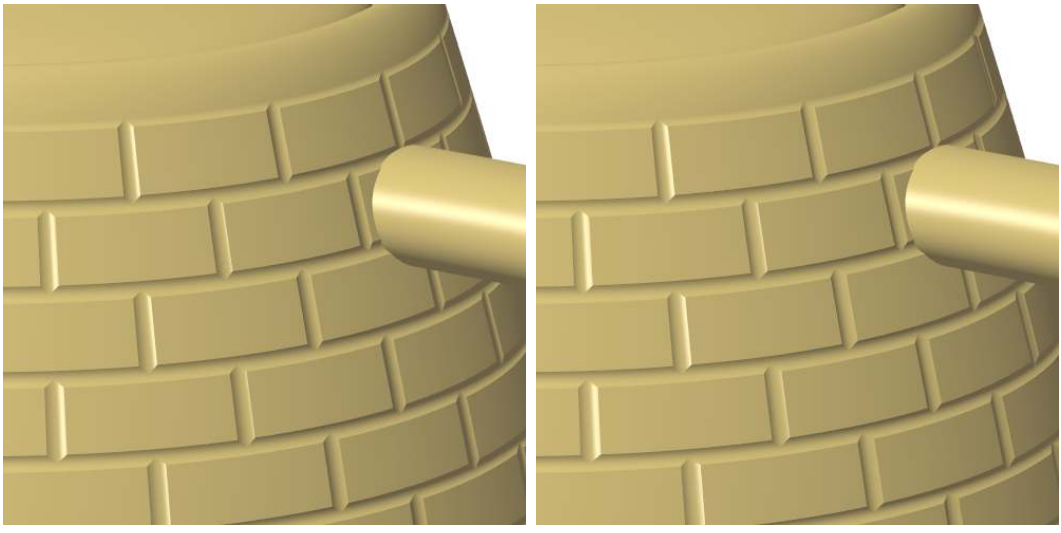
```
#define A 0.05
#define B (-0.003)

const mat3x2 dtxm5 = mat3x2 ( 5.0, 0.0,  0.0, 5.0,  0.0, 0.0 );

vec3 NTex56 ( float dx, float dy )
{
  float x, d, ds, dt;

  if ( dx >= 0.5 ) {
    dx -= 0.5;
    dy = dy >= 0.5 ? dy-0.5 : dy+0.5;
  }
  ds = dt = 0.0;
  if ( dx <= A ) {
    if ( dy <= dx ) { x = dy/A;  ds = 2.0*x/A; }
    else if ( dy >= 1.0-dx ) { x = (dy-1.0)/A;  ds = 2.0*x/A; }
    else { x = dx/A;  dt = 2.0*x/A; }
  }
  else if ( dx >= 0.5-A ) {
    if ( dx <= 0.5-dy ) { x = dy/A;  ds = 2.0*x/A; }
    else if ( dx <= dy-0.5 ) { x = (dy-1.0)/A;  ds = 2.0*x/A; }
    else { x = (dx-0.5)/A;  dt = 2.0*x/A; }
  }
  else {
    if ( dy <= A ) { x = dy/A;  ds = 2.0*x/A; }
    else if ( dy >= 1.0-A ) { x = (dy-1.0)/A;  ds = 2.0*x/A; }
    else return vec3 ( B, 0.0, 0.0 );
  }
  d = x*x;
  return B*vec3 ( d, dt, ds );
} /*NTex56*/

vec3 NormalTex5 ( vec2 pc, mat2 L, out float dz )
{
  vec2 Ddq;
  vec3 d, dpu, dpv;

  d = NTex56 ( pc.x, pc.y );
  dz = d.x;
  Ddq = L * d.yz;
  dpu = In.pu + Ddq.x*pnormal;
  dpv = In.pv + Ddq.y*pnormal;
  return normalize ( cross ( dpu, dpv ) );
} /*NormalTex5*/
```

```
46:
47: vec3 NormalTexture5 ( vec2 pc, out float dz )
48: {
49:   if ( In.instance < 4 || In.instance >= 8 )
50:     { dz = 0.0;  return pnormal; }
51:   pc = fract ( dtxm5*vec3 ( pc, 1.0 ) );
52:   return NormalTex5 ( pc, mat2(dtxm5), dz );
53: } /*NormalTexture5*/
54:
55: vec3 NormalTexture6 ( vec2 pc, out float dz )
56: {
57:   vec2 Ddq, f, pd;
58:   vec3 dd, dpu, dpv, APu, APv, AN;
59:   mat2 Df;
60:
61:   if ( In.instance < 4 || In.instance >= 8 )
62:     { dz = 0.0;  return pnormal; }
63:   pd = fract ( dtxm5*vec3 ( pc, 1.0 ) );
64:   dd = NTex56 ( pd.x, pd.y );
65:   APu = mat3(trb.vpm) * In.pu;
66:   APv = mat3(trb.vpm) * In.pv;
67:   AN = mat3(trb.vpm) * pnormal;
68:   f = dd.x*AN.xy;
69:   Df = mat2 ( APu.xy, APv.xy );
70:   pc -= inverse ( Df ) * f;
71:   pd = fract ( dtxm5*vec3 ( pc, 1.0 ) );
72:   dd = NTex56 ( pd.x, pd.y );
73:   dz = dd.x;
74:   Ddq = mat2(dtxm5) * dd.yz;
75:   dpu = In.pu + Ddq.x*pnormal;
76:   dpv = In.pv + Ddq.y*pnormal;
77:   return normalize ( cross ( dpu, dpv ) );
78: } /*NormalTexture6*/
79:
80: vec3 GetNormalVector ( vec2 pc, out float dz )
81: {
82:   vec3  nv;
83:
84:   switch ( NormalSource ) {
85:  ....
86:  case 5: nv = NormalTexture5 ( pc, dz );  break;
87:  case 6: nv = NormalTexture6 ( pc, dz );  break;
88:   }
89:   dz *= 0.5*dot ( M3ROW ( trb.vpm, 2 ), pnormal ) * gl_FragCoord.w;
90:   return nv;
91: } /*GetNormalVector*/
```

W linii 72 procedura `NTex56` zostaje wywołana ponownie, ze zmodyfikowanymi współrzędnymi tekstury. Obliczenie wielkości przesunięcia punktu odpowiadającego fragmentowi i wektora normalnego $\hat{\boldsymbol{n}}$ zaburzonego płata $\hat{\boldsymbol{p}}$ jest wykonywane tak samo jak w procedurze `NormalTex5`, przy czym *zamiast** wektorów $\boldsymbol{p}_u(u_1, v_1)$, $\boldsymbol{p}_v(u_1, v_1)$ i $\boldsymbol{n}(u_1, v_1)$, których szader fragmentów (ani żaden inny) nie oblicza, są podstawiane wektory $\boldsymbol{p}_u(u_0, v_0)$, $\boldsymbol{p}_v(u_0, v_0)$ i $\boldsymbol{n}(u_0, v_0)$. Obliczenie wektora $\hat{\boldsymbol{n}}$ następuje w linii 77.

W liniach 86–87 są pokazane dodane do procedury `GetNormalVector` instrukcje umożliwiające użycie opisanych tu tekstur.

W czterech miejscach w tekście tego punktu postawiłem gwiazdkę; każda z tych gwiazdek oznacza dokonane uproszczenie, którego skutkiem jest otrzymanie pewnego przybliżenia zamiast dokładnego rozwiązania postawionego zadania. Jeśli tekstura wprowadza dostatecznie małe odkształcenie powierzchni $\boldsymbol{p}$, to skutki tych uproszczeń są niezauważalne, a efekt na końcowym obrazie bywa istotnie lepszy niż efekt osiągnięty bez modyfikacji współrzędnych tekstury. Trzeba jednak pamiętać, że opisany tu algorytm nie zawsze musi dawać równie dobre wyniki. Jeśli na przykład pochodne cząstkowe funkcji d są nieciągłe w punktach, w których funkcja ta przyjmuje niezerowe wartości, to na końcowym obrazie mogą pojawić się wyraźnie widoczne błędy. Ponadto opisany w całym tym rozdziale sposób odkształcania powierzchni nigdy nie zmieni sylwetek rysowanych obiektów.



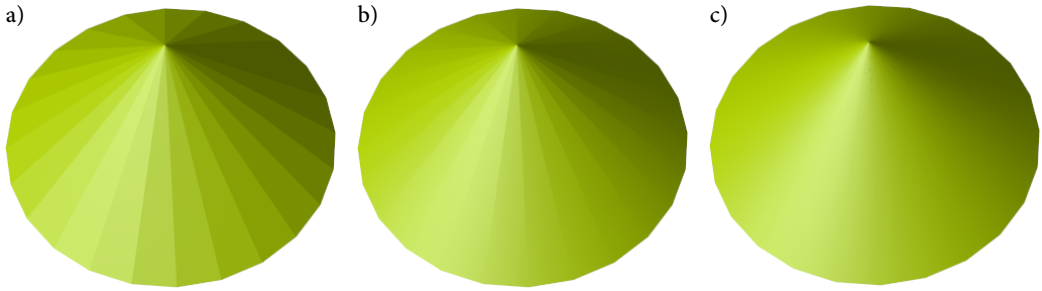
Rysunek 21.7. Porównanie oryginalnej i zmodyfikowanej tekstury odkształceń

Jeśli odkształcenia na obrazie są bardzo małe, to efekt modyfikowania współrzędnych tekstury jest niedostrzegalny. Różnice są widoczne w powiększeniu. Obrazek po lewej stronie rysunku 21.7 został wykonany bez modyfikacji, która była wprowadzona podczas wykonywania obrazka po prawej stronie; ocenę wyglądu tych powierzchni pozostawiam Czytelnikom.

21.5.3. *Rysowanie osobliwości punktowych

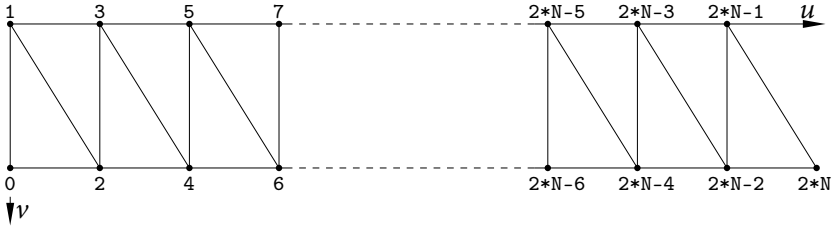
Osobliwość na powierzchni gładkiej jest to miejsce, w którym ta powierzchnia nie jest gładka[8]. Jak wiemy, obraz powierzchni gładkiej powstanie, gdy wyświetlając trójkąty przybliżające małe fragmenty takiej powierzchni, zastąpimy wektory normalne tych trójkątów wektorami normalnymi powierzchni i w szczególności w każdym wspólnym wierzchołku trójkątów podamy ten sam wektor normalny dla nich wszystkich. Jeśli na powierzchni ma być widoczny ostry grzbiet (tzn. krzywa, podczas przejścia przez którą wektor normalny zmienia się skokowo), to wystarczy osobno narysować gładkie fragmenty powierzchni po obu stronach grzbietu, przybliżając go łamaną składającą się z odpowiednich boków trójkątów.

Trudniej jest wykonać dobry obraz osobliwości punktowej, na przykład stożka z widocznym wierzchołkiem. Zobaczmy przykład na rysunku 21.8. Wszystkie obrazy przedstawiają te same trójkąty, przy czym na rysunku 21.8a oświetlenie było otrzymane za pomocą obliczonych przez szader geometrii wektorów normalnych tych trójkątów. W tablicy wierzchołków punkt osobliwy — wierzchołek stożka — był podany tylko raz, a procedura `glDrawArrays` otrzymała pierwszy parametr `GL_TRIANGLE_FAN`.



Rysunek 21.8. Obrazy powierzchni stożkowej

Pozostałe dwa obrazy powstały przez wyświetlenie taśm trójkątowych; wierzchołki miały w tablicy wierzchołków podane położenia i wektory normalne. W szczególności $N = 24$ egzemplarze wierzchołka stożka miały różne wektory normalne.



Rysunek 21.9. Taśma trójkątowa dla stożka

[8]Na powierzchni parametrycznej jest to miejsce, w którym pochodne cząstkowe parametryzacji są nieciągłe lub liniowo zależne.

Rysunek 21.9 przedstawia schemat użytej taśmy, składającej się z $2N-1$ trójkątów. Wierzchołki o numerach nieparzystych odpowiadają wierzchołkowi stożka i mają to samo położenie (w punkcie $(0,0,1)$), wskutek czego co drugi trójkąt w taśmie jest zdegenerowany do odcinka i nie jest widoczny na obrazach. Wierzchołki o numerach parzystych są rozmieszczone na okręgu; wierzchołek o numerze $2i$ jest położony w punkcie $(\cos\frac{2i\pi}{N}, \sin\frac{2i\pi}{N}, 0)$.

Obraz na rysunku 21.8b powstał w ten sposób, że z wierzchołkiem o numerze i w taśmie został związany wektor $(\cos\frac{i\pi}{N}, \sin\frac{i\pi}{N}, 1)$. Jeśli i jest parzyste, to jest to wektor normalny powierzchni stożka w punkcie opowiadającym wierzchołkowi taśmy. W wierzchołku stożka wektor normalny jest nieokreślony (to właśnie jest osobliwość), ale wektory przypisane nieparzystym wierzchołkom taśmy są wektorami normalnymi powierzchni stożka na jego odpowiednich tworzących.

Otrzymany efekt nie jest idealny, bo wprawdzie na brzegu powierzchni wektor otrzymany przez interpolację wektorów podanych jako wektory normalne wierzchołków taśmy zmienia się w sposób ciągły, ale nieciągłości na wspólnych krawędziach trójkątów są wyraźnie widoczne w pobliżu wierzchołka stożka. Aby otrzymać obraz powierzchni gładkiej (poza osobliwością), taki jak na rysunku 21.8c, trzeba zadbać o ciągłość połączenia wektorów normalnych otrzymanych przez interpolację wzdłuż całych wspólnych krawędzi trójkątów.

Wspólna krawędź narysowanych trójkątów jest tworzącą stożka i w każdym jej punkcie (oprócz wierzchołka stożka) wektor normalny jest taki sam, dlatego powinien być wspólny dla obu trójkątów przylegających do tej krawędzi. Ale to oznacza, że w wierzchołku trójkąta odpowiadającym wierzchołkowi stożka muszą być podane *dwa* wektory normalne — po jednym dla każdej wychodzącej z niego krawędzi. Obraz na rysunku 21.8c powstał przez wyświetlenie taśmy z rysunku 21.9, której każdy wierzchołek miał określone położenie w przestrzeni, trzy wektory normalne (trzecim był wektor normalny trójkąta) i dwie współrzędne, u, v, potrzebne do interpolacji wektorów normalnych stożka. W sumie atrybuty każdego wierzchołka trójkąta poddanego rasteryzacji były opisane przez 14 liczb, z których 11 było podanych w buforze wierzchołków, a pozostałe 3 (współrzędne wektora normalnego płaszczyzny trójkąta) obliczył szader geometrii.

Listing 21.9 przedstawia procedurę obliczającą dla podanej liczby N atrybuty wierzchołków trójkątów, których obraz ma być obrazem stożka (tj. wspomniane 11 liczb dla każdego wierzchołka), i przesyła je do pamięci GPU.

W liniach 6–9 procedura rezerwuje obiekt tablicy wierzchołków i bufor na ich atrybuty. W pętli w liniach 12–18 są obliczane atrybuty wierzchołków o parzystych numerach w taśmie. Kolejne wartości nadawane zmiennej `j` określają miejsca przechowywania w buforze pierwszego atrybutu (współrzędnej x położenia) bieżącego wierzchołka (o numerze `2*i`). Dla każdego z tych wierzchołków oba wektory normalne są identyczne. Współrzędna u wierzchołka o numerze $2i$ otrzymuje wartość i, a współrzędna v jest równa 1.

Wszystkie atrybuty ostatniego wierzchołka w taśmie (o numerze $2N$) z wyjątkiem współrzędnej u, która ma wartość N, są identyczne z atrybutami pierwszego wierzchołka. Nadanie im wartości odbywa się w liniach 19–20. Pętla w liniach 21–27 nadaje wartości atrybutom wierzchołków nieparzystych. Wszystkie one mają to samo położenie. Wektory normalne są kopiowane z sąsiednich wierzchołków parzystych (które dlatego zostały wygenerowane

Listing 21.9. Tworzenie taśmy trójkątowej do rysowania stożka

```C
void InitConeVAO ( int N, GLuint *conevao, GLuint *conevbo )
{
  GLfloat *auxb;
  int     i, j;

  glGenVertexArrays ( 1, conevao );
  glBindVertexArray ( *conevao );
  glGenBuffers ( 1, conevbo );
  glBindBuffer ( GL_ARRAY_BUFFER, *conevbo );
  if ( !(auxb = malloc ( (2*N+1)*11*sizeof(GLfloat) )) )
    ExitOnError ( "InitConeVAO" );
  for ( i = j = 0;  i < N;  i++, j += 22 ) {
    auxb[j] = auxb[j+3] = auxb[j+6] = cos ( i*2.0*PI/(float)N );
    auxb[j+1] = auxb[j+4] = auxb[j+7] = sin ( i*2.0*PI/(float)N );
    auxb[j+2] = 0.0;
    auxb[j+5] = auxb[j+8] = 1.0;
    auxb[j+9] = (GLfloat)i;  auxb[j+10] = 1.0;
  }
  memcpy ( &auxb[j], &auxb[0], 11*sizeof(GLfloat) );
  auxb[j+9] = (GLfloat)N;
  for ( i = 0, j = 11;  i < N;  i++, j += 22 ) {
    auxb[j] = auxb[j+1] = 0.0;
    auxb[j+2] = 1.0;
    memcpy ( &auxb[j+3], &auxb[j-8], 3*sizeof(GLfloat) );
    memcpy ( &auxb[j+6], &auxb[j+14], 3*sizeof(GLfloat) );
    auxb[j+9] = (GLfloat)i;  auxb[j+10] = 0.0;
  }
  glBufferData ( GL_ARRAY_BUFFER, (2*N+1)*11*sizeof(GLfloat),
                 auxb, GL_STATIC_DRAW );
  glEnableVertexAttribArray ( 0 );
  glVertexAttribPointer ( 0, 3, GL_FLOAT, GL_FALSE, 11*sizeof(GLfloat),
                          (GLvoid*)0 );
  glEnableVertexAttribArray ( 1 );
  glVertexAttribPointer ( 1, 3, GL_FLOAT, GL_FALSE, 11*sizeof(GLfloat),
                          (GLvoid*)(3*sizeof(GLfloat)) );
  glEnableVertexAttribArray ( 2 );
  glVertexAttribPointer ( 2, 3, GL_FLOAT, GL_FALSE, 11*sizeof(GLfloat),
                          (GLvoid*)(6*sizeof(GLfloat)) );
  glEnableVertexAttribArray ( 3 );
  glVertexAttribPointer ( 3, 2, GL_FLOAT, GL_FALSE, 11*sizeof(GLfloat),
                          (GLvoid*)(9*sizeof(GLfloat)) );
  free ( auxb );
  glBindVertexArray ( 0 );
  ExitIfGLError ( "InitConeVAO" );
} /*InitConeVAO*/
```

wcześniej). Dla wierzchołka $2i+1$ mamy „lewy" wektor normalny, odpowiadający krawędzi łączącej go z wierzchołkiem $2i$ i „prawy" wektor normalny, który jest wektorem normalnym powierzchni stożkowej na krawędzi dochodzącej do wierzchołka $2i+2$. Współrzędne u i v wierzchołka $2i+1$ otrzymują wartości i oraz 0.

Dane są przesyłane do pamięci GPU w liniach 28–29, po czym w obiekcie tablicy wierzchołków są rejestrowane poszczególne atrybuty: położenie wierzchołka otrzymuje numer 0, „lewy" wektor normalny numer 1, „prawy" wektor normalny numer 2 i wektor współrzędnych (u, v) numer 3.

Dowolny szader części przedniej potoku przetwarzania grafiki musi obliczyć współrzędne położenia wierzchołka w układach świata i kostki standardowej oraz współrzędne obu wektorów normalnych stożka w układzie świata[9]. Do tego szader geometrii ma obliczyć współrzędne wektora normalnego płaszczyzny trójkąta. Dane te oraz współrzędne u, v mają trafić do etapu rasteryzacji. Szader fragmentów otrzyma wynik ich interpolacji. Liczba v leży w przedziale $[0, 1]$, a liczba u jest nieujemna. Jeśli część ułamkowa współrzędnej u jest równa 0, to fragment odpowiada punktowi na „lewej" krawędzi trójkąta (zobacz rys. 21.9). Dla punktów na „prawej" krawędzi część ułamkowa u jest równa v. Zatem wynikiem interpolacji między wektorami normalnymi „lewym", $\boldsymbol{n}_L$, a „prawym", $\boldsymbol{n}_R$, powinien być wektor

$$\boldsymbol{n} = (1-t)\boldsymbol{n}_L + t\boldsymbol{n}_R,$$

gdzie $t = \lfloor u \rfloor / v$. Obliczenie to wykonuje procedura pokazana na listingu 21.10, która w przypadku gdy $v = 0$, oblicza sumę $\boldsymbol{n}_L + \boldsymbol{n}_R$. I, oczywiście, dokonuje normalizacji, aby podać do obliczeń oświetlenia wektor jednostkowy.

Listing 21.10. Procedura interpolacji wektorów normalnych szadera fragmentów

GLSL

```
in FVertex {
    vec3      Position;
    vec3      LNormal, RNormal;
    flat vec3 TNormal;
    vec2      uv;
  } In;

vec3 NormalVector ( void )
{
  if ( In.uv.y > 0.0 )
    return normalize ( mix ( In.LNormal, In.RNormal,
                             fract(In.uv.x)/In.uv.y ) );
  else
    return normalize ( In.LNormal+In.RNormal );
} /*NormalVector*/
```

[9] Na wejściu szadera wierzchołków położenie wierzchołka i wektory normalne są podane w układzie modelu. Jeśli przejście od układu modelu do układu świata opisuje macierz M, to wektory normalne trzeba pomnożyć przez macierz M^{-T} i unormować. W bloku zmiennych jednolitych `TransBlock`, zadeklarowanym w szaderach opisanych w tej książce, macierze M i M^{-T} są przechowywane w polach `mm` i `mmti`.

Ćwiczenie. Napisz szader wierzchołków, który oblicza współrzędne wierzchołka i wektory normalne na podstawie numeru wierzchołka w taśmie i liczby trójkątów podanej w zmiennej jednolitej, oraz procedurę w C rysującą stożek bez używania bufora ze współrzędnymi wierzchołków, przy czym poziom szczegółowości (tzn. liczba trójkątów) ma być określony przez parametr procedury. Całość ma działać podobnie do procedury rysowania walca podanej na listingach 12.16 i 12.17.

Ćwiczenie. Napisz szadery i procedury w C umożliwiające rysowanie powierzchni stożkowych nieobrotowych, tj. składających się z półprostych wychodzących z jednego punktu i przechodzących przez punkty zadanej krzywej gładkiej, zamkniętej lub nie.

21.5.4. Anizotropowy model oświetlenia

Jak zobaczyliśmy, na odbijanie światła przez powierzchnię ma zasadniczy wpływ jej faktura[10], czyli drobne szczegóły kształtu, w tym także niewidoczne gołym okiem chropowatości. Opis modeli oświetlenia opartych na analizie fizycznego oddziaływania światła z powierzchnią obiektu jest podany w podrozdziale 28.4. Podstawą tych modeli jest założenie, że chropowata powierzchnia składa się z **mikrościanek**. Padający na mikrościankę foton, jeśli przez nią nie przejdzie, odbije się zgodnie z zasadami optyki geometrycznej — w kierunku wyznaczonym przez wektor normalny mikrościanki. Dlatego jednym z elementów każdego takiego modelu jest jakiś ustalony rozkład kierunków wektorów normalnych mikrościanek chropowatej powierzchni. Intensywność światła dochodzącego do obserwatora zależy od tego, jak wiele mikrościanek w otoczeniu danego punktu na powierzchni jest ustawionych tak, że światło dochodzące z kierunku $\boldsymbol{l}_i$ (od i-tego źródła światła) zostaje przez nie odbite w kierunku wektora $\boldsymbol{v}$ (czyli do obserwatora). Możemy zauważyć, że wektorem normalnym tych właśnie mikrościanek jest wektor $\boldsymbol{h}_i$ występujący we wzorze (18.2).

W modelach oświetlenia Phonga i Blinna-Phonga powierzchnie są **izotropowe**, co oznacza, że obracanie obiektu wokół osi prostopadłej do powierzchni w jej ustalonym punkcie nie spowoduje żadnych zmian koloru tego punktu na końcowym obrazie. Tymczasem wiele obiektów[11] ma powierzchnie chropowate w sposób **anizotropowy** — są na nich drobne rysy mające określony kierunek. Rozkład mikrościanek takiej powierzchni sprawia, że choć same rysy nie są widoczne, ich obecność i kierunek są natychmiast zauważalne.

Najprostszy model oświetlenia powierzchni anizotropowej można otrzymać, modyfikując model Blinna-Phonga. W tym celu trzeba określić kierunek rys, podając pewien wektor $\boldsymbol{r}$, styczny do powierzchni. Dla płata parametrycznego można określić odpowiedni wektor jako kombinację liniową wektorów pochodnych cząstkowych parametryzacji[12]. Modyfikacja

[10] Odpowiednikiem angielskiego słowa *texture* jest „faktura" i w zasadzie to tego słowa powinno się używać w polskiej mowie i piśmie. Ale pozwoliłem sobie rozróżnić znaczenia: słowa faktura używam tylko w odniesieniu do detali kształtu takich jak chropowatość, natomiast słowem „tekstura" określam *wszystkie* własności powierzchni wpływające na jej wygląd, a więc zarówno chropowatość, jak i kolor (obraz nałożony na powierzchnię obiektu), przezroczystość itp.

[11] na przykład czajników

[12] Oczywiście, jest to tylko jeden z wielu możliwych sposobów.

polega na zastąpieniu wektora $\boldsymbol{h}_i$ we wzorze (18.2) przez wektor $\tilde{\boldsymbol{h}}_i$ określony wzorami

$$\tilde{\boldsymbol{h}}_i = \frac{1}{\|\hat{\boldsymbol{h}}_i\|}\hat{\boldsymbol{h}}_i, \quad \hat{\boldsymbol{h}}_i = P_0\boldsymbol{w}_i + P_1\boldsymbol{w}_i + aP_2\boldsymbol{w}_i, \quad \boldsymbol{w}_i = \boldsymbol{l}_i + \boldsymbol{v}.$$

We wzorach tych symbole P_0, P_1 i P_2 oznaczają rzuty prostopadłe (podrozdz. 5.5) odpowiednio na kierunek wektora normalnego powierzchni $\boldsymbol{n}$, na kierunek wektora $\boldsymbol{r}$ i na kierunek prostopadły do wektorów $\boldsymbol{n}$ i $\boldsymbol{r}$. Współczynnik $a \in [0,1]$ określa „stopień anizotropowości" powierzchni; im ten współczynnik jest bliższy zera, tym silniejszy jest efekt anizotropii, a dla $a = 1$ otrzymamy $\tilde{\boldsymbol{h}}_i = \boldsymbol{h}_i$, czyli powierzchnię izotropową. Zatem modyfikacji uległ tylko składnik opisujący odblask światła na powierzchni. Składnik lambertowski (opisujący odbicie rozproszone) pozostał bez zmian.

Przed napisaniem szadera przekształcimy wzory. Mając wektory *jednostkowe* $\boldsymbol{n}$ i $\boldsymbol{r}$, możemy obliczyć

$$P_0\boldsymbol{w}_i = \boldsymbol{n}\langle \boldsymbol{n}, \boldsymbol{w}_i\rangle, \quad P_1\boldsymbol{w}_i = \boldsymbol{r}\langle \boldsymbol{r}, \boldsymbol{w}_i\rangle, \quad P_2\boldsymbol{w}_i = \boldsymbol{w}_i - P_0\boldsymbol{w}_i - P_1\boldsymbol{w}_i,$$

skąd wynika, że

$$\hat{\boldsymbol{h}}_i = \boldsymbol{n}\langle \boldsymbol{n}, \boldsymbol{w}_i\rangle + \boldsymbol{r}\langle \boldsymbol{r}, \boldsymbol{w}_i\rangle + a\boldsymbol{w}_i.$$

Możemy teraz zmodyfikować szader fragmentów w sposób pokazany na listingu 21.11. Wystarczy dodać jedną zmienną jednolitą i dwa podprogramy, z których drugi powstał z kopii procedury realizującej model Blinna-Phonga przez zmianę nazwy oraz dodanie dwóch instrukcji i zmienienie innych dwóch. Dodane instrukcje (linie 18–19) obliczają wektor $\boldsymbol{r}$ na podstawie pochodnych cząstkowych parametryzacji (płata Béziera), $\boldsymbol{p}_u$ i $\boldsymbol{p}_v$, otrzymanych w bloku wejściowym `In`. W linii 18 jest obliczany wektor $\hat{\boldsymbol{r}} = c_1\boldsymbol{p}_u + c_2\boldsymbol{p}_v$, przy czym współczynniki c_1, c_2 są współrzędnymi x, y wektora przypisanego zmiennej jednolitej `anc`. Wektor $\hat{\boldsymbol{r}}$ jest styczny do powierzchni oryginalnej, zatem wektor normalny powierzchni zaburzonej przez nałożenie tekstury odkształceń może nie być do niego prostopadły. Dlatego w linii 19 wektor $\hat{\boldsymbol{r}}$ jest rzutowany na płaszczyznę prostopadłą do wektora normalnego i normalizowany. Otrzymany w ten sposób wektor jednostkowy $\boldsymbol{r}$ jest styczny do powierzchni zaburzonej.

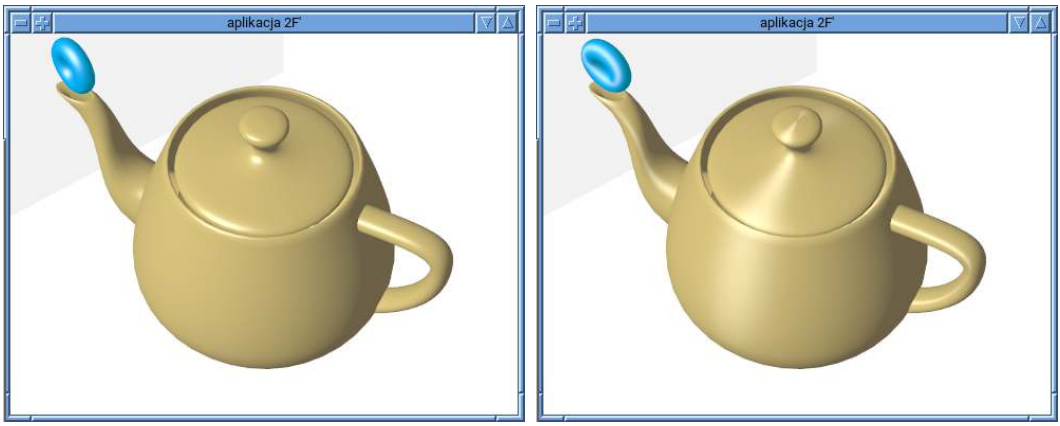
Wektor $\tilde{\boldsymbol{h}}_i$ jest obliczany przez procedurę `AnisotropicHVector` na podstawie wzorów wyprowadzonych wyżej. Współczynnik a, określający stopień anizotropowości, jest współrzędną z wektora przechowywanego w zmiennej jednolitej `anc`.

Pozostawiam jako ćwiczenie dla Czytelników takie zmodyfikowanie szadera fragmentów i aplikacji, aby umożliwić nadawanie zmiennej jednolitej `anc` odpowiednich wartości i przypisywanie zmiennej `LightingModel` wartości 2, która spowoduje wywołanie (w instrukcji przełącznika) procedury `AnisotropicLighting` jako alternatywy dla procedur używanych wcześniej. Rysunek 21.10 przedstawia wynik otrzymany ze współczynnikami $c_1 = 0$ i $c_2 = 1$

Listing 21.11. Procedury anizotropowego modelu oświetlenia

GLSL

```
uniform vec3 anc;

vec3 AnisotropicHVector ( vec3 w, vec3 normal, vec3 r )
{
  vec3 p0w, p1w;

  p0w = normal*dot ( normal, w );
  p1w = r*dot ( r, w );
  return normalize ( p0w + p1w + anc.z*w );
} /*AnisotropicHVector*/

vec3 AnisotropicLighting ( void )
{
  vec3  r, lv, vv, hv, Colour;
  float a, d, e, f, dist;
  uint  i, mask;

  r = anc.x*In.pu + anc.y*In.pv;
  r = normalize ( r - normal*dot ( r, normal ) );
  vv = posDifference ( trb.eyepos, In.Position, dist );
  e = dot ( vv, tnormal );
  ....
  Colour = vec3(0.0);
  for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
    if ( (light.mask & mask) != 0 ) {
      if ( e > 0.0 ) {
        ....
          Colour += (a * d * light.ls[i].direct) * mm.dirref;
          hv = AnisotropicHVector ( lv+vv, normal, r );
          f = pow ( dot ( hv, normal ), mm.shininess );
          Colour += (a*f*wFactor(d,mm.wa,mm.we))*mm.specref;
        ....
      }
      else {
        ....
          Colour -= (a * d * light.ls[i].direct) * mm.dirref;
          hv = AnisotropicHVector ( lv+vv, normal, r );
          f = pow ( -dot ( hv, normal ), mm.shininess );
          Colour += (a*f*wFactor(-d,mm.wa,mm.we))*mm.specref;
        ....
      }
    }
  return clamp ( Colour, 0.0, 1.0 );
} /*AnisotropicLighting*/
```



Rysunek 21.10. Porównanie modelu oświetlenia Blinna-Phonga z modelem anizotropowym

(którym odpowiadają rysy pokrywające się z liniami stałego parametru v wszystkich płatów Béziera) oraz $a = 0.2$.

22

Aplikacja druga G

Z dosyć już głębokiej wody przeskoczymy do jeszcze trochę głębszej: wprowadzimy na obrazach cienie. Nasze źródła światła są punktowe. Dowolny punkt $\boldsymbol{p}$ w przestrzeni jest **bezpośrednio oświetlony** przez punktowe źródło światła położone w punkcie $\boldsymbol{z}$, jeśli odcinek $\overline{\boldsymbol{zp}}$ nie ma żadnego punktu wspólnego z powierzchnią dowolnego obiektu w rysowanej scenie — z wyłączeniem punktu $\boldsymbol{p}$, który (jeśli jest punktem rysowanej powierzchni) zasłania przed światłem punkty położone dalej[1]. To samo dotyczy źródeł światła położonych w nieskończonej odległości od sceny (tylko że wtedy rozważamy półprostą zamiast odcinka).

Zbiór punktów, które nie są bezpośrednio oświetlone przez dane źródło światła, nazwiemy **obszarem cienia** tego źródła. Aby otrzymać poprawne cienie na obrazie, należy dla każdego fragmentu (piksela) zbadać, czy punkty powierzchni odpowiadające temu fragmentowi są bezpośrednio oświetlone, czy też należą do obszaru cienia. Algorytmy cieni odpowiednie do implementacji działającej na GPU tworzą pewną reprezentację obszaru cienia. Są dwa podstawowe sposoby reprezentowania go.

Pierwszy sposób[2] polega na zbudowaniu i narysowaniu **bryły cienia**, która jest (teoretycznie) sumą **elementarnych brył cienia**, tj. ściętych nieograniczonych ostrosłupów (dla źródeł światła w odległości skończonej) lub graniastosłupów. „Przednią" podstawą każdego z nich jest trójkątna ściana należącego do sceny obiektu, a boczne ściany są zbiorami punktów zasłoniętych od światła przez krawędzie tego trójkąta. Elementarne bryły cienia trzeba obciąć do bryły widzenia, wyznaczając ściany części wspólnej tych brył[3]. Wszystkie ściany brył cienia muszą być spójnie zorientowane, na przykład tak, aby wektor normalny obliczony dla podanej kolejności wierzchołków był zwrócony na zewnątrz bryły. Scenę rysuje się dwukrot-

[1] Ale żaden punkt sam siebie od światła nie zasłania, chyba że obserwator i źródło światła znajdują się po przeciwnych stronach nieprzezroczystej powierzchni, na której leży ten punkt.

[2] Wynalazł go w 1977 r. Frank Crow, dalej rozwijało go wiele osób, przy czym najczęściej łączy się to podejście z nazwiskiem Johna Carmacka, który użył go w 2000 r. w grze *Doom 3*.

[3] Celem jest otrzymanie obiektu z zamkniętą objętością. Można uniknąć wyznaczania ścian przecięcia, wprowadzając dla każdej elementarnej bryły cienia jej „tylną ścianę", tj. „drugą podstawę" za obszarem, w którym znajdują się obiekty sceny. Wtedy trzeba rysować ściany brył cienia z włączoną za pomocą procedury `glEnable` opcją `GL_DEPTH_CLAMP`.

nie, przy czym jedynym potrzebnym wynikiem pierwszego rysowania jest zawartość bufora głębokości — informacja o głębokości widocznego punktu dla każdego piksela. Następnie trzeba narysować wszystkie ściany elementarnych brył cienia. Dla każdego piksela wprowadzamy licznik o początkowej wartości 0. Rysując w tym kroku fragment, trzeba porównać głębokość punktu z liczbą zapisaną w buforze głębokości dla danego piksela i dalej przetwarzać tylko te fragmenty, których głębokość jest mniejsza (albo większa — mamy dwa warianty algorytmu, zwane *depth pass* i *depth fail*). Dla fragmentów ścian odwróconych przodem licznik zwiększamy, a dla odwróconych tyłem zmniejszamy o 1. Ostatni krok algorytmu to ponowne rysowanie sceny. Obliczenia oświetlenia możemy wykonywać tylko dla fragmentów widocznych (o głębokości równej głębokości zapamiętanej po pierwszym rysowaniu sceny), co oszczędza czas. Jeśli licznik dla danego piksela ma wartość 0, to punkt jest oświetlony[4]. Efektywne implementacje, umożliwiające nawet animowanie cieni w czasie rzeczywistym (np. w grach), wykorzystują bufor maski (*stencil buffer* — wspomniane liczniki mogą być przechowywane w nim), czym tu się jednak nie zajmiemy.

Zastosujemy drugie podejście, które polega na narysowaniu obrazu sceny widzianej z punktu położenia źródła światła lub z kierunku padania światła dochodzącego z nieskończonej odległości. Ten obraz jest niepotrzebny, ale otrzymana podczas jego tworzenia zawartość bufora głębokości (nazywana niestety **mapą cienia**[5]) dla obrazu o dostatecznie dużej rozdzielczości jest dostatecznie dokładną reprezentacją obszaru cienia. Podczas rysowania właściwego obrazu szader dla każdego fragmentu sprawdzi, czy odpowiadający temu fragmentowi punkt w przestrzeni należy do obszaru cienia i uwzględni ten fakt w obliczeniach oświetlenia.

22.1. Konstrukcja rzutowania sceny dla źródeł światła

Jeśli źródło światła jest położone w nieskończonej odległości od sceny, to potrzebny jest rzut równoległy; kierunek rzutowania jest kierunkiem do źródła światła, a rzutnia powinna (choć teoretycznie nie musi) być do tego kierunku prostopadła. Jeśli odległość do źródła światła jest skończona, to potrzebny jest rzut perspektywiczny. Rozwiążemy problem uproszczony, w którym źródło światła leży dostatecznie daleko od sceny, aby można było ją przedstawić w całości na obrazie wykonanym w jednym rzucie[6]. Założymy, że scena jest zawarta w kuli[7] o środku $\boldsymbol{s}$ i promieniu R. Przyjmiemy, że opisane tu rozwiązanie jest stosowalne, jeśli odleg-

[4] Jest tak przy założeniu, że położenie obserwatora jest poza bryłą cienia, jeśli w etapie drugim braliśmy pod uwagę fragmenty o głębokości mniejszej. Pod tym względem lepszy jest drugi wariant (*depth fail*), w którym licznik zwiększamy lub zmniejszamy, jeśli punkt na ścianie bryły cienia ma większą głębokość, bo w tym wariancie nie ma znaczenia, czy obserwator jest w cieniu, czy nie jest.

[5] Ja tak nie zamierzam.

[6] Jeśli tak nie jest (np. gdy źródło światła jest lampą umieszczoną między obiektami w pomieszczeniu), to punkt położenia źródła światła trzeba „obudować" sześcienną kostką i narysować obrazy sceny na wszystkich ścianach tej kostki. Sposób zrobienia tego jest przedstawiony w p. 26.4.3, a tymczasem zajmiemy się prostszym przypadkiem.

[7] Jest, mam nadzieję, jasne, że aby otrzymać dobrą dokładność obrazu cienia, dla każdej konkretnej sceny trzeba starać się o znalezienie otaczającej scenę kuli o jak najmniejszym promieniu.

łość d położenia $\boldsymbol{z}$ źródła światła od punktu $\boldsymbol{s}$ jest nie mniejsza niż $R\sqrt{2}$, co daje gwarancję, że całą scenę można zrzutować (perspektywicznie) na jedną ścianę pewnego sześcianu, którego środek — położenie źródła światła — jest środkiem rzutowania. W obu przypadkach potrzebujemy macierzy V opisującej przejście od układu świata do układu obserwatora i macierzy P reprezentującej przejście od układu obserwatora do układu kostki standardowej. Założymy, że obraz jest kwadratowy, tj. odpowiedni kwadrat położony na rzutni będzie odwzorowany na klatkę, której wysokość i szerokość w pikselach są takie same.

Zaczniemy od skonstruowania macierzy przekształceń dla źródeł światła położonych w odległości nieskończonej. Początek układu obserwatora umieścimy w punkcie $\boldsymbol{s}$. Wektor $\boldsymbol{l}$ opisujący kierunek, z którego dochodzi światło, po unormowaniu ma być wersorem osi z w układzie obserwatora. Przyjmiemy, że wersory osi x i y w układzie świata są wektorami jednostkowymi i wszystkie trzy wersory mają być do siebie nawzajem prostopadłe. Zauważmy, że te warunki zostawiają nam sporą swobodę — możemy te dwa wektory dowolnie obracać wokół osi z, możemy też zmienić ich zwroty lub kolejność (czyli zmienić orientację układu). Dlatego w konstrukcji użyjemy **odbicia Householdera**, którego skutkiem będzie przypadkowe (ale poprawne) położenie osi x i y, i które ma dobre własności numeryczne. Przeprowadzi ono wektor $\boldsymbol{e}_3 = (0,0,1)$ (wersor osi z układu świata) na wektor o kierunku $\boldsymbol{l}$. W tym celu obliczamy wektor normalny płaszczyzny odbicia

$$\boldsymbol{v} = \boldsymbol{l} \pm \|\boldsymbol{l}\|\boldsymbol{e}_3,$$

przy czym wybieramy znak „+" albo „−" tak, aby wektor $\boldsymbol{v}$ był jak najdłuższy (zatem wybieramy „+", jeśli trzecia współrzędna wektora $\boldsymbol{l}$ jest dodatnia, a „−" w przeciwnym razie). Dalej obliczamy czynnik $\gamma = 2/\langle \boldsymbol{v}, \boldsymbol{v}\rangle$ i poddajemy odbiciu wersory osi x i y (wektory $\boldsymbol{e}_1 = (1,0,0)$ i $\boldsymbol{e}_2 = (0,1,0)$), tj. obliczamy

$$\boldsymbol{w}_i = \boldsymbol{e}_i - \boldsymbol{v}\gamma\langle \boldsymbol{v}, \boldsymbol{e}_i\rangle, \quad i = 1,2.$$

Następnie konstruujemy macierz 4×4 opisującą przejście od układu obserwatora do układu świata:

$$V^{-1} = \begin{bmatrix} \boldsymbol{w}_1 & \boldsymbol{w}_2 & \boldsymbol{w}_3 & \boldsymbol{s} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \text{gdzie} \quad \boldsymbol{w}_3 = \frac{1}{\|\boldsymbol{l}\|}\boldsymbol{l}.$$

Macierz V przejścia od układu świata do układu obserwatora otrzymamy, znajdując odwrotność[8] macierzy V^{-1}.

[8] Przejście opisane przez macierz V^{-1} jest afiniczną izometrią; macierz ta ma strukturę

$$\begin{bmatrix} Q & \boldsymbol{s} \\ \boldsymbol{0}^T & 1 \end{bmatrix}$$

z blokiem ortogonalnym $Q = [\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3]$. Odwrotność takiej macierzy ma postać

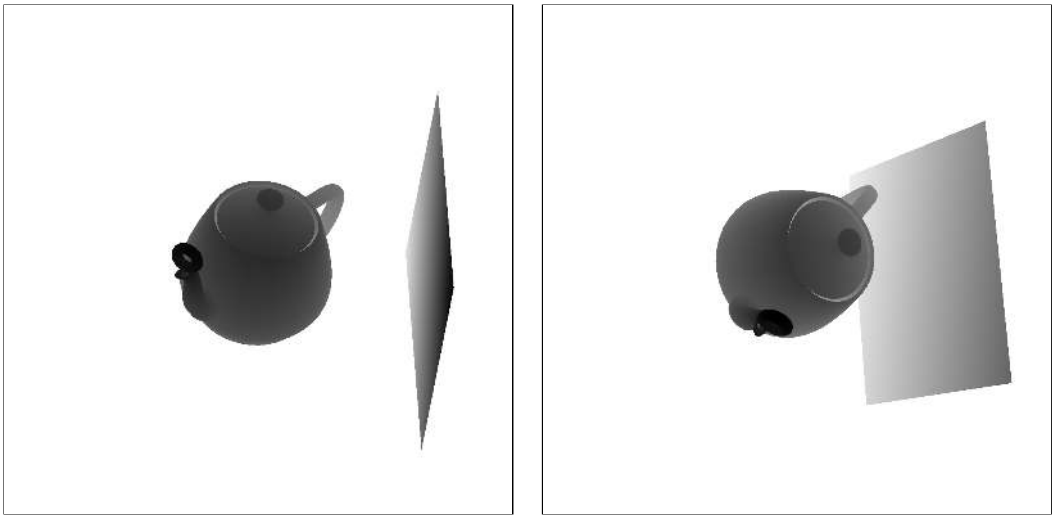
$$\begin{bmatrix} Q^T & -Q^T\boldsymbol{s} \\ \boldsymbol{0}^T & 1 \end{bmatrix}.$$

Użycie w tym przypadku ogólnej procedury znajdującej macierz odwrotną jest trochę mało eleganckie, choć dopuszczalne. Wzór podany wyżej jest tańszy i daje dokładniejszy wynik.

Zawierająca scenę kula o promieniu R i środku $\boldsymbol{s}$ jest w układzie obserwatora kulą o promieniu R i środku $\mathbf{0}$, możemy ją zatem wpisać w kostkę $[-R, R]^3$. Stąd współczynniki odpowiedniej macierzy P obliczy procedura `M4x4Orthof` (zobacz rozdz. 6) wywołana z parametrami $l = b = n = -R$ i $r = t = f = R$.

Dla źródła światła położonego w punkcie $\boldsymbol{z}$ (w skończonej odległości od sceny) układ obserwatora ma mieć początek w punkcie $\boldsymbol{z}$. Oś z tego układu będzie miała kierunek i zwrot wektora $\boldsymbol{z} - \boldsymbol{s}$ o długości d. Konstruując macierz V^{-1} przejścia od układu obserwatora do układu świata, podstawimy ten wektor zamiast wektora $\boldsymbol{l}$ w opisanym wyżej rachunku dla światła dochodzącego z nieskończenie daleka, po czym pierwsze dwie kolumny macierzy V^{-1} obliczymy tak jak poprzednio, a trzecią, normując wektor $\boldsymbol{z}-\boldsymbol{s}$. Macierz P, opisującą przejście do układu kostki standardowej, skonstruuje procedura `M4x4Frustumf` wywołana z parametrami $n = d - R$, $f = d + R$, $l = b = -n \operatorname{tg} \alpha$, $r = t = n \operatorname{tg} \alpha$, gdzie $\sin \alpha = R/d$.

Na rysunku 22.1 jest pokazana zawartość bufora głębokości dla dwóch źródeł światła oświetlających scenę rysowaną przez naszą aplikację. Jasność punktów na obrazach jest powiązana z ich głębokością — im punkt jest dalej, tym jest jaśniejszy, w szczególności punkty położone na płaszczyźnie tylnej ściany kostki standardowej (tj. płaszczyźnie $z = 1$) i za nią są białe. Obraz z lewej strony powstał dla światła umieszczonego w odległości nieskończonej (i scena jest przedstawiona w rzucie równoległym), a obraz z prawej odpowiada źródłu w skończonej odległości (i rzut jest perspektywiczny). Punkt $\boldsymbol{s}$ jest tu początkiem układu świata, $R = 2.2$, drugie źródło światła znajduje się w punkcie $(-6, 2, 8)$ (zatem $d \approx 10.2$, $\alpha \approx 0.2174$).



Rysunek 22.1. Zawartość bufora głębokości dla rzutów sceny określonych przez dwa źródła światła

Na listingu 22.1 są pokazane procedury realizujące opisane wyżej obliczenia. Zmieniona struktura `LSPar`, opisująca źródło światła, ma nowe pola `shadow_fbo` i `shadow_txt`,

Listing 22.1. Procedury konstrukcji macierzy V i P dla źródeł światła

C

```
typedef struct LSPar {
    GLfloat position[4];
    GLfloat ambient[3];
    GLfloat direct[3];
    GLfloat attenuation[3];
    GLuint  shadow_fbo, shadow_txt;
    GLfloat shadow_view[16], shadow_proj[16];
  } LSPar;

typedef struct LightBl {
    GLuint nls, mask, shmask;
    LSPar  ls[MAX_NLIGHTS];
  } LightBl;

LightBl light;

static void SetupShadowViewerTrans ( GLfloat org[3], GLfloat zv[3],
                                     GLfloat vmat[16] )
{
  GLfloat v[3], a[16], g, r, s;
  int     i, j;

  memcpy ( v, zv, 3*sizeof(GLfloat) );
  r = sqrt ( V3DotProductf ( v, v ) );
  v[2] += v[2] > 0.0 ? r : -r;
  g = 2.0/V3DotProductf ( v, v );
  M4x4Identf ( a );
  for ( i = 0; i < 2; i++ ) {
    s = v[i]*g;
    for ( j = 0; j < 3; j++ )
      a[4*i+j] -= s*v[j];
  }
  a[8] = zv[0]/r;  a[9] = zv[1]/r;  a[10] = zv[2]/r;
  memcpy ( &a[12], org, 3*sizeof(GLfloat) );
  M4x4InvertAffineIsometryf ( vmat, a );
} /*SetupShadowViewerTrans*/

void SetupShadowTransformations ( LightBl *light, int l, float sc[3],
                                  float R )
{
  GLfloat *lvm, *lpm;
  GLfloat lpos[3], v[3], d, n, s, t;
  int     i;

  if ( l < 0 || l >= MAX_NLIGHTS )
```

```
46:     return;
47:   if ( light->ls[l].shadow_txt ) {
48:     lvm = light->ls[l].shadow_view;
49:     lpm = light->ls[l].shadow_proj;
50:     if ( light->ls[l].position[3] != 0.0 ) {
51:             /* źródło światła w skończonej odległości */
52:       for ( i = 0; i < 3; i++ ) {
53:         lpos[i] = light->ls[l].position[i]/light->ls[l].position[3];
54:         v[i] = lpos[i] - sc[i];
55:       }
56:       SetupShadowViewerTrans ( lpos, v, lvm );
57:       d = V3DotProductf ( v, v );
58:       if ( d > 2.0*R*R ) {
59:         d = sqrt ( d );  n = d-R;
60:         s = R/d;                  /* sin(alpha) */
61:         t = s/sqrt ( 1.0-s*s );  /* tg(alpha) */
62:         M4x4Frustumf ( lpm, NULL, -n*t, n*t, -n*t, n*t, n, d+R );
63:       }
64:       else {  /* alpha == PI/4 */
65:         n = 0.4142*R;   /* sqrt(2)-1 ≅ 0.4142, sqrt(2)+1 ≅ 2.4143 */
66:         M4x4Frustumf ( lpm, NULL, -n, n, -n, n, n, 2.4143*R );
67:       }
68:     }
69:     else {  /* źródło światła w odległości nieskończonej */
70:       memcpy ( lpos, sc, 3*sizeof(GLfloat) );
71:       memcpy ( v, light->ls[l].position, 3*sizeof(GLfloat) );
72:       SetupShadowViewerTrans ( lpos, v, lvm );
73:       M4x4Orthof ( lpm, NULL, -R, R, -R, R, -R, R );
74:     }
75:   }
76: } /*SetupShadowTransformations*/
```

przechowujące identyfikatory bufora ramki i tekstury używanej jako bufor głębokości, w którym powstaje reprezentacja obszaru cienia, oraz tablice `shadow_view` i `shadow_proj`, w których będą przechowywane macierze V i P potrzebne w algorytmie cieni. Zmienna globalna `light` jest strukturą, w której mamy liczbę źródeł światła (w polu `nls`), maskę bitową opisującą źródła światła włączone w danej chwili i tablicę struktur `LSPar` opisujących źródła światła.

Dodatkowe pole `shmask` jest maską bitową, w której pamiętamy, dla których źródeł światła jest utworzona tekstura obszaru cienia — być może nie dla wszystkich źródeł światła zechcemy ją utworzyć, jako że to jest dosyć kosztowne (w tym sensie, że zajmuje dużo pamięci GPU) i możemy chcieć wyznaczać cienie tylko dla niektórych źródeł światła[9].

Pomocnicza procedura `SetupShadowViewerTrans` ma za zadanie skonstruować macierz V, której współczynniki mają trafić do tablicy `vmat`. Parametr `org` jest tablicą ze współ-

[9] Wbrew pozorom to miewa sens.

rzędnymi początku układu obserwatora (czyli środka kuli $\boldsymbol{s}$ albo punktu położenia źródła światła $\boldsymbol{z}$). Liczby w tablicy zv są współrzędnymi (w układzie świata) wektora $\boldsymbol{l}$ albo $\boldsymbol{z} - \boldsymbol{s}$, który ma kierunek i zwrot wersora osi z układu obserwatora (i który może mieć dowolną długość, byle nie 0). W liniach 23–25 jest konstruowany wektor normalny płaszczyzny odbicia $\boldsymbol{v}$. W linii 26 obliczamy czynnik γ, w linii 27 wpisujemy do tablicy a współczynniki macierzy jednostkowej, a w pętli w liniach 28–32 stosujemy odbicie do wersorów pierwszych dwóch osi układu świata — zauważmy, że iloczyn skalarny wektora $\boldsymbol{e}_i$ i wektora $\boldsymbol{v}$ jest i-tą współrzędną tego drugiego wektora. Zamiast odbijać wersor osi z (tj. wektor $\boldsymbol{e}_3$), a potem ewentualnie korygować zwrot otrzymanego wektora, w linii 33 dzielimy współrzędne wektora podanego w tablicy zv przez długość tego wektora i wpisujemy ilorazy do trzeciej kolumny konstruowanej macierzy V^{-1}. W linii 34 wpisujemy do czwartej kolumny współrzędne początku układu obserwatora. Wywołana w linii 35 procedura `M4x4InvertAffineIsometryf` (dodana chyłkiem do pliku `utilities.c`) znajduje odwrotność V macierzy V^{-1} sposobem opisanym w przypisie 8 na stronie 545.

Procedura `SetupShadowTransformations` otrzymuje jako parametry numer źródła światła, środek kuli otaczającej scenę i promień R tej kuli. Jeśli jest utworzona tekstura obszaru cienia związanego z l-tym źródłem światła, to procedura konstruuje odpowiednie macierze V_l i P_l (tj. V i P). Przypomnijmy, że jeśli wektor współrzędnych jednorodnych ma współrzędną wagową równą 0, to reprezentuje punkt niewłaściwy, czyli kierunek, w którym znajduje się źródło światła położone nieskończenie daleko. Zatem warunek sprawdzany w linii 50 jest spełniony przez źródła światła położone w odległości skończonej. Współrzędne kartezjańskie punktu $\boldsymbol{z}$ (położenia źródła światła) i współrzędne wektora $\boldsymbol{z} - \boldsymbol{s}$ są obliczane w pętli w liniach 52–55. Wynikiem wywołania procedury `SetupShadowViewerTrans` w linii 56 jest macierz V, którą zapamiętujemy w polu `shadow_view` struktury `LSPar`. W liniach 58–67 jest konstruowana macierz P, przy czym jeśli odległość źródła światła od punktu $\boldsymbol{s}$ jest większa niż $R\sqrt{2}$ (to sprawdzamy w linii 58), to obliczamy tangens takiego kąta α, aby cała kula się zmieściła na obrazie, a w przeciwnym razie przyjmujemy $\alpha = \frac{\pi}{4}$. Obliczenia w liniach 70–73, których wynikiem są macierze V i P dla światła dochodzącego z oddali, pozostawię bez komentarza.

22.2. Szadery

Do wykonywania obrazów z cieniami potrzebujemy *dwóch różnych* programów rysujących płaty Béziera, do wyznaczania reprezentacji obszaru cienia i do wykonywania końcowego obrazu[10]. W etapie wyznaczania reprezentacji obszaru cienia chcemy otrzymać tylko odpo-

[10] W zasadzie można użyć jednego programu (i na początku uruchamiania aplikacji tak zrobiłem). Ale: szader fragmentów programu dla końcowego obrazu wykonuje wiele niepotrzebnych w pierwszym etapie obliczeń. Można by je pominąć, umieszczając odpowiednie instrukcje w instrukcji warunkowej sterowanej za pomocą zmiennej jednolitej. Rzecz w tym, że blok wejściowy dla szadera fragmentów jest teraz bardzo długi, bo zawiera współrzędne cienia dla *wszystkich zadeklarowanych* źródeł światła (także tych wyłączonych w danej chwili), co oznacza ogromną ilość danych, które musiałyby być przetwarzane (interpolowane) w etapie rasteryzacji — a potem ignorowane. Szkoda na to czasu. Dlatego lepiej mieć dwa osobne programy, z których każdy wykonuje tylko potrzebne obliczenia i przekazuje między etapami tylko potrzebne dane.

wiednią zawartość bufora głębokości, zatem możemy i powinniśmy użyć w tym etapie maksymalnie uproszczonych szaderów — nie mają dla nas znaczenia kolory pikseli i w gruncie rzeczy nawet nie musimy tworzyć obrazu. Rysowanie odbędzie się w trybie pozaekranowym, tj. w teksturze będącej załącznikiem do bufora ramki utworzonego specjalnie w tym celu. Tekstura ta, którą nazywamy **teksturą obszaru cienia**, zostanie użyta jako bufor głębokości, a potem będzie udostępniona szaderowi fragmentów programu wykonującego końcowy obraz. Szader fragmentów wstępnego etapu nie musi zatem wykonywać żadnych obliczeń; może przypisać na wyjście dowolny kolor[11], który i tak będzie zignorowany, więc nawet tego nie musi robić. Natomiast szadery części przedniej (wierzchołków, rozdrabniania i geometrii) muszą spowodować wygenerowanie, w etapie rasteryzacji, fragmentów *tych samych* trójkątów otrzymanych podczas rozdrabniania płata, które będą rysowane na końcowym obrazie. Ale nie muszą one dostarczać żadnych danych potrzebnych tylko do obliczania kolorów pikseli, w tym wektora normalnego, współrzędnych cienia ani współrzędnych tekstury.

Szadery wierzchołków i sterowania rozdrabnianiem obu programów rysowania płatów Béziera są takie same — są to szadery pokazane na listingach 15.2 i 15.3. Szader rozdrabniania programu używanego do znajdowania reprezentacji obszaru cienia powstał przez uproszczenie szadera wykorzystywanego wcześniej. Na listingu 22.2 są umieszczone najważniejsze jego części. Szader ten oblicza tylko współrzędne leżącego na płacie wierzchołka (w układzie kostki standardowej) i nie ma żadnych dodatkowych danych wyjściowych poza strukturą `gl_PerVertex` (z polem `gl_Position`, któremu nadaje wartość).

Zmiany w procedurach `BCHorner2f`, `BPHorner2f`, `BCHorner4f` i `BPHorner4f` są podobne do tych wprowadzonych w pokazanych na listingu procedurach dla trójwymiarowych płatów wielomianowych. W szczególności zostały z nich usunięte wszystkie parametry i zmienne używane w obliczeniach pochodnych cząstkowych i wektorów normalnych oraz instrukcje wykonujące te obliczenia[12].

Listing 22.2. Uproszczony szader rozdrabniania

GLSL

```
#version 420 core
#define MAX_DEG 10
layout(quads,equal_spacing,ccw) in;
in TCInstance { .... } In[];
layout(std430,binding=0) buffer CPoints { .... } cp;
layout(std430,binding=1) buffer CPIndices { .... } cpi;
layout(std430,binding=2) buffer BezPatch { .... } bezp;
uniform TransBlock { .... } trb;
int inst;

```

[11] Na przykład dowolny kolor Forda T.

[12] W związku z uproszczeniem algorytmu wyznaczania punktu na krzywej Béziera (który już nie oblicza wektora pochodnej parametryzacji) oba szadery rozdrabniania płatów dostarczą takie same wyniki z dokładnością do błędów zaokrągleń. To są małe błędy, a ponadto dalsze przekształcenia prowadzące do różnego rzutowania podczas wyznaczania obszaru cienia i podczas wykonywania końcowego obrazu wprowadzą dodatkowe, różne błędy zaokrągleń. To nie szkodzi, te błędy zostaną skompensowane w sposób opisany dalej.

```
11: void BCHorner2f ( int n, vec2 bcp[MAX_DEG+1], float t, out vec2 p )
12: { .... } /*BCHorner2f*/
13: void BPHorner2f ( float u, float v, out vec4 pos )
14: { .... } /*BPHorner2f*/
15:
16: void BCHorner3f ( int n, vec3 bcp[MAX_DEG+1], float t, out vec3 p )
17: {
18:   int   i, b;
19:   float s, d;
20:   vec3  q;
21:
22:   s = 1.0-t;  d = t;  b = n;
23:   q = bcp[0];
24:   for ( i = 1; i <= n; i++ ) {
25:     q = s*q + (b*d)*bcp[i];
26:     d *= t;  b = (b*(n-i))/(i+1);
27:   }
28:   p = q;
29: } /*BCHorner3f*/
30:
31: void BPHorner3f ( float u, float v, out vec4 pos )
32: {
33:   vec3 p[MAX_DEG+1], q[MAX_DEG+1], r;
34:   int  i, j, k, l, i0;
35:   .... /* początek bez zmian */
36:   for ( i = k = 0;  i <= bezp.udeg;  i++ ) {
37:     .... /* wybieranie punktów kontrolnych z tablicy bez zmian */
38:     BCHorner3f ( bezp.vdeg, p, v, q[i] );
39:   }
40:   BCHorner3f ( bezp.udeg, q, u, r );
41:   pos = vec4 ( r, 1.0 );
42: } /*BPHorner3f*/
43:
44: void BCHorner4f ( int n, vec4 bcp[MAX_DEG+1], float t, out vec4 p )
45: { .... } /*BCHorner4f*/
46: void BPHorner4f ( float u, float v, out vec4 pos )
47: { .... } /*BPHorner4f*/
48:
49: void main ( void )
50: {
51:   vec4 pos;
52:
53:   inst = In[0].instance;
54:   pos = vec4 ( 0.0 );
55:   switch ( bezp.dim ) {
56: case 2: BPHorner2f ( gl_TessCoord.x, gl_TessCoord.y, pos );  break;
57: case 3: BPHorner3f ( gl_TessCoord.x, gl_TessCoord.y, pos );  break;
```

```
58: case 4: BPHorner4f ( gl_TessCoord.x, gl_TessCoord.y, pos );  break;
59:   }
60:   gl_Position = trb.vpm * (trb.mm * pos);
61: } /*main*/
```

W programie używanym do znalezienia obszaru cienia można zrezygnować z szadera geometrii, ponieważ w razie jego braku w programie trójkąty wygenerowane przez szader rozdrabniania trafiają od razu do etapu obcinania (zobacz rys. 1.2). Szader fragmentów może mieć pustą treść, ale jest potrzebny, aby odpowiednie informacje trafiły do bufora głębokości — jego obecność powoduje przystąpienie do pracy ostatniego etapu potoku przetwarzania grafiki, który wpisuje informacje do tego bufora.

Teraz opiszę zmiany szaderów rozdrabniania, geometrii i fragmentów używanych do wykonania końcowego obrazu płatów Béziera; one powstają przez *dodanie nowych* rzeczy do szaderów z poprzedniej wersji aplikacji. Na listingu 22.3 jest pokazany blok wyjściowy szadera rozdrabniania (który jest blokiem wejściowym szadera geometrii). Porównując go z blokiem na listingu 21.1, zauważamy nowe pole, tablicę `ShadowPos`; każdy element tej tablicy odpowiada jednemu źródłu światła. Szader rozdrabniania ma mu przypisać współrzędne jednorodne wierzchołka w opisanym niżej **układzie kostki jednostkowej** określonym dla tego źródła; nazwiemy je **współrzędnymi cienia**[13].

Listing 22.3. Szader rozdrabniania płatów Béziera dla końcowego obrazu

GLSL

```
1: #version 430 core
2:
3: #define MAX_NLIGHTS 8
4:
5: uniform TCInstance { .... } In[];
6:
7: out GVertex {
8:     int  instance;
9:     vec4 Colour;
10:    vec3 Position;
11:    vec3 pu, pv, Normal;
12:    vec2 PatchCoord, TexCoord;
13:    vec4 ShadowPos[MAX_NLIGHTS];
14:  } Out;
15:
16: layout(std430,binding=0) buffer CPoints { .... } cp;
17: layout(std430,binding=1) buffer CPIndices { .... } cpi;
18: layout(std430,binding=2) buffer BezPatch { .... } bezp;
19: uniform TransBlock { .... } trb;
20:
21: struct LSPar {
22:     vec4 position;
```

[13] Można uniknąć korzystania z tablicy współrzędnych cienia w sposób opisany w p. 26.4.3. Oba rozwiązania mają swoje wady i zalety.

```
23:     vec3 ambient;
24:     vec3 direct;
25:     vec3 attenuation;
26:     mat4 shadow_vpm;
27:   };
28:
29: uniform LSBlock {
30:     uint  nls;              /* liczba źródeł światła */
31:     uint  mask;             /* maska włączonych źródeł */
32:     uint  shmask;           /* maska tekstur cienia */
33:     LSPar ls[MAX_NLIGHTS];  /* poszczególne zródła światła */
34:   } light;
35:
36: .... /* procedury obliczające punkty i wektory normalne płatów Béziera */
37: .... /* jak w aplikacji 2F */
38:
39: void main ( void )
40: {
41:   vec4 pos, wpos, pu, pv, nv;
42:   uint l, mask;
43:
44:   .... /* początek bez zmian */
45:   Out.PatchCoord = gl_TessCoord.xy;
46:   wpos = trb.mm * pos;
47:   for ( l = 0, mask = 0x00000001;  l < light.nls;  l++, mask <<= 1 )
48:     if ( (light.mask & mask) != 0 )
49:       Out.ShadowPos[l] = light.ls[l].shadow_vpm * wpos;
50:   gl_Position = trb.vpm * wpos;
51:   Out.Position = wpos.xyz;
52:   .... /* dalsze instrukcje bez zmian */
53:   if ( !bezp.BezNormals || dot ( nv, nv ) < 1.0e-10 ) ....
54:   ....
55: } /*main*/
```

Szader rozdrabniania musi mieć teraz dostęp do bloku zmiennych jednolitych opisujących źródła światła (tego, na którego podstawie szader fragmentów oblicza oświetlenie), ponieważ odwołuje się do nowego pola `shadow_vpm` struktury opisującej źródła światła. W polu tym jest podana macierz będąca iloczynem trzech macierzy, BP_lV_l. Macierze P_l i V_l opisują przejścia od układu świata do układu obserwatora związanego z l-tym źródłem światła i dalej do układu kostki standardowej. Natomiast macierz

$$B = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

taka sama dla wszystkich źródeł światła, opisuje przekształcenie, które kostkę *standardową* $[-1,1]^3$ odwzorowuje na kostkę *jednostkową* $[0,1]^3$. Przekształcenie to jest potrzebne dlatego, że ewaluatory tekstur (zgodnie z ogólną konwencją przyjętą w OpenGL-u) spodziewają się

Listing 22.4. Szader geometrii dla obrazu płatów Béziera z cieniami

GLSL

```
#version 430

layout(triangles) in;
layout(triangle_strip,max_vertices=3) out;
in GVertex { .... } In[];
out FVertex { .... } Out;
struct LSPar { .... };
uniform LSBlock { .... } light;

void main ( void )
{
  uint i, l, mask;
  vec3 v1, v2, nv;

  v1 = In[1].Position - In[0].Position;
  v2 = In[2].Position - In[0].Position;
  nv = normalize ( cross ( v2, v1 ) );
  for ( i = 0; i < 3; i++ ) {
    gl_Position = gl_in[i].gl_Position;
    Out.Position = In[i].Position;
    if ( dot ( In[i].Normal, In[i].Normal ) < 1.0e-10 ) {
      Out.pu = In[i].pu - dot ( In[i].pu, nv )*nv;
      Out.pv = In[i].pv - dot ( In[i].pv, nv )*nv;
      Out.Normal = nv;
    }
    else {
      Out.pu = In[i].pu;
      Out.pv = In[i].pv;
      Out.Normal = In[i].Normal;
    }
    Out.Colour = In[i].Colour;
    Out.PatchCoord = In[i].PatchCoord;
    Out.TexCoord = In[i].TexCoord;
    Out.instance = In[i].instance;
    for ( l = 0, mask = 0x00000001;  l < light.nls;  l++, mask <<= 1 )
      if ( (light.mask & mask) != 0 )
        Out.ShadowPos[l] = In[i].ShadowPos[l];
    EmitVertex ();
  }
  EndPrimitive ();
} /*main*/
```

współrzędnych tekstury z przedziału $[0,1]$. Instrukcja w linii 46 oblicza współrzędne wierzchołka w układzie świata (tj. dokonuje przejścia od układu modelu do układu świata). W liniach 47–49 obliczane są współrzędne cienia wierzchołka dla poszczególnych (włączonych) źródeł światła.

Nowy szader geometrii jest pokazany na listingu 22.4. Przypomnijmy, że jego zadaniem jest skopiowanie danych z wejścia na wyjście i ewentualne obliczenie (i wyprowadzenie) wektora normalnego płaszczyzny trójkąta, jeśli szader rozdrabniania przekazał zerowy wektor normalny, bo płat Béziera ma osobliwość lub aplikacja nadała polu `BezNormals` bloku `BezPatch` wartość `false`.

Bloki wejściowy `GVertex` i wyjściowy `FVertex` mają identyczne pola jak blok wyjściowy `GVertex` na listingu 22.3 (prawie, bo pole `FVertex.instance` ma kwalifikator `flat`). Szader geometrii odwołuje się też do bloków `BezPatch` i `LSBlock`, których deklaracje zostały dodane do jego treści. W dodatku do dotychczasowych zadań, w pętli w liniach 35–37 szader kopiuje współrzędne cienia wierzchołka dla włączonych źródeł światła z bloku wejściowego do wyjściowego.

Etap rasteryzacji dokonuje interpolacji współrzędnych wierzchołków (w układzie kostki standardowej), a także wszystkich danych dodatkowych obecnych w bloku wyjściowym szadera geometrii — współrzędnych koloru, współrzędnych położenia w układzie świata, pochodnych cząstkowych, wektora normalnego, parametrów płata, współrzędnych tekstury i współrzędnych cienia dla *wszystkich* (nie tylko włączonych) źródeł światła. Zobaczmy teraz, jak z tej informacji korzysta szader fragmentów dla końcowego obrazu (listing 22.5).

W liniach 4–10 są wymienione bloki interfejsu (tj. wejścia/wyjścia i zmiennych jednolitych) szadera, przy czym bloki o niezmienionej budowie (w porównaniu z szaderem aplikacji drugiej F) przerobiłem na szaro. W liniach 12 i 13 są zadeklarowane zmienne jednolite (ewaluatory tekstury), do których zostanie przywiązana tekstura koloru farby (opisująca obrazki na czajniku) i tekstury obszarów cienia dla poszczególnych źródeł światła. Zwróćmy uwagę na podane (w kwalifikatorach `layout`) numery punktów dowiązania tekstur — dla zmiennej `tex` jest to numer 0 (w OpenGL-u identyfikowany przez stałą symboliczną `GL_TEXTURE0`), a dla elementów tablicy `shtex` to są numery od 2 do 9, jako że ustalona w aplikacji maksymalna liczba źródeł światła to 8. Punktu dowiązania o numerze 1 użyjemy dla tekstury nakładanej na lustro — w tej aplikacji nie może to być numer 0 (tzn. ten sam numer, którego używamy dla tekstury nałożonej na czajnik), bo całość nie zadziałałaby poprawnie. Typ

Listing 22.5. Fragmenty szadera fragmentów dla obrazów z cieniami

GLSL

```
1: #define MAX_NLIGHTS    8
2: #define MAX_MATERIALS 10
3:
4: in FVertex { .... } In;
5: out vec4 out_Colour;
6: uniform TransBlock { .... } trb;
7: struct LSPar { .... };
8: uniform LSBlock { .... } light;
9: struct Material { .... };
```

```
10: uniform MatBlock { .... } mat;
11:
12: layout(binding=0) uniform sampler2D tex;
13: layout(binding=2) uniform sampler2DShadow shtex[MAX_NLIGHTS];
14: Material mm;
15:
16: vec3 MatEmission ( float cosnv ) { .... }
17:
18: float IsEnlighted ( uint l )
19: {
20:   return textureProj ( shtex[l], In.ShadowPos[l] );
21: } /*IsEnlighted*/
22:
23: vec3 LambertLighting ( void )
24: {
25:   vec3  lv, vv, Colour;
26:   float d, e, s, dist;
27:   uint  i, mask;
28:
29:   vv = posDifference ( trb.eyepos, In.Position, dist );
30:   e = dot ( vv, tnormal );
31:   Colour = MatEmission ( dot ( normal, vv ) );
32:   for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
33:     if ( (light.mask & mask) != 0 ) {
34:       Colour += light.ls[i].ambient * mm.ambref;
35:       s = ((light.shmask & mask) != 0) ? IsEnlighted ( i ) : 1.0;
36:       if ( s > 0.0 ) {
37:         lv = posDifference ( light.ls[i].position, In.Position, dist );
38:         d = dot ( lv, normal );
39:         if ( e > 0.0 ) {
40:           if ( d > 0.0 ) {
41:             if ( light.ls[i].position.w != 0.0 )
42:               d *= attFactor ( light.ls[i].attenuation, dist );
43:             Colour += (s*d * light.ls[i].direct) * mm.dirref;
44:           }
45:         }
46:         else {
47:           if ( d < 0.0 ) {
48:             if ( light.ls[i].position.w != 0.0 )
49:               d *= attFactor ( light.ls[i].attenuation, dist );
50:             Colour -= (s*d * light.ls[i].direct) * mm.dirref;
51:           }
52:         }
53:       }
54:     }
55:   return clamp ( Colour, 0.0, 1.0 );
56: } /*LambertLighting*/
```

ewaluatora `sampler2DShadow` jest przeznaczony specjalnie do badania (w teksturze użytej jako bufor głębokości), czy dany punkt leży w obszarze cienia.

Funkcja `IsEnlighted` w liniach 18–21 dokonuje badania, czy dany punkt należy do obszaru cienia. Parametr tej funkcji jest numerem źródła światła. Obliczenie wygląda tak: pierwszym parametrem funkcji wbudowanej `textureProj` jest odpowiednia tekstura, a drugi parametr to wektor (jednorodnych) współrzędnych cienia przetwarzanego punktu. Na jego podstawie są obliczane współrzędne kartezjańskie x, y, z (przez podzielenie pierwszych trzech współrzędnych jednorodnych przez czwartą), które mają być liczbami z przedziału $[0,1]$. Również w tekselach są przechowywane liczby z tego przedziału, przy czym 0 odpowiada punktowi na przedniej ścianie kostki, a 1 punktowi na tylnej ścianie. Funkcja `textureProj` (dla ewaluatora, którego parametrom aplikacja nadała odpowiednie wartości, opis będzie dalej) porównuje współrzędną z z wartością tekstury[14] w punkcie (x, y) i przekazuje wartość 1, jeśli współrzędna z jest mniejsza, oraz 0, jeśli jest większa niż wartość tekstury w tym punkcie.

W liniach 23–56 jest pokazana zmodyfikowana procedura realizująca model oświetlenia Lamberta; analogiczne zmiany trzeba też wprowadzić w procedurze z modelem Blinna-Phonga. W linii 34 do światła odbijanego przez piksel jest dodawany składnik odpowiadający światłu pochodzącemu z danego źródła, rozproszonego w otoczeniu (czyli oświetlającego punkty także w obszarze cienia). W linii 35 szader bada, czy dla danego źródła światła tekstura obszaru cienia została utworzona; jeśli tak, to zmiennej `s` przypisywana jest wartość funkcji `IsEnlighted`, a jeśli nie, to wartość 1 — wtedy z założenia punkt jest bezpośrednio oświetlony. W liniach 43 i 50 czynnik s jest uwzględniony w składniku opisującym światło dochodzące bezpośrednio od źródła. Dlaczego mnożymy wyrażenie obliczone w liniach 38, 42 i 49 przez zmienną `s`, która ma wartość 1? Bo później zmienimy funkcję `IsEnlighted` (która będzie mogła przyjmować wartości ułamkowe), aby poprawić wygląd cieni.

Dwóch programów szaderów potrzebujemy także do rysowania lustra. Program używany do znajdowania obszaru cienia nie powinien nakładać na lustro tekstury reprezentującej obraz odbitej w lustrze sceny. Program rysujący lustro na końcowym obrazie w zasadzie nie powinien nakładać cienia na odbity obraz, ale można rozważyć cienie na odwrotnej stronie lustra, jeśli chcemy rysować bardziej skomplikowane sceny[15]. Natomiast darowałem sobie rysowanie cieni na odcinkach siatek kontrolnych płatów. I już.

22.3. Przygotowanie programów szaderów

Listing 22.6 przedstawia procedurę, która kompiluje, łączy i przygotowuje do pracy programy szaderów przeznaczone do rysowania obiektów sceny — czajnika i torusa. Trzy programy, których identyfikatory są przechowane w tablicy `program_id` struktury typu `BPRenderPrograms`, służą odpowiednio do rysowania płatów w celu znalezienia reprezentacji obszaru cienia, płatów oświetlonych i pokrytych teksturą na końcowym obrazie i siatek kontrolnych

[14] Pamiętajmy: tekstura jest funkcją, tablica tekseli jest tylko reprezentacją tej funkcji.

[15] Można rozważyć światło odbite w lustrze, oświetlające obiekty i narysować cienie dla takiego światła. Napisanie takiego programu to już spore wyzwanie.

Listing 22.6. Procedura kompilacji szaderów dla płatów Béziera

C

```
typedef struct {
    GLuint progid[3];
    GLint  ColourSourceLoc, colour_source,
           LightingModelLoc, lighting_model,
           NormalSourceLoc, normal_source,
           ModifyDepthLoc, modify_depth;
  } BPRenderPrograms;

void LoadBPShaders ( BPRenderPrograms *brprog )
{
  static const char *filename[] =
    { "app2g0.vert.glsl", "app2g0.tesc.glsl", "app2g0.tese.glsl",
      "app2g0.geom.glsl", "app2g0.frag.glsl", "app2g1.vert.glsl",
      "app2g1.frag.glsl", "app2g3.tese.glsl", "app2g3.frag.glsl" };
  static const GLuint shtype[] =
    { GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER,
      GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER, GL_VERTEX_SHADER,
      GL_FRAGMENT_SHADER, GL_TESS_EVALUATION_SHADER, GL_FRAGMENT_SHADER };
  static const GLchar *UVarNames[] =
    { "ColourSource", "LightingModel", "NormalSource", "ModifyDepth" };
  GLuint shader_id[9], shid[5];
  int    i;

  for ( i = 0; i < 9; i++ )
    shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
  shid[0] = shader_id[0];  shid[1] = shader_id[1];
  shid[2] = shader_id[7];  shid[3] = shader_id[8];
  brprog->program_id[0] = LinkShaderProgram ( 4, shid, "0" );
  brprog->program_id[1] = LinkShaderProgram ( 5, shader_id, "1" );
  brprog->program_id[2] = LinkShaderProgram ( 2, &shader_id[5], "2" );
  GetAccessToTransBlockUniform ( brprog->program_id[1] );
  GetAccessToLightMatUniformBlocks ( brprog->program_id[1] );
  GetAccessToBezPatchStorageBlocks ( brprog->program_id[1], true, false );
  brprog->ColourSourceLoc =
      glGetUniformLocation ( brprog->program_id[1], UVarNames[0] );
  brprog->LightingModelLoc =
      glGetUniformLocation ( brprog->program_id[1], UVarNames[1] );
  brprog->NormalSourceLoc =
      glGetUniformLocation ( brprog->program_id[1], UVarNames[2] );
  brprog->ModifyDepthLoc =
      glGetUniformLocation ( brprog->program_id[1], UVarNames[3] );
  AttachUniformTransBlockToBP ( brprog->program_id[0] );
  AttachUniformTransBlockToBP ( brprog->program_id[2] );
  for ( i = 0; i < 9; i++ )
    glDeleteShader ( shader_id[i] );
```

```
46:   ExitIfGLError ( "LoadBPShaders" );
47: } /*LoadBPShaders*/
```

w obu etapach rysowania. Pozostałe pola tej struktury przechowują położenia zmiennych jednolitych w drugim programie.

Szader fragmentów zapisany w pliku `app2g3.frag.glsl`, którego procedura `main` nie wykonuje żadnej instrukcji, jest częścią programów używanych do znajdowania cieni rzucanych przez płaty Béziera i przez lustro. Szader ten jest kompilowany także przez procedurę kompilacji programu do rysowania lustra (listing 22.7), co niezauważalnie wydłuża czas przygotowania aplikacji do podjęcia interakcji z użytkownikiem, ale znacznie upraszcza procedury kompilacji szaderów dla płatów Béziera i lustra, czyniąc je mniej zależnymi od siebie. Ponieważ punkt dowiązania bloku zmiennych jednolitych `TransBlock` jest ustalany przez procedurę `LoadBPShaders`, a procedura `LoadMirrorShaders` wiąże ten punkt z programami rysowania lustra, określa to właściwą kolejność wywoływania tych procedur podczas inicjalizacji danych aplikacji.

Listing 22.7. Procedura kompilacji szaderów dla lustra

C

```
1: void LoadMirrorShaders ( GLuint program_id[2] )
2: {
3:   static const char *filename[] =
4:     { "app2g4.vert.glsl", "app2g3.frag.glsl",
5:       "app2g2.vert.glsl", "app2g2.frag.glsl" };
6:   static const GLuint shtype[] =
7:     { GL_VERTEX_SHADER, GL_FRAGMENT_SHADER,
8:       GL_VERTEX_SHADER, GL_FRAGMENT_SHADER };
9:   GLuint shader_id[4];
10:  int    i;
11:
12:  for ( i = 0; i < 4; i++ )
13:    shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
14:  program_id[0] = LinkShaderProgram ( 2, shader_id, "3" );
15:  program_id[1] = LinkShaderProgram ( 2, &shader_id[2], "4" );
16:  AttachUniformTransBlockToBP ( program_id[0] );
17:  AttachUniformTransBlockToBP ( program_id[1] );
18:  for ( i = 0; i < 4; i++ )
19:    glDeleteShader ( shader_id[i] );
20:  ExitIfGLError ( "LoadMirrorShaders" );
21: } /*LoadMirrorShaders*/
```

22.4. Tworzenie buforów ramki i tekstur dla obszarów cienia

Na listingu 22.8 są pokazane procedury, które tworzą, przygotowują do pracy i sprzątają bufory ramki i będące ich załącznikami tekstury obszarów cienia dla poszczególnych źródeł światła. Procedura `ConstructShadowTxtFBO` powinna być wywołana dla każdego źródła

Listing 22.8. Procedury obsługi tekstur dla obszarów cienia

C

```
1: void ConstructShadowTxtFBO ( LightBl *light, int l )
2: {
3:   GLuint fbo, txt;
4:
5:   if ( l < 0 || l >= MAX_NLIGHTS )
6:     return;
7:   glGenTextures ( 1, &txt );
8:   glGenFramebuffers ( 1, &fbo );
9:   if ( (light->ls[l].shadow_txt = txt) &&
10:       (light->ls[l].shadow_fbo = fbo) ) {
11:     glBindFramebuffer ( GL_FRAMEBUFFER, fbo );
12:     glActiveTexture ( GL_TEXTURE2+l );
13:     glBindTexture ( GL_TEXTURE_2D, txt );
14:     glBindBuffer ( GL_UNIFORM_BUFFER, light->lsbuf );
15:     light->shmask |= 0x01 << l;
16:     glBufferSubData ( GL_UNIFORM_BUFFER, lsbofs[2], sizeof(GLuint),
17:                       &light->shmask );
18:     glTexStorage2D ( GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32,
19:                      SHADOW_MAP_SIZE, SHADOW_MAP_SIZE );
20:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
21:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
22:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
23:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
24:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
25:                       GL_COMPARE_REF_TO_TEXTURE );
26:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL );
27:     glBindTexture ( GL_TEXTURE_2D, 0 );
28:     glFramebufferTexture ( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
29:                            light->ls[l].shadow_txt, 0 );
30:     glDrawBuffer ( GL_NONE );
31:     if ( glCheckFramebufferStatus ( GL_FRAMEBUFFER ) !=
32:               GL_FRAMEBUFFER_COMPLETE )
33:       ExitOnError ( "ConstructShadowTxtFBO" );
34:     glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
35:     ExitIfGLError ( "ConstructShadowTxtFBO" );
36:   }
37: } /*ConstructShadowTxtFBO*/
38:
39: void UpdateShadowMatrix ( LightBl *light, int l )
40: {
41:   int                   ofs;
42:   GLfloat               lvpm[16], a[16];
43:   const GLfloat b[16] = {0.5,0.0,0.0,0.0,0.0,0.5,0.0,0.0,
44:                          0.0,0.0,0.5,0.0,0.5,0.5,0.5,1.0};
45:
```

```
46:   M4x4Multf ( a, light->ls[l].shadow_proj, light->ls[l].shadow_view );
47:   M4x4Multf ( lvpm, b, a );
48:   ofs = l*(lsbofs[8]-lsbofs[3]) + lsbofs[7];
49:   glBindBuffer ( GL_UNIFORM_BUFFER, light->lsbuf );
50:   glBufferSubData ( GL_UNIFORM_BUFFER, ofs, 16*sizeof(GLfloat), lvpm );
51:   ExitIfGLError ( "UpdateShadowMatrix" );
52: } /*UpdateShadowMatrix*/
53:
54: void BindShadowTxtFBO ( TransBl *trans, LightBl *light, int l )
55: {
56:   if ( l < 0 || l >= MAX_NLIGHTS )
57:     return;
58:   if ( (light->ls[l].shadow_txt == 0) )
59:     return;
60:   LoadShTrans ( trans, light->ls[l].shadow_view, ight->ls[l].shadow_proj,
61:                 light->ls[l].position );
62:   glBindFramebuffer ( GL_FRAMEBUFFER, light->ls[l].shadow_fbo );
63:   ExitIfGLError ( "BindShadowTxtFBO" );
64: } /*BindShadowTxtFBO*/
65:
66: void DeleteShadowFBO ( LightBl *light )
67: {
68:   int l;
69:
70:   glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
71:   for ( l = 0; l < MAX_NLIGHTS; l++ ) {
72:     if ( light->ls[l].shadow_fbo != 0 )
73:       glDeleteFramebuffers ( 1, &light->ls[l].shadow_fbo );
74:     if ( light->ls[l].shadow_txt != 0 )
75:       glDeleteTextures ( 1, &light->ls[l].shadow_txt );
76:   }
77:   ExitIfGLError ( "DeleteShadowFBO" );
78: } /*DeleteShadowFBO*/
```

światła, dla którego aplikacja ma wyznaczać cienie. Instrukcje w liniach 7 i 8 rezerwują identyfikatory tekstury i bufora ramki. Jeśli to się powiedzie, co jest sprawdzane w liniach 9 i 10 z jednoczesnym zapamiętaniem tych identyfikatorów w odpowiednich polach struktury opisującej l-te źródło światła, to wykonywany jest ciąg instrukcji nadających potrzebne własności tym obiektom.

W linii 12 uaktywniamy punkt dowiązania tekstury o numerze $l+2$, a w linii 13 przywiązujemy nową teksturę do celu `GL_TEXTURE_2D` w tym punkcie. W linii 18 następuje określenie rodzaju danych przechowywanych w tekselach oraz szerokości i wysokości tekstury (makro `SHADOW_MAP_SIZE` rozwija się do stałej 1024, można je zmieniać), a ponadto jest rezerwowany blok pamięci GPU na tablicę tekseli. Trzeci parametr (`internalFormat`) o wartości `GL_DEPTH_COMPONENT32` określa 32 bity na teksel.

Wartości GL_CLAMP_TO_EDGE nadane parametrom sterującym obcinaniem w liniach 20 i 21 oznaczają, że współrzędne x, y punktu w układzie kostki jednostkowej, jeśli nie należą do przedziału $[0,1]$, to mają być zastąpione przez 0 albo 1.

Sposób filtrowania tekstury określony przez parametry procedury glTexParameteri w liniach 22 i 23 polega na interpolacji liniowej tekseli.

Dla badania, czy punkt jest w obszarze cienia, istotne są parametry podane w liniach 24–26; dla tekstury głębokości wartość GL_COMPARE_REF_TO_TEXTURE parametru GL_TEXTURE_COMPARE_MODE oznacza, że wbudowana w GLSL funkcja texture (lub textureProj) ma dokonać porównania współrzędnej z punktu w kostce jednostkowej z wartością tekstury i podać wynik tego porównania. Funkcja porównująca GL_LEQUAL oznacza, że funkcja ma wartość 1, jeśli współrzędna z punktu jest mniejsza lub równa wartości tekstury, co oznacza, że punkt jest bezpośrednio oświetlony. Po określeniu wszystkich potrzebnych parametrów teksturę odczepiamy od celu, w linii 27.

Jednocześnie z teksturą obszaru cienia przygotowujemy do pracy bufor ramki. W linii 11 jest on przyczepiany do celów GL_DRAW_FRAMEBUFFER i GL_READ_FRAMEBUFFER (jednocześnie do obu; to nie szkodzi). W liniach 28–29 dodajemy do niego załącznik — świeżo utworzoną teksturę, która będzie buforem głębokości. Nie zawracamy sobie (ani OpenGL-owi) głowy teksturą dla obrazu, który jest niepotrzebny, tylko wywołujemy procedurę glDrawBuffer z parametrem GL_NONE. W linii 15 ustawiamy bit maski, aby zaznaczyć, że dla l-tego źródła światła tekstura obszaru cienia została utworzona i ma być użyta przez szader, po czym przesyłamy tę maskę do bloku zmiennych jednolitych zawierającego opisy źródeł światła. W linii 34 odczepiamy bufor ramki od obu celów, a gdy przyjdzie właściwa pora, znów go przyczepimy.

Procedura UpdateShadowMatrix oblicza iloczyn macierzy BP_lV_l, który opisuje przejście od układu świata do układu kostki jednostkowej, a następnie przesyła ten iloczyn do pola shadow_vpm w strukturze LSPar opisującej l-te źródło światła w pamięci GPU. Współczynniki macierzy B są podane w tablicy b, a współczynniki macierzy V_l i P_l są brane z opisu źródła światła w pamięci CPU, z miejsca, w którym zostały zapamiętane przez wywołaną wcześniej procedurę SetupShadowTransformations z listingu 22.1. Przesunięcie obliczone w linii 48 wyznacza położenie pola shadow_vpm l-tego elementu tablicy ls w bloku zmiennych jednolitych LSBlock.

Procedura BindShadowTxtFBO, za pomocą procedury LoadShTrans (listing 22.9), przesyła macierze V_l i P_l oraz ich iloczyn i (niepotrzebne tu) położenie obserwatora (tj. źródła światła) do bloku zmiennych jednolitych TransBlock, a następnie przwiązuje odpowiedni bufor ramki do celu GL_DRAW_FRAMEBUFFER i już można rysować scenę (przy użyciu uproszczonych szaderów) w celu otrzymania reprezentacji obszaru cienia dla l-tego źródła światła.

Procedura DeleteShadowFBO przegląda tablicę opisów źródeł światła i likwiduje tekstury obszarów cienia i bufory ramki, których te tekstury były załącznikami. Wypada ją wywołać podczas końcowego sprzątania.

Procedura LoadShTrans jest umieszczona w pliku źródłowym trans.c, który jest opakowaniem wszystkich procedur przesyłających dane do bloku zmiennych jednolitych TransBlock; w szczególności tylko w tym pliku jest widoczna tablica trbofs, w której są zapa-

miętane przesunięcia pól w tym bloku. Procedura przesyła do bloku `TransBlock` macierze przekształceń przekazane bezpośrednio jako parametry.

Listing 22.9. Procedura `LoadShTrans`

```C
void LoadShTrans ( TransBl *trans, GLfloat vm[16], GLfloat pm[16],
                   GLfloat eyepos[4] )
{
  GLfloat vpm[16];

  M4x4Multf ( vpm, pm, vm );
  glBindBuffer ( GL_UNIFORM_BUFFER, trans->trbuf );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[2], 16*sizeof(GLfloat), vm );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[3], 16*sizeof(GLfloat), pm );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[4], 16*sizeof(GLfloat), vpm );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[5], 4*sizeof(GLfloat),
                    eyepos );
  ExitIfGLError ( "LoadShTrans" );
} /*LoadShTrans*/
```

22.5. Zmiany w aplikacji

Instrukcje procedury `GetAccessToLightMatUniformBlocks`, pokazanej na listingu 18.6 nie wymagają żadnych zmian; trzeba tylko do tablicy `ULSNames` dodać nazwy dwóch nowych pól w bloku zmiennych jednolitych `LSBlock` opisującym źródła światła i wydłużyć (przez zmianę makra `NLSOFS`) tablicę `lsbofs`, tak jak na listingu 22.10, oraz odpowiednio zmienić indeksy do tej tablicy w instrukcjach przesyłających dane do bufora zawierającego ten blok.

Listing 22.10. Zmieniona tablica nazw pól w bloku `LSBlock`

```C
#define NLSOFFS 9

const GLchar *ULSNames[] =
  { "LSBlock", "LSBlock.nls", "LSBlock.mask", "LSBlock.shmask",
    "LSBlock.ls[0].ambient", "LSBlock.ls[0].direct",
    "LSBlock.ls[0].position", "LSBlock.ls[0].attenuation",
    "LSBlock.ls[0].shadow_vpm", "LSBlock.ls[1].ambient" };
```

Listing 22.11 przedstawia zmiany struktury `AppData`, zawierającej wszystkie dane części graficznej aplikacji. Jest tu nowe pole `shadows`, którego wartość `true` oznacza, że podczas rysowania należy uwzględnić cienie, a `false` powoduje rysowanie bez cieni.

W procedurze `InitMyWorld` (listing 22.12) trzeba tylko usunąć operator `&` z wywołania procedury kompilującej szadery do rysowania lustra, bo teraz parametr tej procedury jest

tablicą, i nadać początkową wartość polu `shadows`, aby domyślnie aplikacja wykonywała obrazy z cieniami.

Listing 22.11. Opakowanie danych aplikacji

C

```
typedef struct {
    Camera           camera;
    BezierPatchObjf  *myteapot, *mytorus;
    Mirror           mirror;
    TransBl          trans;
    LightBl          light;
    MatBl            mat;
    GLuint           mytexture;
    GLint            BezNormals, TessLevel;
    char             cnet, skeleton, animate, shadows;
    float            model_rot_axis[3];
    double           teapot_rot_angle, torus_rot_angle;
    GLfloat          teapot_mmatrix[16], torus_mmatrix[16];
    BPRenderPrograms brprog;
    GLuint           miprog[2];
  } AppData;
```

Listing 22.12. Procedura `InitMyWorld`

C

```
void InitMyWorld ( int argc, char *argv[], int width, int height )
{
  float axis[4] = {0.0,0.0,1.0};

  memset ( &appdata, 0, sizeof(AppData) );
  LoadBPShaders ( &appdata.brprog );
  LoadMirrorShaders ( appdata.miprog );
  appdata.trans.trbuf = NewUniformTransBlock ();
  .... /* instrukcje bez zmian */
  appdata.cnet = appdata.skeleton = appdata.animate = false;
  appdata.shadows = true;
  ConstructMyTeapot ( &appdata );
  ConstructMyTorus ( &appdata );
  ConstructMirror ( &appdata.mirror );
  InitLights ( &appdata );
  appdata.mytexture = LoadMyTextures ();
} /*InitMyWorld*/
```

Nowa procedura `InitLights` pokazana na listingu 22.13 (porównaj z listingiem 15.16) po przesłaniu parametrów opisujących źródło światła o numerze 0 i „włączeniu” go tworzy bufor ramki i teksturę cienia oraz konstruuje macierze przekształceń dla rysowania sceny widzianej z kierunku padania światła i przesyła iloczyn macierzy BP_0V_0 do pamięci GPU.

Dodając kolejne źródła światła wokół sceny, należałoby zrobić dla nich to samo. Jeśli położenia źródeł światła miałyby być animowane, to po każdej zmianie położenia źródła światła trzeba wywołać procedury `SetupShadowTxtTransformations` i `UpdateShadowMatrix`.

Listing 22.13. Procedura `InitLights`

C

```
1: void InitLights ( AppData *ad )
2: {
3:   GLfloat amb0[3] = { 0.2, 0.2, 0.3 };
4:   GLfloat dir0[3] = { 0.8, 0.8, 0.8 };
5:   GLfloat pos0[4] = { -0.2, 1.0, 1.0, 0.0 };
6:   GLfloat atn0[3] = { 1.0, 0.0, 0.0 };
7:   GLfloat csc[3]  = { 0.0, 0.0, 0.0 };
8:
9:   SetLightAmbient ( &ad->light, 0, amb0 );
10:  SetLightDirect ( &ad->light, 0, dir0 );
11:  SetLightPosition ( &ad->light, 0, pos0 );
12:  SetLightAttenuation ( &ad->light, 0, atn0 );
13:  SetLightOnOff ( &ad->light, 0, 1 );
14:  ConstructShadowTxtFBO ( &ad->light, 0 );
15:  SetupShadowTxtTransformations ( &ad->light, 0, csc, 2.2 );
16:  UpdateShadowMatrix ( &ad->light, 0 );
17: } /*InitLights*/
```

Na listingu 22.14 jest pokazana procedura rysowania czajnika; podobne (choć prostsze, bo torus nie ma nakładanej tekstury) zmiany trzeba wprowadzić w procedurze `DrawMyTorus`. Jeśli parametr `final` ma wartość `true`, to ma być wykonany obraz końcowy; wtedy (w liniach 9–16) nadajemy wartości zmiennym jednolitym programu szderów, wybieramy opis materiału i (jeśli trzeba) przyczepiamy teksturę, która ma być nałożona na czajnik, do celu `GL_TEXTURE_2D` w punkcie dowiązania tekstury `GL_TEXTURE0`. Jeśli parametr `final` ma wartość `false`, to rysujemy czajnik bez tych czynności wstępnych, przy użyciu programu uproszczonego, który nie oblicza kolorów pikseli.

Listing 22.14. Procedura `DrawMyTeapot`

C

```
1: void DrawMyTeapot ( AppData *ad, char final )
2: {
3:   if ( ad->skeleton )
4:     glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
5:   else
6:     glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
7:   if ( final ) {
8:     glUseProgram ( ad->brprog.program_id[1] );
9:     glUniform1i ( ad->brprog.ColourSourceLoc, ad->brprog.colour_source );
10:    glUniform1i ( ad->brprog.NormalSourceLoc, ad->brprog.normal_source );
11:    glUniform1i ( ad->brprog.ModifyDepthLoc, ad->brprog.modify_depth );
12:    ChooseMaterial ( &ad->mat, 0 );
```

```
13:     if ( ad->colour_source == 2 ) {
14:       BindBezPatchTextureBuffer ( ad->myteapot );
15:       glActiveTexture ( GL_TEXTURE0 );
16:       glBindTexture ( GL_TEXTURE_2D, ad->mytexture );
17:       DrawBezierPatches ( ad->myteapot );
18:       glBindTexture ( GL_TEXTURE_2D, 0 );
19:       return;
20:     }
21:   }
22:   else
23:     glUseProgram ( ad->brprog.program_id[0] );
24:   DrawBezierPatches ( ad->myteapot );
25: } /*DrawMyTeapot*/
```

Listing 22.15 przedstawia nową procedurę rysowania lustra, która w celu znalezienia obszaru cienia rzucanego przez lustro lub programu dla obrazu końcowego z nałożoną na lustro teksturą przedstawiającą obraz odbity używa programu uproszczonego. Wartość parametru `final` wpływa na przywiązanie tekstury, która podczas znajdowania cienia jest niepotrzebna.

Listing 22.15. Procedura rysowania lustra

C

```
1: void DrawMirror ( Mirror *mirror, GLuint program_id, char final )
2: {
3:   glUseProgram ( program_id );
4:   glBindVertexArray ( mirror->mirror_vao );
5:   if ( final ) {
6:     glActiveTexture ( GL_TEXTURE1 );
7:     glBindTexture ( GL_TEXTURE_2D, mirror->mirror_txt[0] );
8:   }
9:   glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
10:  glDrawArrays ( GL_TRIANGLE_FAN, 0, 4 );
11:  glBindVertexArray ( 0 );
12:  ExitIfGLError ( "DrawMirror" );
13: } /*DrawMirror*/
```

Zobaczmy wreszcie procedury rysowania całej sceny; mamy je na listingu 22.16 (porównaj z listingiem 20.13). Procedura `DrawScene` ma dodatkowy parametr `final` o wartości `false` albo `true`. Procedury `DrawSceneToMirror` i `DrawSceneToWindow`, których zadaniem jest narysowanie obiektów z uwzględnieniem oświetlenia, nadają temu parametrowi wartość `true`. Wcześniej wywołana procedura `DrawSceneToShadows` nadaje temu parametrowi wartość `false`, aby obszar cienia został znaleziony przy użyciu uproszczonego programu szaderów.

W pętli w liniach 32–39 znajdowane są obszary cienia dla kolejnych źródeł światła. W linii 34 jest uaktywniany pozaekranowy bufor ramki z załącznikiem — teksturą, w której powstaje reprezentacja obszaru cienia, a potem scena, tj. czajnik i torus oraz lustro są rysowane.

Listing 22.16. Procedury rysowania sceny z cieniami

C

```
void DrawScene ( AppData *ad, char final )
{
  LoadMMatrix ( ad, &ad->trans, ad->teapot_mmatrix );
  DrawMyTeapot ( ad, final );
  if ( ad->cnet )
    DrawMyCNet ( ad->myteapot, ad->brprog.program_id[1] );
  LoadMMatrix ( ad, &ad->trans, ad->torus_mmatrix );
  DrawMyTorus ( ad, final );
  if ( ad->cnet )
    DrawMyCNet ( ad->mytorus, ad->brprog.program_id[1] );
} /*DrawScene*/

void DrawSceneToMirror ( AppData *ad )
{
  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, ad->mirror_fbo );
  glViewport ( 0, 0, MIRRORTXT_W, MIRRORTXT_H );
  LoadVPMatrix ( &ad->trans, true );
  glClearColor ( 0.95, 0.95, 0.95, 1.0 );
  glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  DrawScene ( ad, true );
  glFlush ();
} /*DrawSceneToMirror*/

void DrawSceneToShadows ( AppData *ad )
{
  int    l;
  GLuint mask;

  glViewport ( 0, 0, SHADOW_MAP_SIZE, SHADOW_MAP_SIZE );
  glEnable ( GL_POLYGON_OFFSET_FILL );
  glPolygonOffset ( 2.0f, 4.0f );
  for ( l = 0, mask = 0x00000001;  l < ad->light.nls;  l++, mask <<= 1 )
    if ( ad->light.shmask & mask ) {
      BindShadowTxtFBO ( &ad->trans, &ad->light, l );
      glClear ( GL_DEPTH_BUFFER_BIT );
      LoadMMatrix ( &ad->trans, ad->mirror.mirror_matrix );
      DrawMirror ( &ad->mirror, ad->miprog[0], false );
      DrawScene ( ad, false );
    }
  glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
  glDisable ( GL_POLYGON_OFFSET_FILL );
  for ( l = 0, mask = 0x00000001;  l < ad->light.nls;  l++, mask <<= 1 )
    if ( ad->light.shmask & mask ) {
      glActiveTexture ( GL_TEXTURE2+l );
      glBindTexture ( GL_TEXTURE_2D, ad->light.ls[l].shadow_txt );
```

```
46:     }
47: } /*DrawSceneToShadows*/
48:
49: void DrawSceneToWindow ( AppData *ad )
50: {
51:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
52:   glViewport ( 0, 0, ad->camera.win_width, ad->camera.win_height );
53:   LoadVPMatrix ( &ad->trans, false );
54:   glClearColor ( 1.0, 1.0, 1.0, 1.0 );
55:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
56:   LoadMMatrix ( &ad->trans, ad->mirror.mirror_matrix );
57:   DrawMirror ( &ad->mirror, ad->miprog[1], true );
58:   DrawScene ( ad, true );
59:   glFlush ();
60: } /*DrawSceneToWindow*/
61:
62: void RedrawMyWorld ( void )
63: {
64:   glEnable ( GL_DEPTH_TEST );
65:   DrawSceneToShadows ( &appdata );
66:   DrawSceneToMirror ( &appdata );
67:   DrawSceneToWindow ( &appdata );
68: } /*RedrawMyWorld*/
```

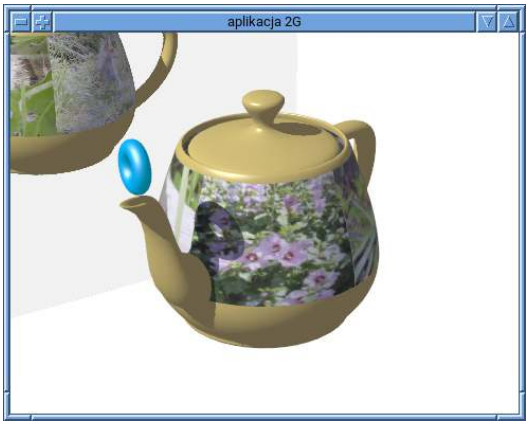
Przed znajdowaniem obszarów cienia wywołujemy procedurę `glEnable` z parametrem `GL_POLYGON_OFFSET_FILL` i procedurę `glPolygonOffset` z parametrami, które opisują korektę głębokości fragmentów zrasteryzowanych trójkątów. Ma to na celu „odsunięcie" powierzchni od obserwatora[16], co prowadzi do wstawienia do bufora głębokości nieco większych liczb. Korekta dodawana do głębokości z fragmentu jest opisana wzorem

$$o = f\,m + u\,\mathrm{ulp}\,z,$$

w którym litery f i u oznaczają parametry procedury `glPolygonOffset`, symbol m oznacza tangens kąta między płaszczyzną trójkąta a osią z układu współrzędnych okna, a $\mathrm{ulp}\,z$ jest wartością najmniej znaczącego bitu liczby z w buforze głębokości, tj. najmniejszym przyrostem głębokości „rozróżnianym" przez reprezentację liczb w tym buforze (zobacz s. 1187). Bez tej korekty punkty na powierzchni, rysowane na końcowym obrazie, mogłyby „same siebie zasłaniać" od światła wskutek błędów zaokrągleń i ograniczonej dokładności reprezentacji obszaru cienia. Korekta kompensuje też różnice błędów zaokrągleń w obliczeniach punktów powierzchni i rzutowaniu podczas wyznaczania obszarów cienia i podczas wykonywania końcowego obrazu. Powinna ona być tym większa, im większy jest zakres współrzędnych z wyznaczających tylną i przednią ścianę bryły widzenia konstruowanej w celu znalezienia reprezentacji obszaru cienia (czyli iloraz parametrów `far` i `near` procedury `M4x4Frustumf`

[16] w tym przypadku od źródła światła

albo różnica tych parametrów procedury `M4x4Orthof`, zobacz listing 22.1). Po otrzymaniu wszystkich reprezentacji obszarów cienia w teksturach korektę wyłączamy za pomocą procedury `glDisable`. W pętli w liniach 42–46 tekstury cieni przywiązujemy do celów `GL_TEXTURE_2D` w kolejnych punktach dowiązania, zaczynając od punktu o numerze 2 (identyfikowanego przez makro `GL_TEXTURE2`).



Rysunek 22.2. Okno aplikacji drugiej G

Listing 22.17. Włączanie i wyłączanie cieni

C

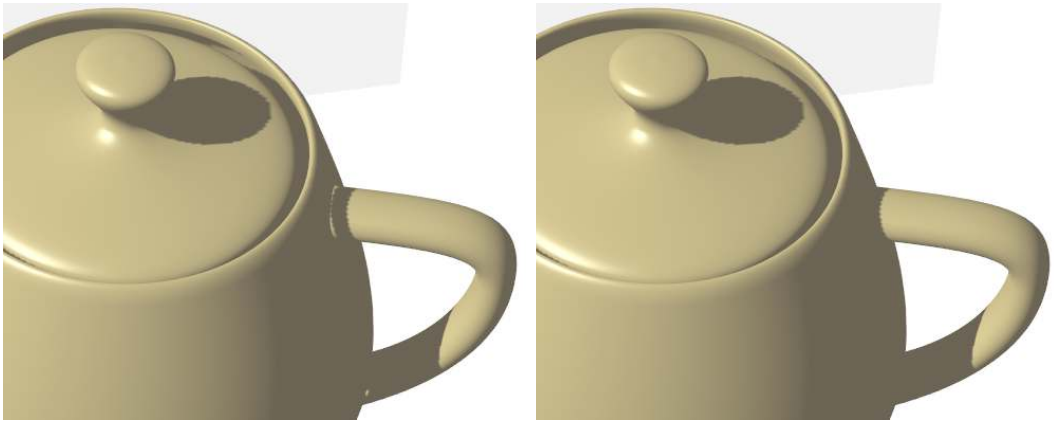
```
1: void SetShadowsOnOff ( LightBl *light, char on )
2: {
3:   GLuint z = 0;
4:
5:   glBindBuffer ( GL_UNIFORM_BUFFER, light->lsbuf );
6:   glBufferSubData ( GL_UNIFORM_BUFFER, lsbofs[2], sizeof(GLuint),
7:                     on ? &light->shmask : &z );
8: } /*SetShadowsOnOff*/
9:
10: char ProcessCharCommand ( char charcode )
11: {
12:   switch ( toupper ( charcode ) ) {
13: ....
14: case 'U':
15:     SetShadowsOnOff ( &appdata.light, appdata.shadows = !appdata.shadows );
16:     return true;
17: ....
18:   }
19:   return false;
20: } /*ProcessCharCommand*/
```

Naciskając klawisz z literą U, użytkownik może wyłączać i ponownie włączać rysowanie cieni. Jest to zrobione w najprostszy możliwy sposób (listing 22.17) — procedura SetShadowsOnOff nadaje zmiennej jednolitej light.shmask wartość maski bitowej opisującej utworzone przez aplikację tekstury cienia albo zero. Jeśli ta zmienna ma wartość 0, to szader fragmentów wykonujący końcowy obraz nie sprawdza, czy dany punkt jest w cieniu, ale wcześniej obszary cieni są niepotrzebnie znajdowane. Modyfikację procedury DrawSceneToShadows, która powinna, gdy cienie są wyłączone, natychmiast wykonać powrót (zamiast znajdować niepotrzebne reprezentacje obszarów cienia), zostawiam jako proste ćwiczenie.

22.6. Uzupełnienia

22.6.1. Poprawianie błędów reprezentacji obszaru cienia

Reprezentowanie obszaru cienia za pomocą tablicy (bufora głębokości wypełnianego podczas rysowania sceny widzianej z kierunku padania światła) wprowadza błędy, które mogą być widoczne na obrazach. Jeśli obraz przedstawia powierzchnie niebędące *całymi* brzegami brył, to w pobliżu przecięć tych powierzchni mogą pojawić się błędnie oświetlone piksele. Przykład jest pokazany na rysunku 22.3 z lewej strony: światło wpadające przez szczelinę między korpusem a pokrywką czajnika „przenika" na niewielką odległość, zależną od parametrów procedury glPolygonOffset, przez nieskończenie cienką powierzchnię korpusu, co powoduje błędne oświetlenie fragmentów uchwytu. Manipulowanie tymi parametrami w celu wyeliminowania takich błędów może tylko doprowadzić do otrzymania obrazów, na których punkty powierzchni same się zasłaniają od światła.



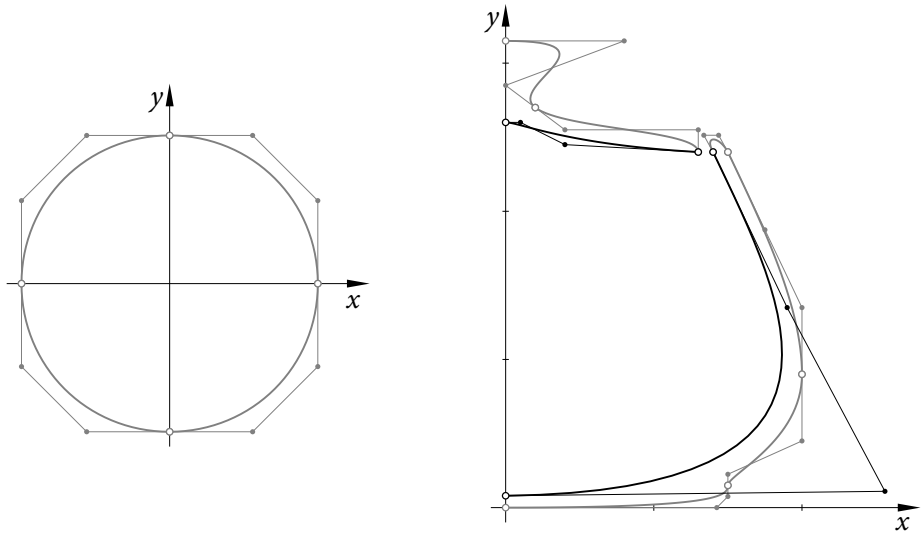
Rysunek 22.3. Obraz z błędnie oświetlonymi pikselami i obraz poprawionego obiektu

Jedynym skutecznym sposobem naprawienia takich błędów jest zmodyfikowanie obiektów: powinny one być obiektami z zamkniętą objętością, tj. powierzchniami brył, a zatem na przykład do czajnika trzeba dorobić powierzchnię wewnętrzną korpusu i pokrywki tak, aby ściany czajnika i pokrywka miały pewną niezerową grubość. Obrazek z prawej strony rysunku 22.3 przedstawia scenę ze zmodyfikowanym czajnikiem. Dziobka nie zmieniłem,

ale gdyby były potrzebne obrazy z obserwatorem umieszczonym wewnątrz dziobka, to należałoby dorobić również powierzchnię wewnętrzną dziobka.

Modyfikacja czajnika polegała na dodaniu ośmiu płatów Béziera (czterech dla korpusu i tylu samo dla pokrywki), przy czym długość tablicy punktów kontrolnych `teapotcp` (i długość bufora w pamięci GPU, w której te punkty mają się znaleźć) trzeba było zwiększyć tak, aby pomieścić dodatkowe 50 punktów; tylko tyle, bo nowe płaty mają wiele wspólnych punktów kontrolnych z innymi nowymi płatami i z płatami oryginalnego czajnika.

Rysunek 22.4 przedstawia krzywe użyte do skonstruowania korpusu i pokrywki czajnika i krzywe, na podstawie których otrzymałem płaty uzupełniające oryginalny model. Każdy płat Béziera będący częścią korpusu i pokrywki jest iloczynem sferycznym dwóch krzywych Béziera trzeciego stopnia. Pierwsza z tych krzywych jest przybliżeniem ćwiartki okręgu jednostkowego[17], a druga jest fragmentem tworzącej. Sposób obliczania punktów kontrolnych iloczynu sferycznego na podstawie punktów kontrolnych krzywych Béziera będących argumentami tego działania jest opisany w podrozdziale 17.1. Jak wiemy (z podrozdz. 15.1), końcowe punkty krzywej Béziera (której dziedziną jest przedział $[0,1]$) pokrywają się z pierwszym i ostatnim punktem kontrolnym krzywej. Dlatego przedłużając tworzące korpusu i pokrywki czajnika o dodatkowe krzywe Béziera, pierwszy punkt kontrolny przyjąłem w tym samym położeniu co odpowiedni punkt kontrolny krzywej użytej do otrzymania oryginalnego korpusu lub pokrywki, a ostatni punkt kontrolny umieściłem na osi y.



Rysunek 22.4. Krzywe Béziera użyte w konstrukcji oryginalnego czajnika i płatów dodatkowych

Zwracam uwagę, że dokonana modyfikacja miała na celu *tylko* poprawienie błędów obrazowania cieni. Chcąc *wyprodukować* czajnik na podstawie modelu komputerowego (np. przy

[17] Odległości punktów każdej z tych krzywych od punktu $(0,0)$ leżą w przedziale $[1, 1.0041]$, zatem błąd aproksymacji okręgu przez te krzywe nie przekracza 0.41% promienia okręgu.

użyciu drukarki 3D), należałoby zapewne wprowadzić tworzącą wewnętrznej powierzchni korpusu o bardziej skomplikowanym kształcie, aby pokrywka nie mogła wpaść do środka. Do otrzymania takiego kształtu trzeba by użyć dwóch (lub więcej) krzywych Béziera trzeciego stopnia, a więc tak poprawiony model miałby jeszcze co najmniej cztery dodatkowe płaty Béziera.

Procedurę `ConstructTheTeapot` pokazaną na listingu 15.12 zmieniłem w ten sposób, że zadeklarowane w niej tablice przeniosłem na zewnątrz, dzięki czemu nowa procedura o nazwie `ConstructAltTeapot` też ma dostęp do tych tablic. Tablice wydłużyłem, dopisując nowe dane na końcu. Wywołując procedurę `EnterBezierPatchesElem`, nowa procedura podaje długości całych tablic, reprezentujących 40 płatów stopnia $(3, 3)$ za pomocą 356 punktów kontrolnych.

22.6.2. Antyaliasing cienia

Wadą zrealizowanego w aplikacji 2G algorytmu są „ząbkowane" brzegi obszarów cienia na narysowanych powierzchniach[18]. Nie można tego całkowicie wyeliminować, choć w zasadzie można zmniejszyć przez zwiększanie rozmiarów tekstury cienia. Ale koszty pamięciowe i czasowe takiego postępowania mogą być duże, konieczny jest więc umiar. Najczęściej jednak cienie mają rozmyte brzegi; idealnie ostre brzegi cieni pojawiają się w obecności niespotykanych na co dzień punktowych źródeł światła. Jeśli źródłem światła jest pewien obszar (np. powierzchnia Słońca lub klosz lampy), to obszary w pełni oświetlone i całkowicie pogrążone w cieniu są rozdzielone obszarem półcienia.

Wygląd brzegów cienia na obrazie można poprawić, „rozmywając" je. Polega to na zbadaniu widoczności, z punktu położenia źródła światła, pewnej liczby punktów w otoczeniu przetwarzanego fragmentu powierzchni. Wyniki testów są uśredniane i przyjmowana jest intensywność bezpośredniego oświetlenia proporcjonalna do obliczonej średniej. Dzięki temu „ząbki" na brzegu cienia stają się słabiej dostrzegalne, choć pozostają widoczne. Metodę tę opublikowali w 1987 r. Reeves, Salesin i Cook, którzy nadali jej nazwę *percentage-closer filtering* (w skrócie *PCF*)[19]. Zrealizujemy jej najprostszy wariant.

W porównaniu z funkcjami o wspólnej (przeciążonej) nazwie `texture` lub `textureProj` funkcje `textureOffset` i `textureProjOffset` (s. 207) mają dodatkowy parametr wektorowy. Współrzędne tego wektora są liczbami całkowitymi, a ich liczba odpowiada wymiarowi tekstury. Funkcje te po obliczeniu indeksów do tablicy tekseli dodają do nich współrzędne wektora podanego jako parametr, wskutek czego wartość tekstury jest obliczana w nieco innym punkcie.

Aby osiągnąć zamierzony cel, wystarczy zmienić funkcję `IsEnlighted` z listingu 22.5 na podprogram pokazany na listingu 22.18. Podprogram ten wykonuje 9 testów, czy dany punkt jest w obszarze cienia, za każdym razem „przesuwając obszar cienia" tak, aby „nad tym punktem" znalazł się inny teksel tekstury cienia. W ten sposób dla punktów położonych w pobliżu brzegu cienia poszczególne testy mogą dać różne wyniki. Wynikiem każdego testu

[18] Wady tej są pozbawione algorytmy rysowania brył cienia wspomniane na początku tego rozdziału.

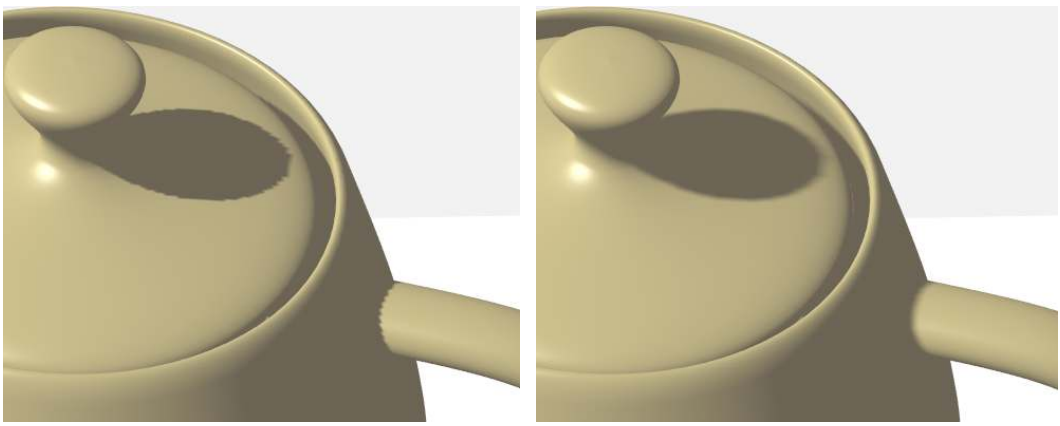
[19] Słowo *percentage*, czyli odsetek, jest przesadne — nie wykonuje się setki testów.

jest liczba 0 lub 1; średnia arytmetyczna tych liczb jest jedną z dziesięciu możliwych wartości funkcji przekazywanych w instrukcji `return`. Półcienie na końcowym obrazie mają więc osiem poziomów pośrednich między pełnym oświetleniem a całkowitym zaćmieniem.

Listing 22.18. Podprogram realizujący metodę PCF

GLSL

```
float IsEnlighted ( uint l )
{
  float s;

  s = textureProj ( shtex[l], In.ShadowPos[l] ) +
      textureProjOffset ( shtex[l], In.ShadowPos[l], ivec2(-1,0) ) +
      textureProjOffset ( shtex[l], In.ShadowPos[l], ivec2(+1,0) ) +
      textureProjOffset ( shtex[l], In.ShadowPos[l], ivec2(0,-1) ) +
      textureProjOffset ( shtex[l], In.ShadowPos[l], ivec2(0,+1) ) +
      textureProjOffset ( shtex[l], In.ShadowPos[l], ivec2(-1,-1) ) +
      textureProjOffset ( shtex[l], In.ShadowPos[l], ivec2(-1,+1) ) +
      textureProjOffset ( shtex[l], In.ShadowPos[l], ivec2(+1,-1) ) +
      textureProjOffset ( shtex[l], In.ShadowPos[l], ivec2(+1,+1) );
  return s/9.0;
} /*IsEnlighted*/
```



Rysunek 22.5. Obraz z cieniem ostrym i obraz z otrzymanym metodą PCF cieniem rozmytym

Przesunięcia realizowane przez trzeci parametr funkcji `textureProjOffset` są równoległe do płaszczyzny xy w układzie obserwatora związanego ze źródłem światła. Jeśli rysowany trójkąt nie leży w płaszczyznie równoległej do tej płaszczyzny, to dany punkt na powierzchni *będzie* zasłonięty od światła przez punkty odpowiadające niektórym ze sprawdzanych tekseli. Dlatego stosując ten algorytm, trzeba zwiększyć współczynniki korygujące podane jako parametry procedury `glPolygonOffset`; w przykładzie pokazanym na rysunku 22.5 nadałem tym parametrom wartości `6.0f` i `2.0f`. Nie należy zwiększać ich

jeszcze bardziej, bo wtedy grubość pokrywki i ścian czajnika otrzymanych po wprowadzeniu dodatkowych płatów stanie się zbyt mała, aby wyeliminować błędy rozważane w p. 22.6.1.

Zamiast przesuwać teksturę cienia, można wprowadzić odpowiednie przesunięcia punktu powierzchni — przesunięcia te powinny być równoległe do płaszczyzny rysowanego trójkąta. Wektory przesunięć trzeba obliczać w układzie obserwatora związanego ze źródłem światła. Można dalej rozwinąć ten algorytm, uzależniając wielkość rozmycia cienia od odległości między danym punktem na powierzchni a punktem zasłaniającym go od światła. Dzięki temu „rozmycie" brzegu cienia (czyli szerokość obszaru półcienia) będzie rosnąć ze wzrostem tej odległości, poprawiając realizm obrazu.

22.7. Ćwiczenia

1. Wykonaj eksperymenty polegające na zmienianiu parametrów procedury `glPolygonOffset` (w procedurze `DrawSceneToShadows`) i obserwowaniu skutków tych zmian.
2. Rozszerz aplikację o animowanie źródła (lub źródeł) światła, polegające na przykład na obracaniu położenia źródła światła wokół osi x z prędkością jednego obrotu na kilka sekund. Po zmianie położenia l-tego źródła światła trzeba uaktualnić macierze V_l i P_l.
3. Utwórz (w dodatku do czajnika oryginalnego) reprezentację czajnika zmodyfikowanego zgodnie z opisem w p. 22.6.1 i zmień aplikację tak, aby można było przełączać wyświetlany model czajnika przy użyciu jakiegoś klawisza. Wykonaj kolejne eksperymenty.
4. *Dodaj do modelu czajnika jeszcze dwa takie płaty Béziera, aby powstała wewnętrzna powierzchnia dziobka. Powstanie w ten sposób obiekt z zamkniętą objętością (zobacz podrozdz. 7.6), co umożliwi otrzymanie poprawnych obrazów czajnika w krótszym czasie, po włączeniu odrzucania ścian odwróconych tyłem do obserwatora.

23

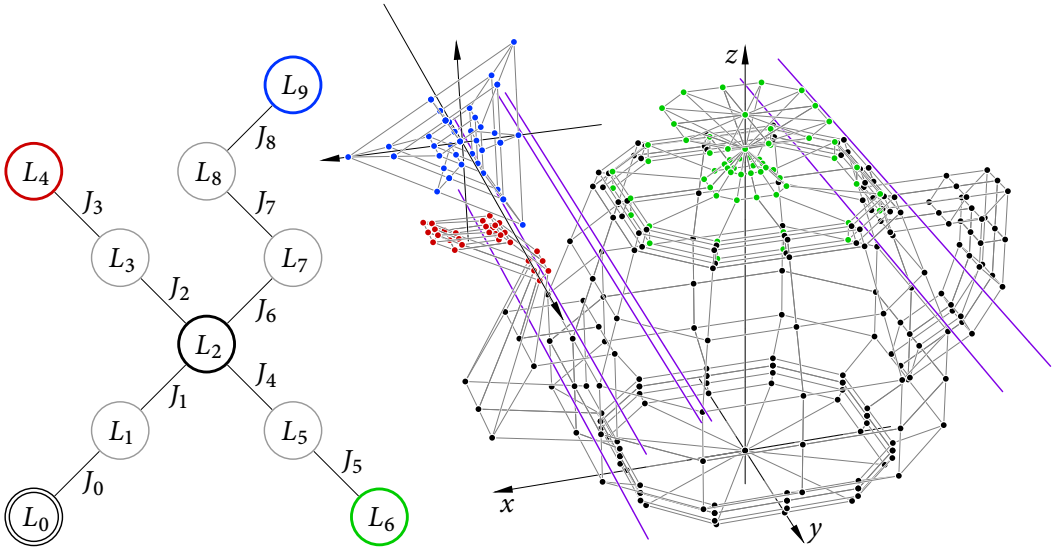
Aplikacja druga H

Dokonamy bardziej skomplikowanej animacji. Utworzymy mianowicie łańcuch kinematyczny, który umożliwi odkształcanie obiektów. Aby usprawnić nalewanie herbaty, będziemy odkształcać dziobek czajnika. Stanie się to okazją do zapoznania się z szaderami obliczeniowymi. Łańcuch zrealizujemy przy użyciu procedur opisanych w podrozdziale 13.2.

23.1. Łańcuch kinematyczny czajnika

Na rysunku 23.1 jest pokazany graf łańcucha tworzonego przez opisaną w tym rozdziale aplikację. Łańcuch ma 10 członów (oznaczonych symbolami $L_0, \ldots, L_9$) i 9 par kinematycznych ($J_0, \ldots, J_8$); jego graf jest drzewem. Punkty kontrolne korpusu, uchwytu i części dziobka czajnika zaznaczone czarnymi kropkami są ustalone w układzie współrzędnych członu L_2. Pozostałe punkty kontrolne dziobka (czerwone) mają ustalone położenia w układzie członu L_4. Punkty kontrolne pokrywki (zielone) są związane z członem L_6, torus (którego punkty kontrolne są zaznaczone na niebiesko) jest związany z członem L_9. Pozostałe człony służą do określenia par kinematycznych realizujących dopuszczalne przemieszczenia obiektów (tj. ich punktów kontrolnych) względem siebie.

W łańcuchu z rysunku 23.1 wszystkie macierze F_j reprezentują przesunięcia, a każda macierz $R_j(\varphi)$ reprezentuje obrót o kąt φ wokół osi y lub z układu współrzędnych. Przypomnijmy (zobacz s. 316), że jeśli F_j jest macierzą przesunięcia o *wektor* $\boldsymbol{f}_j$ oraz $B_j = F_j^{-1}$, to iloczyn $F_j R_j(\varphi) B_j$ reprezentuje obrót o kąt φ wokół przechodzącej przez *punkt* $\boldsymbol{f}_j$ osi równoległej do osi obrotu $R_j(\varphi)$ (która przechodzi przez początek układu współrzędnych). Ponadto, jeśli z kolejnymi dwiema krawędziami łańcucha, J_j i J_k, znajdującymi się w drodze od korzenia do pewnego członu L_i są związane macierze F_j i F_k opisujące przesunięcia o wektory $\boldsymbol{f}_j$ i $\boldsymbol{f}_k$ oraz $B_j = F_j^{-1}$, to w wyrażeniu opisującym macierz A_i przejścia od układu tego członu do układu świata występuje iloczyn $B_j F_k$, który reprezentuje przesunięcie o wektor $\boldsymbol{t}_{kj} = \boldsymbol{f}_k - \boldsymbol{f}_j$. Jeśli macierze $R_j(\varphi_j)$ i $R_k(\varphi_k)$ opisują obroty wokół *tej samej* osi, to osie odpowiednich obrotów realizowanych w łańcuchu będą równoległe i druga oś będzie przesunięta względem pierwszej o wektor $\boldsymbol{t}_{kj}$.

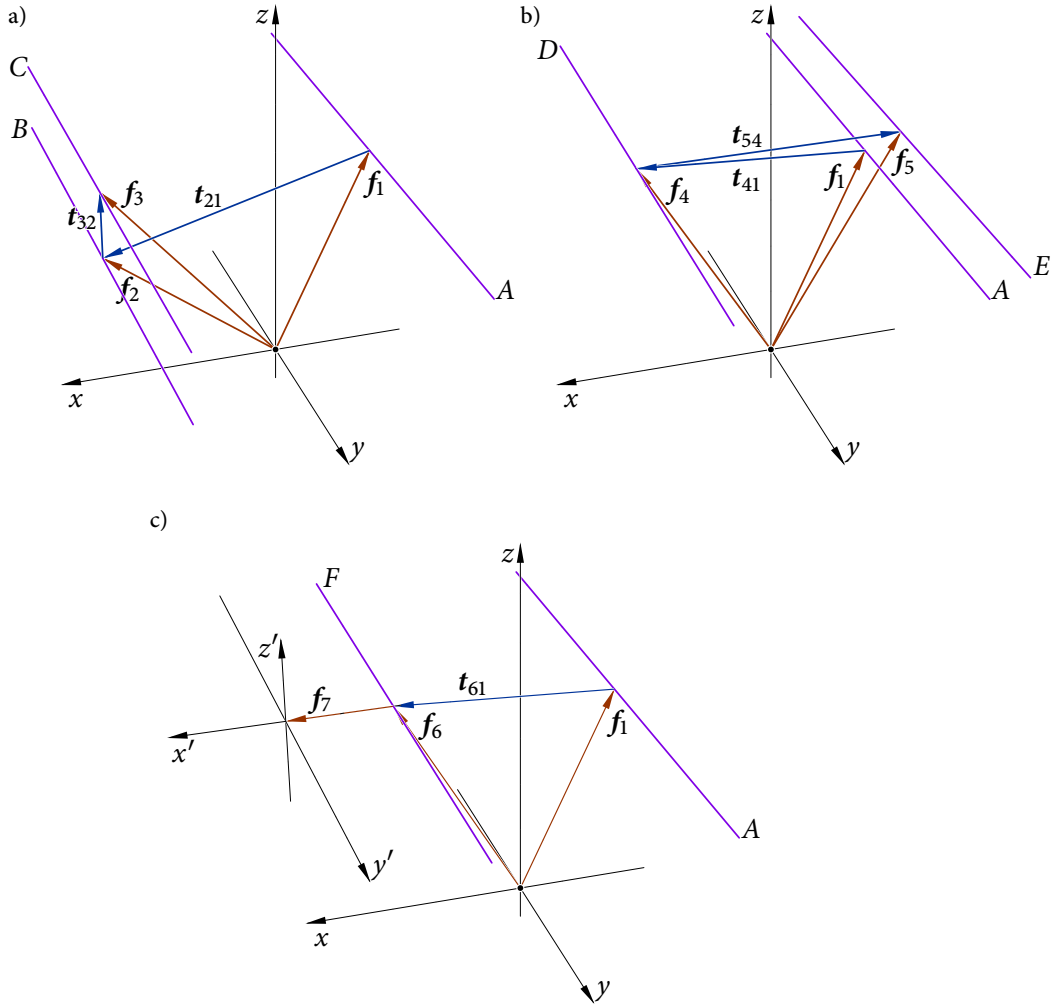


Rysunek 23.1. Łańcuch kinematyczny czajnika z torusem

Na rysunku 23.2 są pokazane opisane wyżej wektory dla trzech ścieżek w grafie łańcucha kinematycznego z rysunku 23.1 i osie obrotów w **położeniu wyjściowym**, przyjmowanym, gdy wszystkie parametry artykulacji (kąty obrotów) są równe 0. Macierze F_j i B_j przyjmiemy tak, aby układy współrzędnych związane z członami $L_0, \ldots, L_6$ w położeniu wyjściowym były identyczne; na rysunkach 23.1 i 23.2 są pokazane osie x, y, z wszystkich tych układów. Na rysunku 23.2c są też osie x', y', z' układu współrzędnych członu L_8 w położeniu wyjściowym.

Para kinematyczna J_0 umożliwia obracanie członów $L_1, \ldots, L_9$ wokół osi z układu współrzędnych członu L_0. Para J_1 odpowiada za obracanie członów $L_2, \ldots, L_9$ wokół prostej A, równoległej do osi y układu współrzędnych członu L_1 i przechodzącej przez przez punkt (o współrzędnych w tym układzie) $(-0.43, 0, 0.92)$, w związku z czym macierz F_1 jest macierzą przesunięcia o wektor $\boldsymbol{f}_1 = (-0.43, 0, 0.92)$. Macierz B_1 jest odwrotnością macierzy F_1. Zauważmy, że macierz $R_1(0)$ (obrotu o kąt 0 wokół osi y) jest jednostkowa, dlatego macierz $F_1R_1(0)B_1$ też jest jednostkowa. Właśnie dzięki temu układy członów L_1 i L_2 (połączonych parą kinematyczną J_1) w położeniu wyjściowym są identyczne.

Pary J_2 i J_3 realizują odkształcenia dziobka czajnika. Człon L_3 może się obracać względem członu L_2 (z którym jest związany korpus czajnika) wokół prostej B, równoległej do osi y i przechodzącej przez punkt $(0.78, 0, 0.59)$ w układzie członu L_2, natomiast człon L_4 może się obracać względem członu L_3 wokół prostej C przechodzącej przez punkt $(0.78, 0, 0.91)$. Macierze F_2 i F_3 są macierzami przesunięć o wektory $\boldsymbol{f}_2 = (0.78, 0, 0.59)$ i $\boldsymbol{f}_3 = (0.78, 0, 0.91)$, a macierze B_2 i B_3 są ich odwrotnościami. Zauważmy, że w układzie współrzędnych członu L_3 prosta C ma położenie ustalone — powstaje ona przez przesunięcie prostej B o wektor $\boldsymbol{t}_{32} = \boldsymbol{f}_3 - \boldsymbol{f}_2$. Natomiast w układzie członu L_2 zmiany parametru φ_2 powodują obracanie prostej C wokół prostej B.



Rysunek 23.2. Osie obrotów w położeniu wyjściowym

Podobnie, pary kinematyczne J_4 i J_5 realizują obroty wokół prostych D i E równoległych do osi y i przechodzących przez punkty $(0.6, 0, 1)$ i $(-0.6, 0, 1)$ (rys. 23.2b). Ta gałąź łańcucha umożliwia przemieszczanie względem korpusu czajnika pokrywki, której punkty kontrolne mają ustalone położenia w układzie współrzędnych członu L_6. Tu również jest $B_4 = F_4^{-1}$ i $B_5 = F_5^{-1}$, co zapewnia pokrywanie się układów współrzędnych członów L_5 i L_6 z układem członu L_2 w położeniu wyjściowym. Co nam to daje? To, że poszczególne punkty kontrolne czajnika, dane w jednym układzie współrzędnych (w którym czajnik został zdefiniowany), możemy wygodnie przywiązać do różnych członów — wystarczy dla każdego punktu zadeklarować, w układzie którego członu jego położenie jest ustalone. Punkty zaznaczone różnymi kolorami na rysunku 23.1 ustalamy w układach członów L_2, L_4 i L_6; w położeniu wyjściowym da to nam czajnik nieodkształcony, ale zmieniając parametry artykulacji spo-

wodujemy przemieszczanie się punktów związanych z poszczególnymi członami względem innych członów łańcucha.

Nieco inaczej traktujemy ścieżkę złożoną z krawędzi J_6, J_7, J_8 w grafie łańcucha, ponieważ punkty kontrolne torusa są dane w układzie, którego początek jest środkiem torusa (a jego osią jest oś z, rys. 23.2c). Chcemy, aby środek torusa pozostawał na osi y' układu członu L_8, obracającej się wokół równoległej do niej prostej F. Chcemy również, aby osie z układu członu L_2 i z' układu członu L_8 pozostawały równoległe. Dlatego w tym przypadku przyjmujemy macierze F_6 i F_7 reprezentujące przesunięcia o wektory $\boldsymbol{f}_6 = (0.52, 0, 0.97)$ i $\boldsymbol{f}_7 = (0.52, 0, 0)$, a na macierze B_6 i B_7 wybieramy macierz jednostkową. Jeśli zapewnimy, że $\varphi_7 = -\varphi_6$, to odpowiednie osie układów członów L_2 i L_8 będą zawsze równoległe, ale w położeniu wyjściowym początek układu członu L_8 jest przesunięty względem początku układu L_2 o wektor $\boldsymbol{f}_6 + \boldsymbol{f}_7$. Para J_8 realizuje obracanie torusa wokół osi z', bez przesunięć (zatem macierz $F_8 = B_8$ jest jednostkowa).

Macierz A_0 w opisanej tu aplikacji reprezentuje przesunięcie o wektor $(0, 0, -0.6)$.

Macierz E_0 opisująca dodatkowe wstępne przekształcenie czajnika (który jest obiektem numer 0 w łańcuchu) reprezentuje skalowanie osi x, y, z o czynniki $\frac{1}{3}$, $\frac{1}{3}$, $\frac{4}{9}$ — w poprzednich wersjach drugiej aplikacji używaliśmy takiego skalowania, aby otrzymać czajnik o dogodnej wielkości, jednocześnie przywracając mu oryginalne proporcje (zobacz s. 393 i 397). Natomiast torus (obiekt numer 1) zdefiniowany przy użyciu promieni $R = 1$ i $r = \frac{1}{2}$ zmniejszamy w skali $\frac{1}{10}$ i obracamy wokół osi x o kąt $\frac{\pi}{2}$, podobnie jak w procedurze `SetupTorusMatrix` na listingu 17.3. Mając dany związany z członem L_i punkt $\boldsymbol{p}$ l-tego obiektu, potrzebujemy znaleźć wektor współrzędnych (jednorodnych) tego punktu w układzie świata. Zadanie obliczania punktów $A_i E_l \boldsymbol{P}$ (gdzie $\boldsymbol{P}$ jest wektorem współrzędnych jednorodnych przywiązanego do i-tego członu punktu $\boldsymbol{p}$ należącego do l-tego obiektu) zlecimy szaderowi obliczeniowemu.

23.2. Szader obliczeniowy artykulacji

Szadery obliczeniowe wywołuje się w blokach zwanych **grupami roboczymi** (*workgroups*). Dokładniej, **globalna grupa robocza** jest trójwymiarową tablicą **lokalnych grup roboczych** będących trójwymiarowymi blokami poszczególnych wątków szadera. Podane w treści szadera wymiary grupy lokalnej po jego skompilowaniu i zbudowaniu programu nie mogą być zmieniane, natomiast wymiary grupy globalnej, ustalane w chwili uruchomienia programu, za każdym razem mogą być inne.

Zadaniem szadera obliczeniowego aplikacji drugiej H (listing 23.1) jest obliczenie współrzędnych w układzie świata punktów kontrolnych czajnika i torusa na podstawie współrzędnych podanych w układach, w których te obiekty zostały zdefiniowane. Kod szadera opisuje obliczenie dla *jednego* punktu. Ponieważ nie ma potrzeby komunikacji ani synchronizacji między wątkami przetwarzającymi poszczególne punkty, najlepiej jest przyjąć, że lokalne grupy robocze mają wymiary $1 \times 1 \times 1$ (czyli każda grupa składa się z jednego wątku), a wymiary globalnej grupy roboczej to $n \times 1 \times 1$ — grupa jest jednowymiarowa, a jej długość jest liczbą punktów do przekształcenia.

Listing 23.1. Szader obliczeniowy artykulacji

GLSL

```
1: #version 450 core
2:
3: layout(local_size_x=1) in;
4:
5: layout(std430,binding=0) buffer CPoints { float cp[]; } cpin;
6: layout(std430,binding=1) buffer CPointsOut { float cp[]; } cpout;
7: layout(std430,binding=2) buffer Tr { mat4 tr[]; } cptr;
8: layout(std430,binding=3) buffer TrInd { uint ind[]; } cptrind;
9:
10: uniform int dim, trnum;
11:
12: void main ( void )
13: {
14:   uint i, k;
15:   vec4 v0, v1;
16:
17:   i = gl_GlobalInvocationID.x;
18:   k = dim*i;
19:   i = trnum >= 0 ? trnum : cptrind.ind[i];
20:   switch ( dim ) {
21: case 3:
22:     v0 = vec4 ( cpin.cp[k], cpin.cp[k+1], cpin.cp[k+2], 1.0 );
23:     v1 = cptr.tr[i]*v0;
24:     cpout.cp[k] = v1.x;  cpout.cp[k+1] = v1.y;  cpout.cp[k+2] = v1.z;
25:     break;
26: case 4:
27:     v0 = vec4 ( cpin.cp[k], cpin.cp[k+1], cpin.cp[k+2], cpin.cp[k+3] );
28:     v1 = cptr.tr[i]*v0;
29:     cpout.cp[k] = v1.x;   cpout.cp[k+1] = v1.y;
30:     cpout.cp[k+2] = v1.z; cpout.cp[k+3] = v1.w;
31:     break;
32: default:
33:     break;
34:   }
35: } /*main*/
```

Wymiary lokalnej grupy roboczej są określone przez kwalifikator w linii 3, przy czym drugi i trzeci wymiar, niepodane w kwalifikatorze, są równe 1 (zobacz podrozdz. 9.15). Numer punktu, który dany wątek ma przekształcić, jest odczytywany w linii 17 ze zmiennej wbudowanej `gl_GlobalInvocationId.x`. Współrzędne położenia punktu szader odczytuje z tablicy `cp` w bloku `CPoints` (lokalnie nazwanym `cpin`) i wpisuje wynik do tablicy `cp` w bloku `CPointsOut` (o lokalnej nazwie `cpout`). Oba bloki magazynowe, `CPoints` i `CPointsOut`, są tak samo zbudowane, zawierają tablicę liczb typu `float`, których trójki lub czwórki opisują położenia punktów kontrolnych. Liczba współrzędnych punktu ma być podana w zmiennej jednolitej `dim`, o czym nie wolno zapomnieć przed przystąpieniem do obliczeń.

W bloku zmiennych jednolitych `Tr` (o lokalnej nazwie `cptr`) znajduje się tablica macierzy 4×4; to są macierze przejścia od układu modelu, poprzez układy odpowiednich członów łańcucha kinematycznego, do układu świata. Wartość zmiennej jednolitej `trnum`, jeśli jest nieujemna (co jest sprawdzane w linii 19), jest indeksem do tablicy `cptr.tr` i wybiera przekształcenie, któremu mają być poddane punkty przetwarzane przez *wszystkie* wątki w grupie roboczej. To zdarza się w sytuacji, gdy wszystkie punkty danego obiektu mają położenia ustalone w układzie jednego członu łańcucha kinematycznego (i nie korzysta się wtedy z tablicy `cptrind.ind`). Jeśli zmienna `trnum` ma wartość ujemną, to macierz przekształcenia należy wybrać z tablicy `cptrind.ind` przy użyciu indeksu, który jest numerem przetwarzanego punktu. Tak więc szader czyta współrzędne jednego punktu, wybiera odpowiednią macierz, oblicza wynik i zapisuje go we właściwym miejscu bufora magazynowego `CPointsOut`. Działające równolegle (na wielu procesorach GPU) wątki szadera nie mają konfliktów w dostępie do tego bufora, bo każdy wątek wpisuje swój wynik do innego miejsca w nim.

Zależnie od liczby współrzędnych punktu w liniach 22–24 albo 27–30 szader wybiera z tablicy odpowiednią trójkę albo czwórkę liczb, tworzy z nich wektor w $\mathbb{R}^4$, wykonuje mnożenie i zapisuje 3 albo wszystkie 4 współrzędne iloczynu w tablicy `cpout.cp`.

Mając przygotowany do pracy (tj. skompilowany i złączony oraz wybrany za pomocą procedury `glUseProgram`) program z opisanym tu szaderem obliczeniowym, po nadaniu stosownych wartości potrzebnym zmiennym jednolitym i przywiązaniu właściwych buforów do odpowiednich punktów dowiązania uruchomimy obliczenia, przez wywołanie procedury `glDispatchCompute`.

Dzięki identycznej budowie bloków `CPoints` i `CPointsOut` możemy zrealizować następujący pomysł: tablice oryginalnych punktów kontrolnych czajnika i torusa umieścimy w odpowiednich buforach w pamięci GPU. Ponadto utworzymy dodatkowe bufory o tych samych wielkościach z przeznaczeniem na punkty przekształcone. Dokonując artykulacji łańcucha, zapamiętamy odpowiednie macierze w buforze, który stanie się blokiem `Tr`. Metoda `postprocess` czajnika lub torusa przywiąże bufor z jego punktami kontrolnymi jako blok `CPoints` i podstawi bufor magazynowy na przekształcone punkty jako blok `CPointsOut`. Rysując następnie czajnik lub torus, bufor z przekształconymi punktami przywiążemy jako blok `CPoints` dla szaderów rozdrabniania, które obliczają punkty płatów Béziera. W ten sposób nie musimy wprowadzać żadnych zmian w szaderach uruchomionych we wcześniejszych wersjach aplikacji.

Listing 23.2 przedstawia struktury danych aplikacji 2H; typy `Camera`, `Mirror` i `BPRenderPrograms` są identyczne jak w aplikacji 2G. Nowa struktura `KLArticulationProgram` jest opakowaniem danych opisujących program z szaderem z listingu 23.1. W polach tej struktury są pamiętane identyfikator programu, położenia zmiennych jednolitych `dim` i `trnum` oraz numery punktów dowiązania bloków magazynowych `cpin`, `cpout`, `cptr` i `cptrind`.

Struktura `KLBezPatches` opisuje obiekt — zespół płatów Béziera — na potrzeby artykulacji łańcucha kinematycznego. Elementy w tablicy `bpatches` wskazują struktury reprezentujące zespoły płatów, przy czym pierwsza z nich reprezentuje płaty nieodkształcone, a druga płaty poddane artykulacji. W obu tych strukturach są pamiętane identyfikatory tych samych buforów w pamięci GPU, z wyjątkiem bufora zawierającego współrzędne położenia punk-

tów kontrolnych. Pole `texture` przechowuje identyfikator tekstury do nałożenia na płaty. W polu `tribuf` jest pamiętany identyfikator bufora z indeksami przekształceń poszczególnych wierzchołków. Wartość pola `ntr`, jeśli jest nieujemna, jest indeksem przekształcenia dla wszystkich wierzchołków. W polu `mtn` jest przechowywany numer materiału.

Listing 23.2. Struktury danych aplikacji 2H

C

```
typedef struct {
    GLuint progid;
    GLuint dim_loc, ncp_loc, trnum_loc;
    GLuint cpibp, cpobp, ctrbp, ctribp;
  } KLArticulationProgram;

typedef struct {
    BezierPatchObjf *bpatches[2];
    GLuint          texture, tribuf;
    GLint           ntr, mtn;
  } KLBezPatches;

typedef struct {
    Camera                camera;
    kl_linkage            *linkage;
    KLBezPatches          bezp[2];
    Mirror                mirror;
    TransBl               trans;
    LightBl               light;
    MatBl                 mat;
    GLuint                lktrbuf;
    GLint                 BezNormals, TessLevel;
    char                  cnet, skeleton, animate, shadows, final;
    double                teapot_rot_angle;
    BPRenderPrograms      bprprog;
    GLuint                miprog[2];
    KLArticulationProgram artprog;
  } AppData;
```

Nowe pola struktury `AppData` są następujące: pole `linkage` jest wskaźnikiem struktury łańcucha kinematycznego, tablica `bezp` zastąpiła wskaźniki `myteapot` i `mytorus`. Pole `lktrbuf` jest identyfikatorem bufora z blokiem `cptr` szadera artykulacji. Pole `final` ma wartość `false` podczas znajdowania obszaru cienia i `true` podczas rysowania obiektów z oświetleniem. W polu `artprog` są zapisane dane umożliwiające używanie programu artykulacji. Pola `torus_rot_angle`, `teapot_mmatrix` i `torus_mmatrix` zostały usunięte, bo odpowiednie dane są teraz przechowywane w łańcuchu kinematycznym.

Listing 23.3 przedstawia procedurę, która kompiluje i łączy program z opisanym wcześniej szaderem obliczeniowym. Odczytane z tego programu numery punktów dowiązania bloków magazynowych i położenia zmiennych jednolitych są zapamiętywane we wskazywanym przez parametr procedury opakowaniu przedstawionym na listingu 23.2.

Listing 23.3. Procedura `LoadLinkageArticulationProgram`

C

```
1: void LoadLinkageArticulationProgram ( KLArticulationProgram *prog )
2: {
3:   static const char *filename[] = { "app2h.comp.glsl" };
4:   static const GLchar *CPINames[] = { "CPoints" };
5:   static const GLchar *CPONames[] = { "CPointsOut" };
6:   static const GLchar *CompTrNames[] =  { "Tr" };
7:   static const GLchar *CompTrIndNames[] = { "TrInd" };
8:   static const GLchar DimName[] = "dim";
9:   static const GLchar TrNumName[] = "trnum";
10:  GLuint shader_id;
11:  GLint  i;
12:
13:  shader_id = CompileShaderFiles ( GL_COMPUTE_SHADER, 1, &filename[0] );
14:  prog->progid = LinkShaderProgram ( 1, &shader_id, "articulate" );
15:  GetAccessToStorageBlock ( prog->progid, 0, &CPINames[0], &i, &i,
16:                            &prog->cpibp );
17:  GetAccessToStorageBlock ( prog->progid, 0, &CPONames[0], &i, &i,
18:                            &prog->cpobp );
19:  GetAccessToStorageBlock ( prog->progid, 0, &CompTrNames[0], &i, &i,
20:                            &prog->ctrbp );
21:  GetAccessToStorageBlock ( prog->progid, 0, &CompTrIndNames[0], &i, &i,
22:                            &prog->ctribp );
23:  prog->dim_loc = glGetUniformLocation ( prog->progid, DimName );
24:  prog->trnum_loc = glGetUniformLocation ( prog->progid, TrNumName );
25:  glDeleteShader ( shader_id );
26:  ExitIfGLError ( "LoadLinkageArticulationProgram" );
27: } /*LoadLinkageArticulationProgram*/
28:
29: void DeleteLinkageArticulationProgram ( KLArticulationProgram *prog )
30: {
31:  glUseProgram ( 0 );
32:  glDeleteProgram ( prog->progid );
33: } /*DeleteLinkageArticulationProgram*/
```

23.3. Budowanie łańcucha kinematycznego i metody jego obiektów

Listing 23.4 przedstawia procedurę budowania łańcucha, przy czym procedury wywoływane przez tę procedurę lub przypisywane jako metody obiektom łańcucha są opisane dalej. Parametr tej procedury to wskaźnik zmiennej typu `AppData`, który ma być przypisany polu `usrdata` struktury łańcucha. Po jej utworzeniu kolejne instrukcje budują poszczególne elementy łańcucha: 10 członów, 2 obiekty, których konstruktory tworzą 4 referencje obiektów, oraz 9 par kinematycznych, każda z jednym parametrem artykulacji.

Listing 23.4. Budowanie łańcucha kinematycznego aplikacji

C

```
kl_linkage *ConstructMyLinkage ( AppData *ad )
{
#define SCF (1.0/3.0)
  kl_linkage *lkg;
  int        l[10], j[9];
  int        i;
  GLfloat    tra[16];

  if ( (lkg = kl_NewLinkage ( 2, 10, 4, 9, 9, (void*)ad )) ) {
    ad->linkage = lkg;
    for ( i = 0; i < 10; i++ )
      l[i] = kl_NewLink ( lkg );
    M4x4Scalef ( tra, SCF, SCF, SCF*4.0/3.0 );
    kl_NewObject ( lkg, 0, 3,  TEAPOT_NPOINTS, tra, (void*)&ad->bezp[0],
                   ConstructMyTeapot, KLTransformBP, KLPostprocessBP,
                   KLRedrawBezPatches, KLDeleteBezPatches );
    M4x4RotateXf ( tra, 0.5*PI );
    M4x4MScalef ( tra, 0.1, 0.1, 0.1 );
    kl_NewObject ( lkg, 0, 4, 49, tra, (void*)&ad->bezp[1],
                   ConstructMyTorus, KLTransformBP, KLPostprocessBP,
                   KLRedrawBezPatches, KLDeleteBezPatches );
    j[0] = kl_NewJoint ( lkg, l[0], l[1], KL_ART_ROT_Z, 0 );
    j[1] = kl_NewJoint ( lkg, l[1], l[2], KL_ART_ROT_Y, 1 );
    j[2] = kl_NewJoint ( lkg, l[2], l[3], KL_ART_ROT_Y, 2 );
    j[3] = kl_NewJoint ( lkg, l[3], l[4], KL_ART_ROT_Y, 3 );
    j[4] = kl_NewJoint ( lkg, l[2], l[5], KL_ART_ROT_Y, 4 );
    j[5] = kl_NewJoint ( lkg, l[5], l[6], KL_ART_ROT_Y, 5 );
    j[6] = kl_NewJoint ( lkg, l[2], l[7], KL_ART_ROT_Y, 6 );
    j[7] = kl_NewJoint ( lkg, l[7], l[8], KL_ART_ROT_Y, 7 );
    j[8] = kl_NewJoint ( lkg, l[8], l[9], KL_ART_ROT_Z, 8 );
    M4x4Translatef ( lkg->current_root_tr, 0.0, 0.0, -0.6 );
    M4x4Translatef ( tra, -0.43, 0.0, 0.92 );
    kl_SetJointFtr ( lkg, j[1], tra, true );
    M4x4Translatef ( tra, 0.78, 0.0, 0.59 );
    kl_SetJointFtr ( lkg, j[2], tra, true );
    M4x4Translatef ( tra, 0.78, 0.0, 0.91 );
    kl_SetJointFtr ( lkg, j[3], tra, true );
    M4x4Translatef ( tra, 0.6, 0.0, 1.0 );
    kl_SetJointFtr ( lkg, j[4], tra, true );
    M4x4Translatef ( tra, -0.6, 0.0, 1.0 );
    kl_SetJointFtr ( lkg, j[5], tra, true );
    M4x4Translatef ( tra, 0.52, 0.0, 0.97 );
    kl_SetJointFtr ( lkg, j[6], tra, false );
    M4x4Translatef ( tra, 0.52, 0.0, 0.0 );
    kl_SetJointFtr ( lkg, j[7], tra, false );
```

```
46:     glGenBuffers ( 1, &ad->lktrbuf );
47:     glBindBuffer ( GL_SHADER_STORAGE_BUFFER, ad->lktrbuf );
48:     glBufferData ( GL_SHADER_STORAGE_BUFFER, lkg->norefs*16*sizeof(GLfloat),
49:                    NULL, GL_DYNAMIC_DRAW );
50:   }
51:   else
52:     ExitOnError ( "ConstructMyLinkage" );
53:   return lkg;
54: #undef SCF
55: } /*ConstructMyLinkage*/
```

W liniach 13 i 17–18 są tworzone macierze E_0 i E_1 opisujące wstępne przekształcenia czajnika i torusa (zobacz podrozdz. 13.1); macierze te są przekazywane jako parametry wywołań procedury `kl_NewObject`, która konstruuje te obiekty. Pola `usrdata` struktur tych obiektów będą wskazywać elementy tablicy `bezp` w strukturze `*ad`, a ostatnie pięć parametrów to wskaźniki procedur, które stają się metodami obiektów. Tylko pierwsza metoda, tj. konstruktor, jest inna dla czajnika i dla torusa.

W liniach 22–30 są wprowadzane pary kinematyczne, a w liniach 31–45 dla poszczególnych par są konstruowane i przypisywane macierze F_j i B_j, którymi są „obłożone" macierze obrotów realizowanych przez pary.

W liniach 46–49 powstaje bufor dla bloku magazynowego `Tr` szadera obliczeniowego z listingu 23.1. Każdej referencji obiektu odpowiada jedna macierz przekształcenia, któremu mają być poddane wierzchołki wymienione w tej referencji. Macierz otrzymana w wyniku artykulacji będzie przesłana do tego bufora przez metodę `transform` obiektu, a później (gdy wszystkie te macierze znajdą się na swoich miejscach) metoda `postprocess` uruchomi ten szader, aby przekształcić wszystkie wierzchołki obiektu.

Listingi 23.5 i 23.6 przedstawiają procedury wywoływane przez `kl_NewObject` w celu inicjalizacji danych specyficznych dla czajnika i torusa. Procedury te są konstruktorami, które tworzą *wszystkie* dane opisujące czajnik i torus potrzebne do ich artykulacji i do rysowania.

Procedura `ConstructTheTeapot` w linii 18 tworzy zestaw płatów Béziera opisujący nieodkształcony czajnik. W linii 19 powstaje struktura opisująca czajnik odkształcony. Bę-

Listing 23.5. Konstruktor czajnika

C

```
1: char ConstructMyTeapot ( kl_linkage *lkg, kl_object *obj )
2: {
3:   const int r0[198] = {  0, .... ,305};
4:   const int r1[30] = {169, .... ,202};
5:   const int r2[62] = {203, .... ,268};
6:   const GLfloat MyColour[4] = { 1.0, 1.0, 1.0, 1.0 };
7:   const GLfloat diffr[4] = { 0.75, 0.65, 0.3, 1.0 };
8:   const GLfloat specr[4] = { 0.7, 0.7, 0.6, 1.0 };
9:   const GLfloat shn = 60.0, wa = 5.0, we = 5.0;
10:  const GLfloat txc[32][4] = {{-1.0,0.0,-1.0,0.0},....,{-1.0,0.0,-1.0,0.0}};
```

```
11:   AppData      *ad;
12:   KLBezPatches *bezp;
13:   GLuint       *cpi;
14:   int          on, rn, i;
15:
16:   ad = (AppData*)lkg->usrdata;
17:   bezp = (KLBezPatches*)obj->usrdata;
18:   bezp->bpatches[0] = ConstructTheTeapot ( MyColour );
19:   bezp->bpatches[1] = malloc ( sizeof(BezierPatchObjf) );
20:   cpi = malloc ( obj->nvert*sizeof(GLuint) );
21:   if ( bezp->bpatches[0] && bezp->bpatches[1] && cpi ) {
22:     on = obj - lkg->obj;
23:     bezp->ntr = -1;
24:     memset ( cpi, 0, obj->nvert*sizeof(GLuint) );
25:     rn = kl_NewObjRef ( lkg, 2, on, 198, NULL );
26:     for ( i = 0; i < 198; i++ ) cpi[r0[i]] = rn;
27:     rn = kl_NewObjRef ( lkg, 4, on,  30, NULL );
28:     for ( i = 0; i <  30; i++ ) cpi[r1[i]] = rn;
29:     rn = kl_NewObjRef ( lkg, 6, on,  62, NULL );
30:     for ( i = 0; i <  62; i++ ) cpi[r2[i]] = rn;
31:     bezp->tribuf = NewStorageBuffer (
32:                       obj->nvert*sizeof(GLuint), ad->artp.ctribp );
33:     glBufferSubData ( GL_SHADER_STORAGE_BUFFER,
34:                        0, obj->nvert*sizeof(GLuint), cpi );
35:     free ( cpi );
36:     bezp->mtn = SetupMaterial ( &ad->mat, -1, diffr, specr, shn, wa, we );
37:     if ( GenBezierPatchTextureBlock ( bezp->bpatches[0] ) )
38:       glBufferSubData ( GL_SHADER_STORAGE_BUFFER,
39:                         0, 32*4*sizeof(GLfloat), txc );
40:     else
41:       ExitOnError ( "ConstructMyTeapot 0" );
42:     SetBezierPatchTessLevel ( bezp->bpatches[0], ad->TessLevel );
43:     SetBezierPatchNVS ( bezp->bpatches[0], ad->BezNormals );
44:     memcpy ( bezp->bpatches[1], bezp->bpatches[0],
45:              sizeof(BezierPatchObjf) );
46:     glGenBuffers ( 1, &bezp->bpatches[1]->buf[1] );
47:     glBindBuffer ( GL_SHADER_STORAGE_BUFFER, bezp->bpatches[1]->buf[1] );
48:     glBufferData ( GL_SHADER_STORAGE_BUFFER,
49:                    TEAPOT_NPOINTS*3*sizeof(GLfloat), NULL, GL_DYNAMIC_DRAW );
50:     bezp->texture = LoadMyTextures ();
51:     ExitIfGLError ( "ConstructMyTeapot" );
52:   }
53:   else
54:     ExitOnError ( "ConstructMyTeapot 1" );
55:   return true;
56: } /*ConstructMyTeapot*/
```

dzie ona mieć wspólne z oryginalnym czajnikiem bloki magazynowe `BezPatch`, `CPIndices` i `BezPatchTexCoord` i własny, utworzony w liniach 46–49 blok `CPoints` z punktami kontrolnymi obliczonymi przez szader artykulacji. Zatem, przed utworzeniem tego bufora (w liniach 44–45) dane opisujące czajnik oryginalny (w tym identyfikatory utworzonych wcześniej buforów w pamięci GPU) są przepisywane do tej drugiej struktury.

W liniach 22–35 konstruktor tworzy referencje obiektu — czajnika. Dla każdej z nich jest podana lista indeksów punktów kontrolnych płata, w tablicy `r0`, `r1` lub `r2`. Numer obiektu jest obliczany w linii 22[1]. Referencje obiektu mają liczbę wierzchołków 0 i nie przechowują indeksów swoich wierzchołków w pamięci CPU. Zamiast tego jest używany bufor z tablicą numerów referencji — dla każdego wierzchołka (punktu kontrolnego) w tej tablicy jest numer referencji, do której ten wierzchołek należy, czyli numer przekształcenia, któremu szader obliczeniowy ma poddać ten wierzchołek. Pętle w liniach 26, 28 i 30 wpisują te numery do pomocniczej tablicy `cpi`, której zawartość jest w liniach 33–34 przesyłana do bufora z blokiem magazynowym `TrInd`. Aby szader brał numery przekształceń z tego bufora, pole `ntr` opisu czajnika otrzymuje wartość `-1`, która będzie nadana zmiennej jednolitej `trnum`.

Uwaga: Przed przeszukaniem listy referencji tablica jest `cpi` wypełniana zerami, bo nie wszystkie punkty podane w reprezentacji czajnika są wyliczone w referencjach[2].

Pozostałe instrukcje konstruktora czajnika mają na celu przygotowanie rysowania go. W linii 36 powstaje opis materiału czajnika, potrzebny w obliczeniach oświetlenia. W liniach 37–39 jest tworzony bufor, w którym zostają umieszczone współrzędne tekstury wierzchołków dziedziny poszczególnych płatów Béziera. Tekstura jest przygotowywana w linii 50. W liniach 42–43 pola w bloku zmiennych jednolitych `BezPatch` opisujące stopień rozdrobnienia oraz sposób określania wektora normalnego w obliczeniach oświetlenia otrzymują wartości początkowe (wcześniej nadane przez aplikację polom struktury `*ad`).

Pokazany na listingu 23.6 konstruktor torusa jest znacznie prostszy, bo torus ma tylko jedną referencję obiektu (wszystkie jego punkty kontrolne są poddawane temu samemu przekształceniu) i nie jest na niego nakładana tekstura. Ale podobnie jak czajnik, torus jest reprezentowany przez dwie struktury typu `BezPatchObjf`, mające wspólne bloki magazynowe z wyjątkiem bloku `CPoints` z tablicą punktów kontrolnych. Wartość 0 przypisana polu `tribuf` w linii 15 jest identyfikatorem pustym; przekształcanie torusa odbywa się bez używania bloku `TrInd`, ponieważ wszystkie jego wierzchołki są poddawane temu samemu przekształceniu. Numer tego przekształcenia, czyli numer referencji jest zapamiętywany w linii 16 zaraz po wprowadzeniu referencji do łańcucha. Instrukcja w linii 17 tworzy opis materiału torusa, a w liniach 20–25 powstaje reprezentacja torusa przekształconego. Przypisana w linii 26 liczba 0 jest w OpenGL-u także pustym identyfikatorem tekstury.

[1] W aplikacji jest jeden czajnik, który jest obiektem numer 0, ale tu pokazałem sposób, w jaki konstruktor przeznaczony dla wielu obiektów może obliczyć numer obiektu konstruowanego w bieżącym wywołaniu.

[2] Zobacz przypis na s. 394. Można oczywiście „wyczyścić" dane, ale zawsze warto tak napisać program, aby dobrze działał także dla danych „niewyczyszczonych". Dzięki wypełnieniu tablicy zerami, każdemu zbędnemu punktowi w tablicy `cptrind.trind` będzie odpowiadał indeks 0, zatem przekształcając je, szader nie będzie czytał macierzy przekształcenia spoza tablicy `cptr.tr`.

Listing 23.6. Konstruktor torusa

C

```
char ConstructMyTorus ( kl_linkage *lkg, kl_object *obj )
{
  GLfloat MyColour[4] = { 0.2, 0.3, 1.0, 1.0 };
  const GLfloat diffr[4] = { 0.0, 0.4, 1.0, 1.0 };
  const GLfloat specr[4] = { 0.7, 0.7, 0.7, 1.0 };
  const GLfloat shn = 20.0, wa = 2.0, we = 5.0;
  AppData      *ad;
  KLBezPatches *bezp;

  ad = (AppData*)lkg->usrdata;
  bezp = (KLBezPatches*)obj->usrdata;
  bezp->bpatches[0] = EnterTorus ( 1.0, 0.5, MyColour );
  bezp->bpatches[1] = malloc ( sizeof(BezierPatchObjf) );
  if ( bezp->bpatches[0] && bezp->bpatches[1] ) {
    bezp->tribuf = 0;
    bezp->ntr = kl_NewObjRef ( lkg, 9, obj - lkg->obj, 49, NULL );
    bezp->mtn = SetupMaterial ( &ad->mat, -1, diffr, specr, shn, wa, we );
    SetBezierPatchTessLevel ( bezp->bpatches[0], ad->TessLevel );
    SetBezierPatchNVS ( bezp->bpatches[0], ad->BezNormals );
    memcpy ( bezp->bpatches[1], bezp->bpatches[0],
             sizeof(BezierPatchObjf) );
    glGenBuffers ( 1, &bezp->bpatches[1]->buf[1] );
    glBindBuffer ( GL_SHADER_STORAGE_BUFFER, bezp->bpatches[1]->buf[1] );
    glBufferData ( GL_SHADER_STORAGE_BUFFER, 49*4*sizeof(GLfloat),
                   NULL, GL_DYNAMIC_DRAW );
    bezp->texture = 0;
    ExitIfGLError ( "ConstructMyTorus" );
  }
  else
    ExitOnError ( "ConstructMyTorus" );
  return true;
} /*ConstructMyTorus*/
```

Listing 23.7 przedstawia procedury, które są metodami `transform` i `postprocess` obiektu składającego się z płatów Béziera. Zadaniem pierwszej metody jest przesłanie do pamięci GPU (do tablicy w bloku `Tr`) obliczonej przez procedurę artykulacji macierzy przekształcenia, któremu mają być poddane punkty kontrolne wyliczone w pewnej (jednej) referencji obiektu. Przekształcenia dokonuje druga metoda, jednocześnie dla wszystkich punktów kontrolnych obiektu, posługując się szaderem obliczeniowym opisanym w poprzednim podrozdziale; metoda ta, kolejno dla wszystkich obiektów, zostaje wywołana przez procedurę `kl_Articulate` po umieszczeniu w pamięci GPU wszystkich macierzy przekształceń.

Procedura `KLPostprocessBP`, czyli metoda `postprocess` płatów Béziera przywiązuje bufory z danymi opisującymi płaty, nadaje wartości zmiennym jednolitym i wywołuje program z szaderem z listingu 23.1. Lokalna grupa robocza ma wymiary $1 \times 1 \times 1$. Globalna grupa

robocza jest jednowymiarowa, jej długość, równa liczbie wierzchołków obiektu, i pozostałe dwa wymiary (równe 1) są podane w linii 33 jako parametry makrodefinicji COMPUTE (listing 9.1), której rozwinięciem jest wywołanie kolejno procedury glDispatchCompute, która uruchamia szader obliczeniowy i procedury glMemoryBarrier, która czeka na zakończenie zapisywania wyników w pamięci GPU[3].

Listing 23.7. Metody transform i postprocess płatów Béziera

C

```
1: static void KLTransformBP ( kl_linkage *lkg, kl_object *obj,
2:                             int refn, GLfloat *tr, int nv, int *vn )
3: {
4:   AppData *ad;
5:
6:   ad = (AppData*)lkg->usrdata;
7:   glBindBuffer ( GL_SHADER_STORAGE_BUFFER, ad->lktrbuf );
8:   glBufferSubData ( GL_SHADER_STORAGE_BUFFER, refn*16*sizeof(GLfloat),
9:                     16*sizeof(GLfloat), tr );
10:  ExitIfGLError ( "KLTransformBP" );
11: } /*KLTransformBP*/
12:
13: static void KLPostprocessBP ( kl_linkage *lkg, kl_object *obj )
14: {
15:   AppData         *ad;
16:   KLArticulationProgram *prog;
17:   KLBezPatches    *bpobj;
18:   BezierPatchObjf **bp;
19:
20:   ad = (AppData*)lkg->usrdata;
21:   prog = &ad->artprog;
22:   bpobj = (KLBezPatches*)obj->usrdata;
23:   glUseProgram ( prog->progid );
24:   bp = (BezierPatchObjf**)bpobj->bpatches;
25:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, prog->ctrbp, ad->lktrbuf );
26:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, prog->cpibp, bp[0]->buf[1] );
27:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, prog->cpobp, bp[1]->buf[1] );
28:   if ( bpobj->ntr == -1 )
29:     glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, prog->ctribp,
30:                        bpobj->tribuf );
31:   glUniform1i ( prog->trnum_loc, bpobj->ntr );
32:   glUniform1i ( prog->dim_loc, bp[0]->dim );
33:   COMPUTE ( obj->nvert, 1, 1 )  /* listing 9.1 */
34:   ExitIfGLError ( "KLPostprocessBP" );
35: } /*KLPostprocessBP*/
```

[3] CPU i GPU działają równolegle, to jest jedna z tych sytuacji, gdy należy ich działania synchronizować.

Listing 23.8 przedstawia procedurę rysowania płatów Béziera i ich siatek kontrolnych. Sposób rysowania jest określony przez dane w strukturach, odpowiednio typu `AppData` i `KLBezPatches`, wskazywanych przez pola `usrdata` podanych jako parametry struktur łańcucha kinematycznego i obiektu. Przypisania w liniach 7 i 8 ułatwiają dostęp do tych danych.

Listing 23.8. Metoda `redraw` płatów Béziera

C

```
1: void KLRedrawBezPatches ( kl_linkage *lkg, kl_object *obj )
2: {
3:   AppData      *ad;
4:   KLBezPatches *bezp;
5:   GLint        cs;
6:
7:   ad = (AppData*)lkg->usrdata;
8:   bezp = (KLBezPatches*)obj->usrdata;
9:   if ( ad->skeleton )
10:    glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
11:  else
12:    glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
13:  if ( ad->final ) {
14:    glUseProgram ( ad->bprprog.program_id[1] );
15:    if ( (cs = ad->colour_source) == 2 ) {
16:      if ( bezp->bpatches[0]->buf[3] ) {
17:        BindBezPatchTextureBuffer ( bezp->bpatches[0] );
18:        glActiveTexture ( GL_TEXTURE0 );
19:        glBindTexture ( GL_TEXTURE_2D, bezp->texture );
20:      }
21:      else
22:        cs = 1;
23:    }
24:    glUniform1i ( ad->bprprog.ColourSourceLoc, cs );
25:    glUniform1i ( ad->bprprog.NormalSourceLoc, ad->normal_source );
26:    glUniform1i ( ad->bprprog.ModifyDepthLoc, ad->modify_depth );
27:    ChooseMaterial ( &ad->mat, bezp->mtn );
28:  }
29:  else
30:    glUseProgram ( ad->bprprog.program_id[0] );
31:  DrawBezierPatches ( bezp->bpatches[1] );
32:  if ( ad->cnet ) {
33:    glUseProgram ( ad->bprprog.program_id[2] );
34:    DrawBezierNets ( bezp->bpatches[1] );
35:  }
36: } /*KLRedrawBezPatches*/
```

W liniach 9–12 następuje wybór sposobu rysowania, przez wyświetlenie wypełnionych trójkątów powstałych przez rozdrabnianie płatów albo tylko krawędzi tych trójkątów.

Pole `ad->final` struktury typu `AppData` ma wartość `false`, gdy rysowanie ma na celu otrzymanie reprezentacji obszaru cienia, oraz `true`, gdy ma być wykonany obraz obiektów oświetlonych i z nałożoną teksturą. Odpowiedni program do rysowania płatów, zależnie od tego parametru, jest wybierany w linii 14 albo 30. Dla końcowego obrazu trzeba też przywiązać teksturę do nałożenia na czajnik.

W liniach 24–27 są nadawane wartości zmiennym jednolitym `ColourSource`, `NormalSource` i `ModifyDepth`. Podczas znajdowania obszaru cienia powyższe przygotowania są zbędne. Rysowanie płatów następuje po wykonaniu instrukcji w linii 31. W liniach 32–35 są, jeśli użytkownik to włączył, rysowane siatki kontrolne płatów — zawsze przy użyciu tego samego programu szaderów.

Listing 23.9. Destruktor obiektów — płatów Béziera

C

```
1: void KLDeleteBezPatches ( kl_linkage *lkg, kl_object *obj )
2: {
3:   KLBezPatches *bezp;
4:
5:   bezp = (KLBezPatches*)obj->usrdata;
6:   DeleteBezierPatches ( bezp->bpatches[0] );
7:   glDeleteBuffers ( 1, &bezp->bpatches[1]->buf[1] );
8:   if ( bezp->tribuf ) glDeleteBuffers ( 1, &bezp->tribuf );
9:   if ( bezp->texture ) glDeleteTextures ( 1, &bezp->texture );
10:  free ( bezp->bpatches[1] );
11: } /*KLDeleteBezPatches*/
```

Procedura na listingu 23.9 jest wywoływana przez procedurę likwidacji łańcucha kinematycznego. Najpierw zwalnia ona reprezentację oryginalnych (tj. nieprzekształconych) płatów Béziera. Reprezentacja płatów przekształconych współdzieli z nią wszystkie bufory magazynowe oprócz bufora ze współrzędnymi przekształconych punktów kontrolnych, dlatego bufor ten jest zwalniany osobno (w linii 7). W linii 8 jest zwalniany bufor z indeksami macierzy przekształceń wierzchołków czajnika (obiekt torusa nie ma takiego bufora), w linii 9 jest usuwana tekstura, a w linii 10 jest zwalniana pamięć zajmowana przez strukturę typu `BezPatchObjf`. Wszystkie bufory w pamięci GPU, których identyfikatory są zapisane w polach tej struktury, zostały już wcześniej zlikwidowane.

23.4. Zmiany w aplikacji

Listing 23.10 przedstawia procedurę inicjalizacji danych aplikacji. Procedura ta rozpoczyna działanie od wypełnienia zerami wskazywanej przez parametr zmiennej typu `AppData`, po czym kompiluje, łączy i przygotowuje do pracy programy szaderów, wywołując (przeniesione bez żadnych zmian) procedury `LoadBPShaders` i `LoadMirrorShaders` z listingu 22.6 oraz procedurę `LoadLinkageArticulationProgram` z listingu 23.3, tworzy bufory dla bloków zmiennych jednolitych z macierzami przekształceń i opisami źródeł światła, inicjalizuje zegar, ustawia jednostkową macierz przejścia od układu współrzędnych modelu do układu

świata, inicjalizuje kamerę, nadaje początkowe wartości przełączników, konstruuje lustro, wprowadza opis źródła światła i wreszcie konstruuje łańcuch kinematyczny i dokonuje jego pierwszej artykulacji.

Listing 23.10. Procedura `InitMyWorld`

C

```
1: void InitMyWorld ( int argc, char *argv[], int width, int height )
2: {
3:   memset ( &appdata, 0, sizeof(AppData) );
4:   LoadBPShaders ( &appdata.bprprog );
5:   LoadMirrorShaders ( appdata.miprog );
6:   LoadLinkageArticulationProgram ( &appdata.artprog );
7:   appdata.trans.trbuf = NewUniformTransBlock ();
8:   appdata.light.lsbuf = NewUniformLightBlock ();
9:   appdata.mat.matbuf = NewUniformMatBlock ();
10:  TimerInit ();
11:  M4x4Identf ( trans.mm );
12:  LoadMMatrix ( &appdata.trans, NULL );
13:  InitCamera ( &appdata, width, height );
14:  appdata.TessLevel = 10;
15:  appdata.BezNormals = GL_TRUE;
16:  appdata.cnet = appdata.skeleton = appdata.animate = false;
17:  appdata.shadows = true;
18:  ConstructMirror ( &appdata.mirror );
19:  InitLights ( &appdata );
20:  if ( !ConstructMyLinkage ( &appdata ) )
21:    ExitOnError ( "InitMyWorld" );
22:  ArticulateMyLinkage ( appdata.linkage );
23: } /*InitMyWorld*/
```

Macierz przejścia od układu współrzędnych modelu do układu świata (macierz modelu) jest (i pozostaje przez cały czas działania aplikacji) jednostkowa. Jej rolę przejmują macierze obliczone przez procedurę artykulacji łańcucha, przy czym przekształcaniem punktów do układu świata zajmuje się szader artykulacji; animacja czajnika jest dokonywana przez ten szader, a nie przez zmienianie macierzy modelu. Pozostawienie jej w obliczeniach wykonywanych przez GPU umożliwia wykorzystanie szaderów rysujących z poprzedniej wersji aplikacji bez żadnych przeróbek.

Procedura `ArticulateMyLinkage` (listing 23.11) oblicza wartości parametrów artykulacji, które są animowane, czyli mają co chwila nadawane nowe wartości. Pierwszy z tych parametrów jest kątem obrotu czajnika (tj. członu L_1 łańcucha kinematycznego wokół osi z członu L_0, która pokrywa się z osią z układu świata) i, zależnie od tego, ile razy użytkownik nacisnął klawisz spacji (powodując zmianę wartości pola `animate` z `false` na `true` lub z `true` na `false`), zmienia się w tempie $\pi/4$ radianów na sekundę lub pozostaje niezmieniony. Pozostałe parametry artykulacji przyjmują wartości funkcji, których argumentem jest czas od początku symulacji. Kod źródłowy podprogramów obliczających wartości tych

funkcji pominąłem, bo nie ma w nich niczego fascynującego, ale na rysunku 23.3 pokazałem wykresy. Po przypisaniu nowych wartości parametrów artykulacji łańcucha procedura artykulacji jest wywoływana w linii 21.

Listing 23.11. Procedura `ArticulateMyLinkage`

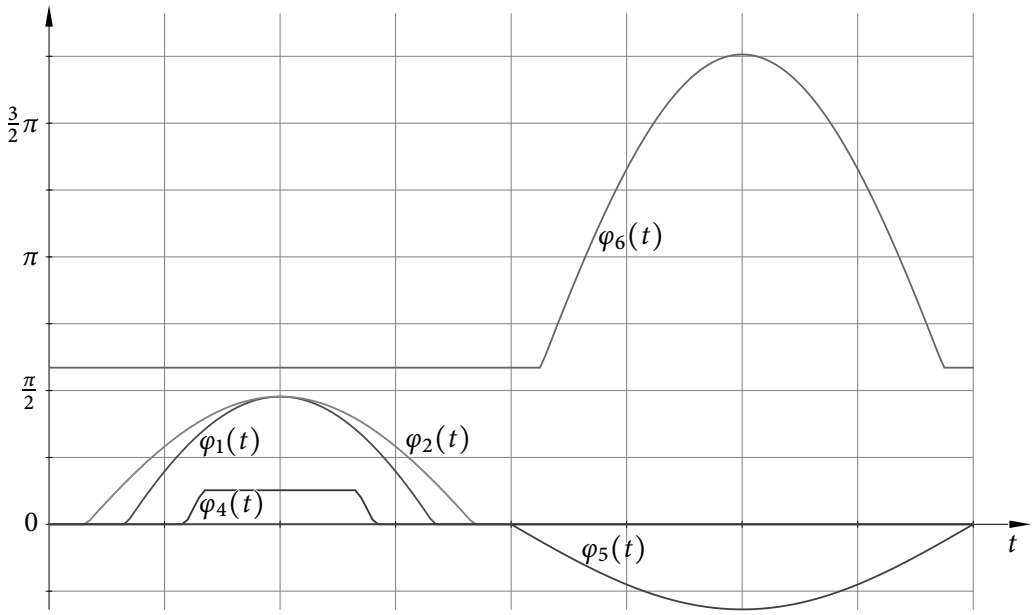
C

```
1: void ArticulateMyLinkage ( kl_linkage *lk )
2: {
3:   AppData *ad;
4:   double  dt, par[9];
5:
6:   ad = (AppData*)lk->usrdata;
7:   dt = TimerTocTic ();
8:   if ( ad->animate ) {
9:     if  ( (ad->teapot_rot_angle += ANGULAR_VELOCITY1 * dt) >= PI )
10:      ad->teapot_rot_angle -= 2.0*PI;
11:  }
12:  par[0] = ad->teapot_rot_angle;
13:  par[1] = TeapotRotAngle2 ( app_time );
14:  par[2] = SpoutAngle ( app_time );
15:  par[3] = -0.5*par[2];
16:  par[4] = LidAngle1 ( app_time );
17:  par[5] = LidAngle2 ( app_time );
18:  par[7] = -(par[6] = TorusRotAngle1 ( app_time ));
19:  par[8] = -2.0*PI*app_time;
20:  kl_SetArtParam ( lk, 0, 9, par );
21:  kl_Articulate ( lk );
22: } /*ArticulateMyLinkage*/
```

Funkcje opisujące parametry artykulacji są okresowe. Ich okres to 8 s, przy czym podziałka na osi czasu (poziomej) jest co jedną sekundę. Funkcje te (na podstawie eksperymentów) zostały dobrane tak, aby torus „wskakując do" i „wyskakując z" czajnika, nie wchodził w kolizję z pokrywką ani korpusem czajnika. Zwracam też uwagę, że parametr φ_3 jest równy $-\frac{1}{2}\varphi_2$, a ponadto $\varphi_7 = -\varphi_6$. Ostatnia równość zapewnia, że oś, wokół której obraca się torus, pozostaje równoległa do osi z członu L_2, z którym jest związany korpus czajnika.

Procedury rysowania sceny są pokazane na listingu 23.12. Scena składa się z czajnika i torusa, wbudowanych do łańcucha kinematycznego, oraz lustra, które jest osobnym obiektem. Można oczywiście przyczepić do łańcucha także lustro, ale pierwszym etapem rysowania sceny jest wykonanie obrazu, który ma być teksturą nałożoną na lustro, a wtedy trzeba by wykonać obraz sceny *z pominięciem* lustra. Potrzebny byłby wtedy dodatkowy przełącznik sterujący rysowaniem lustra. Czajnik i torus są zatem rysowane przez metody wywoływane przez procedurę `DrawMyLinkage`. Pierwszym jej parametrem jest wskaźnik struktury łańcucha kinematycznego, a drugi parametr wybiera rysowanie cieni albo rysowanie obrazu z oświetleniem.

Procedury `DrawSceneToShadows`, `DrawSceneToMirror` i `DrawSceneToWindow`, a także procedura `RedrawMyWorld`, która je wywołuje, są przeniesione z aplikacji 2G bez żadnych



Rysunek 23.3. Wykresy parametrów artykulacji w funkcji czasu

Listing 23.12. Procedura rysowania sceny i procedury animacji

C

```
1: void DrawMyLinkage ( AppData *ad, char final )
2: {
3:   ad->final = final;
4:   kl_Redraw ( ad->linkage );
5: } /*DrawMyLinkage*/
6:
7: void DrawScene ( AppData *ad, char final )
8: {
9:   DrawMyLinkage ( ad, final );
10: } /*DrawScene*/
11:
12: void RedrawMyWorld ( void )
13: {
14:   ArticulateMyLinkage ( appdata.linkage );
15:   glEnable ( GL_DEPTH_TEST );
16:   DrawSceneToShadows ( &appdata );
17:   DrawSceneToMirror ( &appdata );
18:   DrawSceneToWindow ( &appdata );
19: } /*RedrawMyWorld*/
20:
21: char MoveOn ( void )
22: {
```

```
23:   return true;
24: } /*MoveOn*/
```

zmian. Procedury te wywołują procedurę `DrawScene`, której jedyną instrukcją jest wywołanie procedury rysowania obiektów dołączonych do łańcucha kinematycznego. Gdyby elementów sceny było więcej (np. gdyby trzeba było narysować nieruchome ściany pomieszczenia i meble), to odpowiednie instrukcje powinny być dodane w tym miejscu.

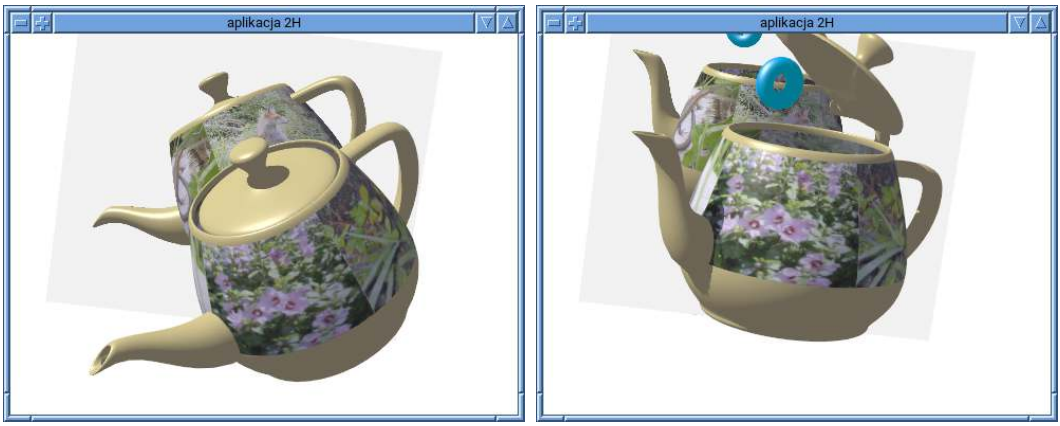
Animacja jest dokonywana przez wywoływanie co chwila procedury `MoveOn`, która tylko zawiadamia część okienkową aplikacji, że trzeba wykonać nowy obraz (zawsze trzeba). Artykulacja łańcucha kinematycznego jest przeprowadzana bezpośrednio przed przystąpieniem do wykonywania obrazu.

Procedurę `ProcessCharCommand` trzeba dostosować do zmienionej struktury `AppData`; zmodyfikowane instrukcje są pokazane na listingu 23.13.

Listing 23.13. Zmiany w procedurze `ProcessCharCommand`

C

```
1: char ProcessCharCommand ( char charcode )
2: {
3:   switch ( toupper ( charcode ) ) {
4: case '+':
5:     if ( appdata.TessLevel < MAX_TESS_LEVEL ) {
6:       SetBezierPatchTessLevel ( appdata.bezp[0].bpatches[0],
7:                                 ++appdata.TessLevel );
8:       SetBezierPatchTessLevel ( appdata.bezp[1].bpatches[0],
9:                                 appdata.TessLevel );
10:      return true;
11:    }
12:    else return false;
13: case '-':
14:    if ( appdata.TessLevel > MIN_TESS_LEVEL ) {
15:      SetBezierPatchTessLevel ( appdata.bezp[0].bpatches[0],
16:                                --appdata.TessLevel );
17:      SetBezierPatchTessLevel ( appdata.bezp[1].bpatches[0],
18:                                appdata.TessLevel );
19:      return true;
20:    }
21:    else return false;
22: case 'N':
23:    appdata.BezNormals = appdata.BezNormals == 0;
24:    SetBezierPatchNVS ( appdata.bezp[0].bpatches[0], appdata.BezNormals );
25:    SetBezierPatchNVS ( appdata.bezp[1].bpatches[0], appdata.BezNormals );
26:    return true;
27: ....
28: default:
29:    return false;
30:  }
31: } /*ProcessCharCommand*/
```



Rysunek 23.4. Okno aplikacji drugiej H

Na listingu 23.14 jest pokazana procedura sprzątania; w porównaniu z procedurą z poprzedniej wersji aplikacji ma ona do skasowania jeszcze jeden program szaderów. Reprezentacje płatów Béziera, bufory z opisami materiałów i tekstura nakładana na czajnik są likwidowane przez destruktor z listingu 23.9. Ponieważ moja implementacja łańcucha kinematycznego nie ma metody wirtualnej — destruktora dla całego łańcucha[4], w linii 14 jest osobno likwidowany bufor, w którym był blok magazynowy `Tr` szadera obliczeniowego. Bufory z przekształceniami oraz dane opisujące lustro są kasowane tak samo jak w aplikacji 2G.

Listing 23.14. Sprzątanie

C

```
1: void DeleteMyWorld ( void )
2: {
3:   int i;
4:
5:   glUseProgram ( 0 );
6:   for ( i = 0; i < 3; i++ )
7:     glDeleteProgram ( appdata.brprog.program_id[i] );
8:   for ( i = 0; i < 2; i++ )
9:     glDeleteProgram ( appdata.miprog[i] );
10:  DeleteLinkageArticulationProgram ( &appdata.artprog );
11:  glDeleteBuffers ( 1, &appdata.trans.trbuf );
12:  glDeleteBuffers ( 1, &appdata.light.lsbuf );
13:  glDeleteBuffers ( 1, &appdata.mat.matbuf );
14:  glDeleteBuffers ( 1, &appdata.lktrbuf );
15:  kl_DestroyLinkage ( appdata.linkage );
16:  DeleteMirror ( &appdata.mirror );
17:  DeleteShadowFBO ( &appdata.light );
18:  DeleteEmptyVAO ();
19: } /*DeleteMyWorld*/
```

[4] Entuzjaści programowania obiektowego w języku C++ zadbają o takie szczegóły w swoich implementacjach łańcuchów kinematycznych.

23.5. *Uzupełnienia — adaptacja stopnia rozdrobnienia płatów

Czajnik z Utah, tak jak wiele innych obiektów wartych narysowania, składa się z płatów Béziera o różnych wielkościach i kształtach. Stopnie rozdrobnienia dziedzin poszczególnych płatów warto dobierać indywidualnie, dostosowując je do wielkości płatów na obrazie; takie postępowanie jest nazywane **adaptacją**, a w grafice komputerowej mówi się o wybieraniu **poziomu szczegółowości** (*level of detail, LOD*).

Liczby określające stopnie rozdrobnienia płatów może podać aplikacja działająca na CPU, choć znacznie lepszym rozwiązaniem jest dobranie ich przy użyciu GPU, przy czym (chyba) najlepiej jest użyć do tego szaderów obliczeniowych. W tym celu reprezentację zespołu płatów rozszerzymy o dodatkowy blok magazynowy. Rolą szadera sterowania rozdrabnianiem będzie tylko przypisanie elementom tablic `gl_TessLevelOuter` i `gl_TessLevelInner`, określającym podział dziedziny płata (zobacz rys. 15.3), liczb wziętych z tego bloku.

Algorytm zrealizowany przez opisane dalej szadery obliczeniowe opiera się na hipotezie (sprawdzonej na przykładzie czajnika, ale w ogólności być może błędnej), że można otrzymać zadowalające wyniki, przyjmując parametry rozdrabniania na podstawie kilku leżących na płacie krzywych stałego parametru oraz ich reprezentacji Béziera[5]; dokładniej, dla płata stopnia (n, m) będziemy badać krzywe stałego parametru $u = \frac{i}{n}$ dla $i = 0, \ldots, n$ oraz krzywe stałego parametru $v = \frac{j}{m}$ dla $j = 0, \ldots, m$.

Zadanie znajdowania parametrów rozdrabniania płata podzielimy na dwa etapy realizowane przez osobne szadery. Wyniki pierwszego etapu — 6 liczb — zależą tylko od kształtu krzywych. Korzystając z tych wyników, drugi szader obliczy parametry rozdrabniania płata odpowiednie do wielkości jego obrazu końcowego. Zatem, dla każdego płata potrzebujemy przechować 12 liczb. Szader z listingu 23.15 w linii 15 oblicza miejsce, z którego szader ma pobrać parametry dla płata określonego przez numer instancji.

Krzywe stałego parametru $u = 0$, $u = 1$, $v = 0$ i $v = 1$, z których składa się brzeg płata, często pokrywają się z krzywymi brzegowymi innych płatów; na przykład płaty w modelu czajnika mają wiele takich wspólnych krzywych. Aby zapewnić ciągłość połączenia trójkątowych przybliżeń płatów mających wspólną krzywą brzegową, parametr rozdrobnienia każdego boku dziedziny płata (tj. liczbę wpisywaną do tablicy `gl_TessLevelOuter`) trzeba określić *tylko* na podstawie krzywej brzegowej odpowiadającej temu bokowi.

Listing 23.15. Szader sterowania rozdrabnianiem

GLSL

```
1: #version 430 core
2:
3: layout(vertices=1) out;
4:
5: layout(location=0) in int instance[];
6: layout(location=0) out int inst[];
7:
```

[5] Tej hipotezy nie da się matematycznie udowodnić ani obalić, bo znaczenie słowa „zadowalające" nie jest precyzyjnie określone. Na pewno można wymyślić lepszy algorytm.

```
8: layout(std430,binding=4) buffer BezPatchTessParams { float tsp[]; };
9:
10: void main ( void )
11: {
12:   int k;
13:
14:   if ( gl_InvocationID == 0 ) {
15:     k = 12*instance[gl_InvocationID];
16:     gl_TessLevelOuter[0] = tsp[k+3];  gl_TessLevelOuter[1] = tsp[k];
17:     gl_TessLevelOuter[2] = tsp[k+5];  gl_TessLevelOuter[3] = tsp[k+2];
18:     gl_TessLevelInner[0] = tsp[k+1];  gl_TessLevelInner[1] = tsp[k+4];
19:     inst[gl_InvocationID] = instance[gl_InvocationID];
20:   }
21: } /*main*/
```

Pierwszą czynnością wykonywaną przez procedury `main` obu szaderów obliczeniowych jest wybranie punktów kontrolnych płata określonego przez numer wątku w grupie roboczej i wpisanie ich do roboczej tablicy `ctp`. Szadery będą prowadzić obliczenia dla płatów wymiernych, jeśli więc mamy wielomianowe płaty płaskie, to do współrzędnych x, y punktów kontrolnych są dołączane współrzędne jednorodne $Z = 0$, $W = 1$, a jeśli płaty są wielomianowe, to ich punkty kontrolne otrzymują współrzędną wagową 1. Pierwszy szader (listing 23.16) natychmiast oblicza współrzędne jednorodne tych punktów w układzie świata[6], a drugi (listing 23.17) od razu przechodzi do układu obserwatora.

Procedura `FindShapeParams`, wywoływana przez procedurę `main` pierwszego szadera, w kolejnych dwóch pętlach (w liniach 77–87 i 89–99) znajduje łamane kontrolne (reprezentacje Béziera) krzywych stałego parametru u i v, wpisując ich wierzchołki do tablicy `bc`, a potem przepisując je do tablicy `bcp` (z tej tablicy procedura `BCHorner4f` odczytuje punkty kontrolne krzywej Béziera, której punkt ma obliczyć; w ten sposób szader unika wielokrotnego kopiowania tablicy będącej parametrem procedur).

Mając punkty kontrolne krzywej stałego parametru (oznaczmy ją literą $\boldsymbol{c}$), szader wywołuje procedurę `AngBCDens` znajdującą najmniejszą taką liczbę d, że łamana, której wierzchołkami są punkty $\boldsymbol{c}_i = \boldsymbol{c}(\frac{i}{d})$, $i = 0, \ldots, d$, dostatecznie dobrze przybliża krzywą. Przyjąłem,

Listing 23.16. Pierwszy szader obliczeniowy adaptacyjnego rozdrabniania płatów Beziera

GLSL

```
1: #version 450 core
2:
3: #define MAX_DEG 10
4: #define MIN_TESS_PARAM  3.0
5: #define MAX_TESS_PARAM 32.0
6:
```

[6] Przejście od układu modelu do układu świata nie musi być podobieństwem geometrycznym; na przykład czajnik jest poddawany nierównomiernemu skalowaniu. Natomiast przyjąłem założenie, że przejście od układu świata do układu obserwatora jest izometrią.

```
7: layout(local_size_x=1) in;
8:
9: layout(std430,binding=0) buffer CPoints { float cp[]; } cp;
10: layout(std430,binding=1) buffer CPIndices { int cpi[]; } cpi;
11: layout(std430,binding=2) buffer BezPatch { .... } bezp;  /* listing 15.3 */
12: layout(std430,binding=4) buffer BezPatchTessParams { float tsp[]; };
13:
14: uniform TransBlock { .... } trb;  /* listing 19.14 */
15:
16: uniform float maxangle = 15.0*0.017453292;  /* 15 stopni */
17:
18: int  ncp;
19: vec4 ctp[(MAX_DEG+1)*(MAX_DEG+1)], bcp[MAX_DEG+1];
20:
21: vec4 BCHorner4f ( int n, float t )
22: {
23:   int   i, b;
24:   float s, d;
25:   vec4  q;
26:
27:   s = 1.0-t;  d = t;  b = n;
28:   q = bcp[0];
29:   for ( i = 1; i <= n; i++ ) {
30:     q = s*q + (float(b)*d)*bcp[i];
31:     d *= t;  b = (b*(n-i))/(i+1);
32:   }
33:   return q;
34: } /*BCHorner4f*/
35:
36: vec3 BCHornerR3f ( int n, float t )
37: {
38:   vec4 q;
39:
40:   q = BCHorner4f ( n, t );
41:   return q.xyz / q.w;
42: } /*BCHornerR3f*/
43:
44: int AngBCDens ( int n )
45: {
46:   int   d, i;
47:   float t, l1, l2, ang;
48:   vec3  v1, v2, p1, p2;
49:   bool  z;
50:
51:   for ( d = max(n, int(MIN_TESS_PARAM)); d <= int(MAX_TESS_PARAM); d++ ) {
52:     p1 = BCHornerR3f ( n, 1.0/float(d) );
53:     v1 = p1 - bcp[0].xyz / bcp[0].w;  z = (l1 = length ( v1 )) == 0.0;
```

```
    for ( i = 2, ang = 0.0;  i <= d;  i++ ) {
      p2 = BCHornerR3f ( n, float(i)/float(d) );
      v2 = p2 - p1;  l2 = length ( v2 );
      if ( l2 > 0.0 ) {
        z = false;
        if ( l1 > 0.0 &&
             (ang = acos ( dot ( v1, v2 ) / (l1*l2) )) > maxangle )
          break;
        p1 = p2;  v1 = v2;  l1 = l2;
      }
    }
    if ( z || ang <= maxangle )
      break;
  }
  return d;
} /*AngBCDens*/

void FindShapeParams ( int l )
{
  int   i, j, k, m, d, e;
  float t;
  vec4  bc[MAX_DEG+1];

  for ( i = e = 0;  i <= bezp.udeg;  i++ ) {
    t = float(i)/float(bezp.udeg);
    for ( j = 0; j <= bezp.vdeg; j++ ) {
      for ( m = 0, k = j;  m <= bezp.udeg;  m++, k += bezp.vdeg+1 )
        bcp[m] = ctp[k];
      bc[j] = BCHorner4f ( bezp.udeg, t );
    }
    for ( j = 0; j <= bezp.vdeg; j++ ) bcp[j] = bc[j];
    if ( (d = AngBCDens ( bezp.vdeg )) > e ) e = d;
    if ( i == 0 ) tsp[l+3] = float(d);
  }
  tsp[l+4] = float(e);  tsp[l+5] = float(d);
  for ( j = e = 0;  j <= bezp.vdeg;  j++ ) {
    t = float(j)/float(bezp.vdeg);
    for ( i = k = 0;  i <= bezp.udeg;  i++ ) {
      for ( m = 0;  m <= bezp.vdeg;  m++, k++ )
        bcp[m] = ctp[k];
      bc[i] = BCHorner4f ( bezp.vdeg, t );
    }
    for ( i = 0; i <= bezp.udeg; i++ ) bcp[i] = bc[i];
    if ( (d = AngBCDens ( bezp.udeg )) > e ) e = d;
    if ( j == 0 ) tsp[l] = float(d);
  }
  tsp[l+1] = float(e);  tsp[l+2] = float(d);
```

```
} /*FindShapeParams*/

void main ( void )
{
  int inst, i, j, k, l, i0;
#define FINDCPOINTS(DIM,Z,W) { \
    if ( bezp.use_ind ) { \
      i0 = inst*bezp.stride_p; \
      for ( i = k = 0;  i <= bezp.udeg;  i++ ) { \
        for ( j = 0;  j <= bezp.vdeg;  j++, k++ ) { \
          l = DIM*cpi.cpi[i0+k]; \
          ctp[k] = trb.mm * vec4 ( cp.cp[l], cp.cp[l+1], Z, W ); \
    } } } \
    else { \
      i0 = (inst / bezp.nq)*bezp.stride_p+(inst % bezp.nq)*bezp.stride_q; \
      for ( i = k = 0;  i <= bezp.udeg;  i++ ) { \
        for ( j = 0, l = i0+i*bezp.stride_u;  j <= bezp.vdeg; \
              j++, k++, l += bezp.stride_v ) { \
          ctp[k] = trb.mm * vec4 ( cp.cp[l], cp.cp[l+1], Z, W ); \
    } } } }

  inst = int(gl_GlobalInvocationID.x);
  ncp = (bezp.udeg+1)*(bezp.vdeg+1);
  switch ( bezp.dim ) {
case 2: FINDCPOINTS ( 2, 0.0, 1.0 )  break;
case 3: FINDCPOINTS ( 3, cp.cp[l+2], 1.0 )  break;
case 4: FINDCPOINTS ( 4, cp.cp[l+2], cp.cp[l+3] )  break;
  }
  FindShapeParams ( 12*inst+6 );
} /*main*/
```

że jest tak wtedy, gdy kąty między kolejnymi odcinkami łamanej są dostatecznie bliskie kąta półpełnego: ich miary mają być większe niż $\pi - \alpha$, gdzie α jest wartością zmiennej jednolitej `alpha`. Podana w jej deklaracji (domyślna) wartość 15° zapewnia, że gładka krzywa zamknięta będzie przybliżona przez co najmniej dwudziestoczterokąt; oczywiście, aplikacja może na polecenie użytkownika zmieniać tę wartość, co ułatwi eksperymentalne dobieranie tego parametru. Podczas badania łamanej odcinki o długości 0 są pomijane. Pętla jest przerywana także wtedy, gdy łamana ma wszystkie odcinki o długości 0 — takiej krzywej $\boldsymbol{c}$ nie trzeba rozdrabniać na wiele części.

W kolejnych elementach tablicy `tsp` szader zapamiętuje pożądany stopień rozdrobnienia (tj. znalezioną liczbę odcinków łamanej przybliżającej) krzywej $u = 0$, maksymalny stopień rozdrobnienia *wszystkich* badanych krzywych $u = \text{const}$, następnie krzywej $u = 1$, krzywej $v = 0$, maksymalny stopień rozdrobnienia krzywych $v = \text{const}$ i krzywej $v = 1$. Zauważmy, że liczby te (wynik pierwszego etapu obliczeń) nie zależą od wielkości ani od położenia płata.

Drugi szader obliczeniowy (listing 23.17) ma za zadanie obliczyć parametry rozdrobnienia krzywych stałego parametru, biorąc pod uwagę wyniki pierwszego etapu i wielkość ob-

razu płata (w pikselach). Po zapamiętaniu w tablicy `ctp` punktów kontrolnych płata szader sprawdza, czy płat znajduje się całkowicie poza bryłą widzenia; jeśli tak, to parametry rozdrobnienia otrzymują najmniejszą dopuszczalną wartość (tj. 3), aby oszczędzić GPU pracy podczas rysowania (płat zostanie przybliżony przez minimalną liczbę trójkątów, z których wszystkie zostaną odrzucone w etapie obcinania), przy czym, z powodów opisanych dalej, aplikacja może wyłączyć to sprawdzenie, nadając zmiennej jednolitej `clip` wartość `false`. Sprawdzenia dokonuje procedura `ClipOff`, wywoływana w liniach 149–154 kolejno dla lewej, prawej, dolnej, górnej, przedniej i tylnej ściany bryły widzenia[7].

Listing 23.17. Drugi szader obliczeniowy adaptacyjnego rozdrabniania płatów Beziera

GLSL

```
#version 450 core

#define MAX_DEG 10
#define MIN_TESS_PARAM 3.0
#define MAX_TESS_PARAM 32.0

layout(local_size_x=1) in;

layout(std430,binding=0) buffer CPoints { float cp[]; } cp;
layout(std430,binding=1) buffer CPIndices { int cpi[]; } cpi;
layout(std430,binding=2) buffer BezPatch { .... } bezp;
layout(std430,binding=4) buffer BezPatchTessParams { float tsp[]; };

uniform TransBlock { .... } trb;  /* listing 19.14 */

uniform float minedgelgt = 3.0, maxedgelgt = 20.0;
uniform bool clip = true;

int   ncp;
vec4  ctp[(MAX_DEG+1)*(MAX_DEG+1)], bcp[MAX_DEG+1];
float lgtu[3], lgtv[3];

bool ClipOff ( int inst, vec4 pnv )
{
  int i, j;

  for ( i = 0; i < ncp; i++ )
    if ( dot ( ctp[i], pnv ) > 0.0 )
      return false;
  for ( i = 0, j = 12*inst;  i < 6;  i++, j++ )
    tsp[j] = MIN_TESS_PARAM;
  return true;
} /*ClipOff*/

```

[7] Drugi parametr procedury jest jednorodną reprezentacją ściany ostrosłupa widzenia — szader jest napisany tylko dla rzutowania perspektywicznego; zobacz też p. 19.8.7.

```
35: bool NonNegativeZ ( int n, float zw[MAX_DEG+1] )
36: {
37:   int i;
38:
39:   for ( i = 0; i <= n; i++ )
40:     if ( zw[i] >= 0.0 )
41:       return true;
42:   return false;
43: } /*NonNegativeZ*/
44:
45: vec4 BCHorner4f ( int n, float t ) { .... } /* listing 23.16 */
46:
47: float PolylineLength ( int n, vec2 p[MAX_DEG+1] )
48: {
49:   int   i;
50:   float l;
51:
52:   l = distance ( p[0], p[1] );
53:   for ( i = 1; i < n; i++ )
54:     l += distance ( p[i], p[i+1] );
55:   return l;
56: } /*PolylineLength*/
57:
58: vec2 ProjectP4 ( vec4 p )
59: {
60:   vec4 q;
61:
62:   q = trb.pm * p;
63:   return vec2 ( trb.viewport.x + 0.5*trb.viewport.z*(q.x/q.w+1.0),
64:                 trb.viewport.y + 0.5*trb.viewport.w*(q.y/q.w+1.0) );
65: } /*ProjectP4*/
66:
67: void FindCPolyLengths ( int udeg, int vdeg )
68: {
69:   int   i, j, k, m;
70:   float t, lgt, zw[MAX_DEG+1];
71:   vec4  p;
72:   vec2  q[MAX_DEG+1];
73: #define StoreLGT(DEG,I,TAB) { \
74:     if ( NonNegativeZ ( DEG, zw ) ) lgt = 1.0e6; \
75:     else lgt = PolylineLength ( DEG, q ); \
76:     if ( I == 0 ) TAB[0] = lgt; \
77:     else if ( I == DEG ) TAB[2] = lgt; \
78:     if ( lgt > TAB[1] ) TAB[1] = lgt; }
79:
80:   lgtu[0] = lgtu[1] = lgtu[2] = lgtv[0] = lgtv[1] = lgtv[2] = 0.0;
81:   for ( i = 0; i <= bezp.udeg; i++ ) {
```

```
82:     t = float(i)/float(bezp.udeg);
83:     for ( j = 0; j <= bezp.vdeg; j++ ) {
84:       for ( m = 0, k = j;  m <= bezp.udeg;  m++, k += bezp.vdeg+1 )
85:         bcp[m] = ctp[k];
86:       q[j] = ProjectP4 ( p = BCHorner4f ( bezp.udeg, t ) );
87:       zw[j] = max ( p.z, -p.w );
88:     }
89:     StoreLGT ( bezp.udeg, i, lgtv )
90:   }
91:   for ( j = 0; j <= bezp.vdeg; j++ ) {
92:     t = float(j)/float(bezp.vdeg);
93:     for ( i = k = 0;  i <= bezp.udeg;  i++ ) {
94:       for ( m = 0;  m <= bezp.vdeg;  m++, k++ )
95:         bcp[m] = ctp[k];
96:       q[i] = ProjectP4 ( p = BCHorner4f ( bezp.vdeg, t ) );
97:       zw[j] = max ( p.z, -p.w );
98:     }
99:     StoreLGT ( bezp.vdeg, j, lgtu )
100:  }
101: } /*FindCPolyLengths*/
102:
103: float TessParam ( int deg, float d, float l )
104: {
105:   if ( l/d < minedgelgt )
106:     d = l/minedgelgt;
107:   l = max ( l/maxedgelgt, d );
108:   if ( l > MAX_TESS_PARAM ) l = MAX_TESS_PARAM;
109:   else {
110:     if ( l < deg ) l = deg;
111:     if ( l < MIN_TESS_PARAM ) l = MIN_TESS_PARAM;
112:   }
113:   return l;
114: } /*TessParam*/
115:
116: void FindTessParams ( int inst )
117: {
118:   int i, j;
119:
120:   for ( i = 0, j = 12*inst;  i < 3;  i++, j++ )
121:     tsp[j] = TessParam ( bezp.udeg, tsp[j+6], lgtu[i] );
122:   for ( i = 0; i < 3;  i++, j++ )
123:     tsp[j] = TessParam ( bezp.vdeg, tsp[j+6], lgtv[i] );
124: } /*FindTessParams*/
125:
126: void main ( void )
127: {
128:   int inst, i, j, k, l, i0;
```

```
129:   .... /* to samo, co w liniach 104-118 na listingu 23.16 */ \
130:   .... /* z dwoma wyjątkami */ \
131: #define FINDCPOINTS(DIM,Z,W) { \
132:   if ( bezp.use_ind ) { \
133:      .... \
134:         ctp[k] = trb.vm * (trb.mm * vec4 ( cp.cp[l], cp.cp[l+1], Z, W )); \
135:   .... } \
136:   else { \
137:      .... \
138:         ctp[k] = trb.vm * (trb.mm * vec4 ( cp.cp[l], cp.cp[l+1], Z, W )); \
139:   .... } }
140:
141:   inst = int(gl_GlobalInvocationID.x);
142:   ncp = (bezp.udeg+1)*(bezp.vdeg+1);
143:   switch ( bezp.dim ) {
144: case 2: FINDCPOINTS ( 2, 0.0, 1.0 )  break;
145: case 3: FINDCPOINTS ( 3, cp.cp[l+2], 1.0 )  break;
146: case 4: FINDCPOINTS ( 4, cp.cp[l+2], cp.cp[l+3] )  break;
147:   }
148:   if ( clip ) {
149:     if ( ClipOff ( inst, vec4( trb.near, 0.0, trb.left, 0.0) ) )    return;
150:     if ( ClipOff ( inst, vec4( -trb.near, 0.0, -trb.right, 0.0) ) ) return;
151:     if ( ClipOff ( inst, vec4( 0.0, trb.near, trb.bottom, 0.0) ) )  return;
152:     if ( ClipOff ( inst, vec4( 0.0, -trb.near, -trb.top, 0.0) ) )   return;
153:     if ( ClipOff ( inst, vec4( 0.0, 0.0, -1.0, -trb.near) ) )       return;
154:     if ( ClipOff ( inst, vec4( 0.0, 0.0, 1.0, trb.far) ) )          return;
155:   }
156:   FindCPolyLengths ( bezp.udeg, bezp.vdeg );
157:   FindTessParams ( inst );
158: } /*main*/
```

Zadaniem procedury `FindCPolyLengths` jest znalezienie obrazów łamanych kontrolnych rozważanych tu krzywych stałego parametru płatów i obliczenie (i zapamiętanie w tablicach `lgtu` i `lgtv`) ich długości w pikselach; końcowe obliczenie parametrów rozdrabniania opiera się na fakcie, że obrazem w rzucie perspektywicznym krzywej Béziera jest wymierna krzywa Béziera reprezentowana przez obraz łamanej kontrolnej, a długość krzywej Béziera[8] nie jest większa niż długość jej łamanej kontrolnej.

Jest tu dodatkowy problem: jeśli część figury geometrycznej (płata lub krzywej) znajduje się „za plecami" obserwatora (czyli w półprzestrzeni $z \geqslant 0$ w układzie obserwatora), to obraz tej figury w rzucie perspektywicznym na płaszczyznę jest nieograniczony. Jeśli więc pewne punkty kontrolne krzywej mają współrzędną $z \geqslant 0$, to pozostaje przyjąć maksymalny stopień rozdrobnienia takiej krzywej, nawet jeśli większość trójkątów otrzymanych w wyniku rozdrabniania płata będzie poza obrazem.

[8] zarówno wielomianowej, jak i wymiernej, jeśli wagi wszystkich punktów kontrolnych są dodatnie

Zatem: procedura `FindCPolyLengths` w pętlach w liniach 81–90 i 91–100 bada kolejne krzywe stałego parametru u = const i v = const. Dla każdej krzywej znajduje (w liniach 83–86 albo 93–96) punkty kontrolne, które poddaje rzutowaniu (przy użyciu procedury `ProjectP4`, realizującej wzory (6.1)). Obrazy punktów kontrolnych są zapamiętywane w tablicy `q`. W tablicy `zw` są pamiętane liczby $\max\{Z_i, -W_i\}$, gdzie Z_i i W_i to ostatnie dwie współrzędne jednorodne i-tego punktu kontrolnego.

Pierwsza instrukcja w makrodefinicji `StoreLGT` dokonuje sprawdzenia, czy wszystkie współrzędne wagowe punktów kontrolnych są dodatnie, a współrzędne z są ujemne; jeśli nie, to zamiast długości obrazu łamanej zmiennej `lgt` jest przypisywana duża liczba rzeczywista (linia 74). W przeciwnym razie procedura `PolylineLength` oblicza długość obrazu łamanej. Długości obrazów łamanych dla krzywych stałego parametru $u = 0$ i $u = 1$ są przypisywane zmienym `lgtv[0]` i `lgtv[2]`, a w zmiennej `lgtv[1]` jest pamiętana największa z długości krzywych u = const; podobnie w tablicy `lgtu` są pamiętane długości krzywych v = const.

Końcowe obliczenie parametrów rozdrobnienia wykonuje procedura `TessParam` (linie 103–114), wywoływana przez procedurę `FindTessParams`. Podstawą tego obliczenia jest stopień n płata ze względu na parametr u albo v, wartość d parametru zaproponowana przez pierwszy szader obliczeniowy i długość l rzutu łamanej kontrolnej odpowiedniej krzywej stałego parametru na płaszczyznę obrazu. Jeśli l/d, tj. oszacowanie średniej długości odcinka, jest mniejsze niż dolny próg tolerancji (wartość zmiennej jednolitej `minedgelgt`, domyślnie 3 piksele), to w linii 106 parametr d jest zmniejszany tak, aby otrzymane na obrazie odcinki nie były za krótkie. Wartość zmiennej `maxedgelgt` (domyślnie 20 pikseli) określa maksymalną długość odcinków, jakie chcemy otrzymać. Wartość parametru podziału obliczona w linii 107 jest następnie sprowadzana do przedziału dopuszczalnego (co najmniej n, ale nie mniej niż 3 i nie więcej niż 32). Procedura `FindTessParams` zapamiętuje parametr podziału w tablicy `tsp` (w bloku magazynowym `BezPatchTessParams`), skąd odczyta je szader sterowania rozdrabnianiem.

Aby narysować jedną klatkę animacji, aplikacja 2H rysuje scenę trzykrotnie, w trzech różnych rzutach: w celu otrzymania reprezentacji obszaru cienia oraz wykonania obrazu przedmiotów odbitych w lustrze i obrazu końcowego. Adaptacyjne dobieranie parametrów rozdrabniania może dla każdego z tych obrazów dać zupełnie inne wyniki, co nie wpłynie dobrze na jakość obrazu końcowego. Parametry rozdrabniania najlepiej jest więc dostosować tylko do obrazu końcowego i użyć ich we wszystkich trzech przypadkach — dlatego (moim zdaniem) parametry rozdrobnienia powinny być obliczane przez osobne programy z szaderami obliczeniowymi, a nie przez szader sterowania rozdrabnianiem korzystający z opisu rzutowania przyjętego dla obrazu wykonywanego w danej chwili.

Może się zdarzyć, że pewne płaty znajdą się poza bryłą widzenia obserwatora, ale ich odbicia w lustrze będą widoczne. Testy wykonane przez procedurę `ClipOff` spowodowałyby nadanie parametrom rozdrabniania wartości minimalnej, dlatego w aplikacji 2H trzeba było z nich zrezygnować, nadając zmiennej jednolitej `clip` (listing 23.17, linia 17) wartość `false`.

Nie opisuję tu wszystkich zmian aplikacji potrzebnych do użycia w niej przedstawionych tu szaderów, zostawiając Czytelnikom pole do działania. Konstruując reprezentacje czajnika

i torusa (lub innego zespołu płatów Béziera), należy utworzyć dodatkowy bufor dla parametrów podziału płatów. Pierwszy szader obliczeniowy powinien być wywołany po każdym odkształceniu płatów (np. po wygięciu dziobka czajnika), a drugi szader obliczeniowy trzeba wywołać bezpośrednio przed rysowaniem, jeśli płaty były odkształcone (i pierwszy szader wykonał swoją pracę) lub zmianie uległo rzutowanie (zmieniło się położenie obserwatora, kształt bryły widzenia lub wymiary okna aplikacji).

23.6. Ćwiczenia

1. Zmodyfikuj aplikację tak, aby obraz czajnika odbitego w lustrze powtarzał ruchy czajnika z opóźnieniem o na przykład 1/4 sekundy.
2. Nawiązując do pierwszego ćwiczenia w podrozdziale 17.4, napisz szader obliczeniowy obliczający punkty kontrolne iloczynu sferycznego dwóch krzywych reprezentowanych przez łamane. Zastosuj w nim dwuwymiarową grupę roboczą. Użyj tego szadera zamiast procedury `EnterRSphericalProduct` z listingu 17.1 do utworzenia reprezentacji torusa w aplikacji.
3. Zbuduj łańcuch kinematyczny z dowolnymi innymi obiektami (np. sztywnymi bryłami wielościennymi) i zmień aplikację tak, aby wyświetlała animację tych obiektów.
4. *Dostosuj opisane w podrozdziale 23.5 szadery rysowania wielu instancji *zespołu* płatów, na przykład w celu narysowania sceny takiej jak na rysunku 15.7, z adaptacyjnym (indywidualnym) dobieraniem parametrów rozdrabniania do poszczególnych powtórzeń każdego płata w zespole.
5. *Szadery z listingów 23.16 i 23.17 wykonują obliczenia dla płata określonego przez numer wątku sekwencyjnie, badając krzywe stałego parametru po kolei. Zrównoleglij te obliczenia, wprowadzając dwuwymiarową grupę roboczą.
6. *Przeczytaj przypis na s. 596 i wymyśl (i wypróbuj) lepszy algorytm[9].

[9] Znaczenie słowa „lepszy” też nie jest precyzyjnie określone.

24

Aplikacja druga I

W kolejnej wersji aplikacji z dziobka czajnika będzie wylatywać para wodna, która będzie się skraplać i ponownie parować. Efekt kłębienia się mgły nad dziobkiem osiągniemy przy użyciu **układu cząsteczek**. Każda cząsteczka reprezentuje jedną kroplę wody poruszającą się w powietrzu, którego obecność powoduje, że początkowa prędkość cząsteczki zmienia się, dążąc do prędkości unoszenia spowodowanej konwekcją. Dodatkowo cząsteczki będą doznawać losowych przyspieszeń. Oczywiście, modelowania ruchu nie oprzemy na prawach fizyki, a tylko oprogramujemy czysto empiryczne wzory z parametrami, których dobieranie umożliwi osiągnięcie dość dobrego efektu na obrazach. Obliczenia ruchu cząsteczek będzie wykonywać szader obliczeniowy; bardzo się tu przyda masywna równoległość obliczeń na GPU.

24.1. Równania ruchu i reguły zachowania cząsteczek

Ruch pojedynczej cząsteczki jest opisany układem równań różniczkowych

$$\begin{cases} \boldsymbol{v}'(t) = \boldsymbol{a}(t), \\ \boldsymbol{p}'(t) = \boldsymbol{v}(t). \end{cases}$$

Zmienna t jest tożsama z czasem. Funkcja wektorowa $\boldsymbol{p}$ opisuje (w układzie współrzędnych świata) położenie cząsteczki, jej pochodna, czyli funkcja $\boldsymbol{v}$, reprezentuje prędkość, pochodna tejże, $\boldsymbol{a}$, jest przyspieszeniem. W każdej chwili przyspieszenie jest jawnie zadane. Dla takiego układu równań trzeba podać **warunki początkowe**, czyli położenie $\boldsymbol{p}_0$ i prędkość $\boldsymbol{v}_0$ w chwili t_0, w której cząsteczka została wystrzelona z czajnika.

Ponieważ dokładność symulacji zachowania cząsteczek *w tym zastosowaniu* nie jest priorytetem, użyjemy bardzo prostej metody całkowania powyższych równań. Szader obliczeniowy jest wywoływany co pewien czas i otrzymuje informację, ile czasu upłynęło od jego poprzedniego wywołania. Czas ten (czyli **krok czasowy**[1]) oznaczymy literą h. Mając daną

[1]Czas między kolejnymi wywołaniami może się zmieniać. Zależy to od mocy procesora graficznego, który,

prędkość cząsteczki $\boldsymbol{v}_{i-1}$ i jej położenie $\boldsymbol{p}_{i-1}$ w chwili $t_{i-1} = t_i - h$, szader obliczy prędkość $\boldsymbol{v}_i$ i położenie $\boldsymbol{p}_i$ w chwili t_i na podstawie wzorów

$$\begin{cases} \boldsymbol{v}_i = B(\boldsymbol{v}_{i-1} + h\boldsymbol{a}_i), \\ \boldsymbol{p}_i = \boldsymbol{p}_{i-1} + \frac{1}{2}h(\boldsymbol{v}_{i-1} + \boldsymbol{v}_i). \end{cases} \tag{24.1}$$

Zakładamy, że przyspieszenie $\boldsymbol{a}_i$ w przedziale czasowym $[t_{i-1}, t_i]$ jest stałe, zatem zmiana położenia cząsteczki jest iloczynem łatwej do obliczenia prędkości średniej i kroku czasowego h. Macierz B wprowadza opisaną dalej korektę prędkości w pobliżu przeszkody. Teraz sztuka polega na wymyśleniu sposobu określania przyspieszenia cząsteczki. Przedstawiając swoje rozwiązanie, chciałbym zachęcić Czytelników do samodzielnych poszukiwań i eksperymentów.

Do obliczenia przyspieszenia przyjąłem wzór

$$\boldsymbol{a}_i = \frac{1}{h}\left(\boldsymbol{v}_{\text{conv}} + \mathrm{e}^{-h/C}(\boldsymbol{v}_{i-1} - \boldsymbol{v}_{\text{conv}})\right) + a\boldsymbol{r}_{i,k}. \tag{24.2}$$

Pierwszy składnik po prawej stronie jest związany z konwekcją — ponieważ para ma znacznie wyższą temperaturę niż otaczające czajnik powietrze, ma też mniejszą gęstość, wskutek czego jest wypierana do góry. Wektor $\boldsymbol{v}_{\text{conv}}$ opisuje pewną docelową prędkość kropel mgły przed końcowym wyparowaniem. Składnik $\frac{1}{h}\left(\boldsymbol{v}_{\text{conv}} + \mathrm{e}^{-h/C}(\boldsymbol{v}_{i-1} - \boldsymbol{v}_{\text{conv}})\right)$ opisuje tłumienie ruchu kropli poruszającej się z inną prędkością. Stała czasowa C opisuje jego intensywność; zwracam uwagę, że ten efekt powinien zależeć (i zależy) od czasu tłumienia w sposób wykładniczy.

Wektor $\boldsymbol{r}_{i,k}$ ma długość co najwyżej 1 i jego współrzędne są pseudolosowe. Obecność we wzorze na przyspieszenie składnika z tym wektorem powoduje cokolwiek chaotyczne zaburzenia ruchu cząsteczki, wpływając na widoczne na obrazach zachowanie symulowanej mgły. Liczba k jest numerem cząsteczki w układzie. Dla każdej cząsteczki i w każdym kroku całkowania ruchu wybierany jest inny wektor $\boldsymbol{r}_{i,k}$. Czynnik a określa maksymalną długość drugiego składnika przyspieszenia.

Gdyby we wzorze (24.1) nie było macierzy B lub gdyby była to zawsze macierz jednostkowa, to cząsteczki mogłyby przenikać przez obiekty. Aby temu zapobiec, przyjąłem, że jeśli cząsteczka znajduje się w pobliżu lustra i zbliża się do niego, należy odpowiednio skorygować jej prędkość. Informacje o wielkości i położeniu lustra zakodowałem „na twardo" w treści szadera[2]; lustro leży w płaszczyźnie $x = 1.5$ (w układzie świata) i jest prostokątem: jego punkty mają współrzędne $y \in [-1.2, 1.2]$, $z \in [-1, 1]$. Jeśli położenie cząsteczki, $\boldsymbol{p}_{i-1} = [x_{i-1}, y_{i-1}, z_{i-1}]^T$, jest punktem prostopadłościanu $[1.3, 1.7] \times [-1.2, 1.2] \times [-1, 1]$ (czyli cząsteczka jest w odległości nie większej niż $d = 0.2$ od lustra) i współrzędna x wektora będącego wartością wyrażenia w nawiasie w pierwszym wzorze (24.1) ma znak przeciwny niż

generując kolejne klatki animacji, może, ale nie musi, nadążać z obliczeniami z częstotliwością odświeżania ekranu (typowo 60 Hz).

[2] Przyznaję, że to nie jest przykład bardzo eleganckiego programowania.

$x_{i-1} - 1.5$, to

$$B = \begin{bmatrix} |x_{i-1} - 1.5|^{0.1} & 0 & 0 \\ 0 & |x_{i-1} - 1.5|^{-0.05} & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Korekta współrzędnej x wektora prędkości hamuje ruch cząsteczki w stronę lustra, a skalowanie współrzędnej y powoduje, że cząsteczki „rozchodzą się" na boki.

Po opisaniu ruchu jednej cząsteczki pora określić zachowanie całego układu. Każda cząsteczka oprócz bieżącego położenia i prędkości ma atrybut interpretowany jako „wiek". Z każdym krokiem całkowania ruchu cząsteczki jej wiek jest powiększany o składnik h/T, gdzie T wyznacza okres aktywności cząsteczki w sekundach. Wiek wpływa na wygląd cząsteczki na obrazie. Zakładam, że po opuszczeniu dziobka czajnika (z wiekiem 0) przezroczysty obłok pary skrapla się, przez co powstaje kropla mgły, która później podlega ponownemu parowaniu — gdy cała kropla wyparuje, co następuje po osiągnięciu wieku 1, cząsteczka znika. Dokładniej, znika kropla, cząsteczka zaś wraca do puli cząsteczek nieaktywnych, z której może być wyemitowana ponownie.

Układ składa się z N cząsteczek, przy czym w aplikacji wypróbowałem $N = 2^{20}$. Cząsteczki są ponumerowane od 0 do $N-1$. Początkowy wiek cząsteczki o numerze k jest równy $\frac{k}{N} - 1$. Cząsteczki o ujemnym wieku (początkowo wszystkie) są nieaktywne; uaktywnienie cząsteczki następuje, gdy po zwiększeniu o kolejny składnik h/T jej wiek staje się dodatni. Wtedy następuje nadanie cząsteczce prędkości i położenia początkowego, a podczas kolejnych wywołań szadera będzie miało miejsce całkowanie ruchu. Jeśli w danym kroku wiek cząsteczki miałby osiągnąć lub przekroczyć 1, jest on zmniejszany o 1 — nowy wiek jest ujemny i cząsteczka będzie użyta do udawania nowej kropli mgły.

Sposób obliczania początkowego położenia i prędkości cząsteczki jest następujący: szader zna położenia $\boldsymbol{p}_{i-1,0}$ i $\boldsymbol{p}_{i,0}$ środka wylotu z dziobka czajnika i wektory określające kierunek wylatywania pary $\boldsymbol{v}_{i-1,0}$ i $\boldsymbol{v}_{i,0}$ odpowiednio w chwilach t_{i-1} i t_i. Cząsteczka w wieku w opuściła dziobek w chwili $t_i - w$ (przy czym warunki początkowe są nadawane, gdy $w < h = t_i - t_{i-1}$). Animowany ruch dziobka czajnika jest zbyt szybki, aby można go było pominąć[3], dlatego szader dokonuje interpolacji, tj. oblicza punkt i wektor

$$\boldsymbol{p}_{w,0} = (1 - w/h)\boldsymbol{p}_{i-1,0} + w/h\boldsymbol{p}_{i,0}, \qquad \boldsymbol{v}_{w,0} = (1 - w/h)\boldsymbol{v}_{i-1,0} + w/h\boldsymbol{v}_{i,0}.$$

Początkowe położenie cząsteczki jest przyjmowane w punkcie $\boldsymbol{p}_{w,0} + d\boldsymbol{r}_{k,0}$, a jej początkowa prędkość to $v_0(\boldsymbol{v}_{w,0} + u\boldsymbol{r}_{k,1})$. Wektory $\boldsymbol{r}_{k,0}$ i $\boldsymbol{r}_{k,1}$ są brane z puli wektorów pseudolosowych szadera. Parametr d określa promień obszaru (kuli), w którym cząsteczki są emitowane, a parametr u odpowiada za rozrzut prędkości początkowych. Parametr v_0 jest (zmieniającym się w czasie, tj. modulowanym przez aplikację) czynnikiem prędkości początkowej.

[3] Określenie warunków początkowych na podstawie punktu $\boldsymbol{p}_{i,0}$ zamiast obliczonego niżej $\boldsymbol{p}_{w,0}$, tj. uwzględnienie tylko położenia wylotu dziobka w chwili t_i ma ten skutek, że mgła wylatuje z szybko poruszającego się dziobka oddzielnymi obłoczkami, które wyglądają niezbyt realistycznie. Proszę zmodyfikować szader z listingu 24.1 i samemu się przekonać.

24.2. Szadery układu cząsteczek

Symulacja układu cząsteczek zgodnie z regułami opisanymi wyżej jest dokonywana przez szader zamieszczony na listingu 24.1. Blok zmiennych jednolitych `PartSys` zawiera parametry układu cząsteczek i zmienne takie jak `iter` (numer iteracji), `dt` (krok czasowy h), `vel` (czynnik prędkości v_0) oraz punkty `pos0`, `pos1` ($\boldsymbol{p}_{i-1,0}$, $\boldsymbol{p}_{i,0}$) i wektory `vel0` i `vel1` ($\boldsymbol{v}_{i-1,0}$, $\boldsymbol{v}_{i,0}$) o wartościach nadawanych przez aplikację przed każdym wywołaniem szadera.

Biblioteka GLSL-a nie zawiera generatora liczb losowych (ani pseudolosowych). Dlatego aplikacja na początku tworzy zbiór (pulę) wektorów o pseudolosowych współrzędnych, rozmieszczonych (jeśli można tak powiedzieć) z rozkładem jednostajnym w kuli jednostkowej. Wektory te musi wygenerować podczas przygotowania do pracy szaderów układu cząsteczek odpowiednia (opisana dalej) procedura działająca na CPU. Wektory te są przechowywane w tablicy `rvpool.rv` umieszczonej w buforze magazynowym `RandomVecPool`. Długość tej i pozostałych tablic (położeń i prędkości cząsteczek), będąca liczbą cząsteczek N, jest podana w zmiennej jednolitej `ps.tablength`.

W buforach magazynowych `PartPos` i `PartVel` są tablice położeń i prędkości cząsteczek. Pola `xyz` k-tego elementu tablicy `pos.pos` zawierają współrzędne kartezjańskie położenia, a pole `w` jest używane do przechowywania wieku k-tej cząsteczki. Na początku działania szadera w tablicach jest wiek, położenie i prędkość cząsteczki w chwili t_{i-1}; zadaniem szadera jest dodanie odpowiednich przyrostów i wpisanie do tablicy wieku, położenia i prędkości w chwili t_i. Numer cząsteczki jest ustalany w linii 24, jest to współrzędna x indeksu wywołania szadera w (jednowymiarowej) globalnej grupie roboczej. W linii 26 zmiennej `w0` jest przypisywany poprzedni wiek cząsteczki (tj. wiek w chwili t_{i-1}), a w linii 27 szader oblicza wiek w chwili t_i. Jeśli ten wiek jest większy niż 1, to cząsteczka wraca do puli cząsteczek nieaktywnych; wartości zmiennych `w0` i `w1` są zmniejszane o 1.

Listing 24.1. Szader obliczeniowy układu cząsteczek

GLSL

```
1: #version 430 core
2:
3: layout(local_size_x=1) in;
4:
5: layout(binding=0) buffer RandomVecPool { vec4 rv[]; } rvpool;
6: layout(binding=1) buffer PartPos { vec4 pos[]; } pos;
7: layout(binding=2) buffer PartVel { vec3 vel[]; } vel;
8:
9: uniform PartSys {
10:     int   tablength, iter;
11:     float dt, Tpart, vel, TC, acc, divv, divp;
12:     vec3  pos0, pos1, vel0, vel1, conv;
13:   } ps;
14:
15: const float xx = 1.5, ymin = -1.2, ymax = 1.2, zmin = -1.0, zmax = 1.0,
16:             d = 0.2;
```

```
17:
18: void main ( void )
19: {
20:   vec3  v0, v, p, u, t;
21:   float w0, w1, a, e;
22:   int   k;
23:
24:   k = int ( gl_GlobalInvocationID.x );
25:   p = pos.pos[k].xyz;
26:   w0 = pos.pos[k].w;
27:   w1 = w0 + ps.dt/ps.Tpart;
28:   if ( w1 > 1.0 )
29:     { w0 -= 1.0;  w1 -= 1.0; }
30:   if ( w0 < 0.0 && w1 >= 0.0 ) {
31:     a = -w0/ps.dt;
32:     v = ps.vel*mix ( ps.vel0, ps.vel1, a ) +
33:         ps.divv*rvpool.rv[(k+1) % ps.tablength].xyz;
34:     p = mix ( ps.pos0, ps.pos1, a ) + ps.divp*rvpool.rv[k].xyz;
35:   }
36:   else {
37:     v0 = vel.vel[k];
38:     v = ps.conv + exp ( -ps.dt/ps.TC ) * (v0 - ps.conv) +
39:              ps.dt*ps.acc*rvpool.rv[(ps.iter+k) % ps.tablength].xyz;
40:     if ( abs ( a = p.x - xx ) < d &&
41:          p.y >= ymin && p.y <= ymax && p.z >= zmin && p.z <= zmax ) {
42:       if ( a > 0.0 != v.x > 0.0 ) {
43:         a = abs ( a ) / d;
44:         v.x *= pow ( a, 0.1 );
45:         v.y /= pow ( a, 0.05 );
46:       }
47:     }
48:     p.xyz += 0.5*ps.dt*(v0+v);
49:   }
50:   vel.vel[k] = v;
51:   pos.pos[k] = vec4 ( p, w1 );
52: } /*main*/
```

Jeśli poprzedni i obecny wiek cząsteczki mają różne znaki, to instrukcje w liniach 31–34 nadają cząsteczce początkowe położenie i prędkość. Obliczona w linii 31 wartość zmiennej a (z przedziału $[0,1]$) służy do interpolacji punktów $\boldsymbol{p}_{i-1,0}$, $\boldsymbol{p}_{i,0}$ i wektorów $\boldsymbol{v}_{i-1,0}$, $\boldsymbol{v}_{i,0}$ (interpolacji dokonuje funkcja `mix`). Zmienna jednolita `ps.vel` przechowuje czynnik prędkości v_0. Wartość zmiennej `ps.divv` jest czynnikiem u, a wartość zmiennej `ps.divp` jest czynnikiem d w podanych wcześniej wzorach na warunki początkowe. Wektor $\boldsymbol{r}_{k,0}$ jest brany z k-tego miejsca tablicy z pulą wektorów pseudolosowych, natomiast wektor $\boldsymbol{r}_{k,1}$ z miejsca $(k+1) \bmod N$.[4]

[4] W układzie miliona cząsteczek nikt nie zauważy, że to „losowanie" nie jest uczciwe.

Całkowanie ruchu cząsteczki na podstawie wzorów (24.1) wykonują instrukcje w liniach 37–48. Instrukcja w liniach 38–39 oblicza wektor będący wartością wyrażenia w nawiasie we wzorze (24.2). W liniach 40–47 następuje korekta prędkości; jeśli cząsteczka znajduje się blisko lustra, to współrzędne x i y obliczonego wektora są mnożone przez odpowiednie czynniki (wzięte z diagonali macierzy B). W linii 48 szader oblicza nowe położenie cząsteczki. W liniach 50 i 51 nowa prędkość i położenie są zapisywane w tablicach. Razem z położeniem zapamiętywany jest aktualny wiek cząsteczki.

Szader wierzchołków używany do rysowania cząsteczek jest bardzo prosty (listing 24.2). Etap pobierania wierzchołków podaje na jego wejście wektor wzięty z bufora, który był widziany przez szader obliczeniowy jako bufor magazynowy `PartPos`. Pierwsze trzy składowe tego wektora są współrzędnymi kartezjańskimi położenia cząsteczki w układzie świata, a czwarta składowa zawiera wiek (**uwaga:** to nie jest wagowa współrzędna jednorodna położenia). Szader oblicza położenie cząsteczki w układzie kostki standardowej za pomocą tych samych macierzy w bloku `TransBlock` co podczas rysowania „zwykłych" obiektów (położenie cząsteczki jest podane w układzie świata, więc pomijamy macierz `trb.mm`).

Listing 24.2. Szader wierzchołków układu cząsteczek

GLSL

```
1: #version 430 core
2:
3: layout(location=0) in vec4 vert;
4:
5: out float age;
6:
7: uniform TransBlock {
8:   mat4 mm, mmti, vm, pm, vpm;
9:   vec4 eyepos;
10: } trb;
11:
12: void main ( void )
13: {
14:   age = vert.w;
15:   gl_Position = trb.vpm * vec4 ( vert.xyz, 1.0 );
16: } /*main*/
```

Szader fragmentów pokazany na listingu 24.3 jest też bardzo prosty. Cząsteczka jest wyświetlana jako kropka o wielkości jednego piksela. Jej kolor jest jasny szary (aplikacja może przypisać przechowującej kolor zmiennej jednolitej `PartColour` wartość inną niż podana w deklaracji). Kolor cząsteczki jest ustalony, natomiast z wiekiem cząsteczki zmienia się jej przezroczystość.

Wiek aktywnej cząsteczki (podany w zmiennej `age`) jest liczbą z przedziału $[0,1]$. Jeśli wiek jest ujemny, to szader pomija rysowanie, wykonując instrukcję `discard`. W przeciwnym razie cząsteczka będzie narysowana w ustalonym kolorze ze składową alfa, której wartość rośnie z wiekiem od zera do wartości maksymalnej, a następnie maleje do zera, gdy

cząsteczka osiąga wiek 1. Ustawiane przez aplikację parametry rysowania cząsteczek sprawią, że składowa alfa będzie zinterpretowana jako nieprzezroczystość cząsteczki. Na początku aktywności (przed skropleniem) i na końcu (po wyparowaniu) cząsteczki są przezroczyste.

Listing 24.3. Szader fragmentów układu cząsteczek

GLSL

```
#version 430 core

layout(location=0) out vec4 colour;

in float age;

uniform vec4 PartColour = vec4(0.95,0.95,0.95,0.7);

void main ( void )
{
  float t;

  if ( age < 0.0 )
    discard;
  else {
    t = 1.0-age;
    colour = vec4 ( PartColour.xyz, PartColour.w*t*t*t*age );
  }
} /*main*/
```

Listing 24.4 przedstawia definicję struktury użytej do opakowania (składających się z szaderów opisanych wyżej) programów, z których pierwszy dokonuje symulacji układu cząsteczek, a drugi je rysuje, oraz procedury kompilacji i likwidacji tych programów. Nie ma już potrzeby szczegółowego komentowania, tak jak na początku kursu, instrukcji wywołujących procedury, które czytają szadery, kompilują je i łączą w programy. W tablicy `psofs` są zapamiętywane przesunięcia pól w bloku zmiennych jednolitych `PartSys`, a w polach `psbp`, `rvbp`, `posbp`, i `velbp` numery punktów dowiązania w celach `GL_UNIFORM_BUFFER` i `GL_SHADER_STORAGE_BUFFER` tego bloku i bloków magazynowych używanych przez program składający się z szadera pokazanego na listingu 24.1. Pole `pcloc` służy do zapamiętania położenia zmiennej jednolitej `PartColour`, której wartość określa kolor rysowanych cząsteczek.

Listing 24.4. Procedury przygotowania i likwidacji programów dla układu cząsteczek

C

```
typedef struct {
    GLuint program_id[2];
    GLint  psofs[14], pcloc, psbsize;
    GLuint psbp, rvpbp, posbp, velbp;
  } PartSysPrograms;

```

```
7: void LoadParticleShaders ( PartSysPrograms *psprog )
8: {
9:   static const char *filename[] =
10:     { "app2i0.comp.glsl", "app2i0.vert.glsl", "app2i0.frag.glsl" };
11:   static const GLchar *SBNames[] =
12:     { "RandomVecPool", "PartPos", "PartVel" };
13:   static const GLchar *UPNames[] =
14:     { "PartSys", "PartSys.tablength", "PartSys.iter", "PartSys.dt",
15:       "PartSys.Tpart", "PartSys.vel", "PartSys.TC", "PartSys.acc",
16:       "PartSys.divv", "PartSys.divp", "PartSys.pos0", "PartSys.pos1",
17:       "PartSys.vel0", "PartSys.vel1", "PartSys.conv" };
18:   static const GLchar PCName[] = "PartColour";
19:   GLuint shader_id[3];
20:   GLint  size;
21:   int    i;
22:
23:   shader_id[0] = CompileShaderFiles ( GL_COMPUTE_SHADER, 1, &filename[0] );
24:   psprog->program_id[0] = LinkShaderProgram ( 1, &shader_id[0], "part0" );
25:   shader_id[1] = CompileShaderFiles ( GL_VERTEX_SHADER, 1, &filename[1] );
26:   shader_id[2] = CompileShaderFiles ( GL_FRAGMENT_SHADER, 1, &filename[2]);
27:   psprog->program_id[1] = LinkShaderProgram ( 2, &shader_id[1], "part1" );
28:   GetAccessToUniformBlock ( psprog->program_id[0], 14, &UPNames[0],
29:                             &psprog->psbsize, psprog->psofs, &psprog->psbp );
30:   GetAccessToStorageBlock ( psprog->program_id[0], 0, &SBNames[0],
31:                             &size, NULL, &psprog->rvpbp );
32:   GetAccessToStorageBlock ( psprog->program_id[0], 0, &SBNames[1],
33:                             &size, NULL, &psprog->posbp );
34:   GetAccessToStorageBlock ( psprog->program_id[0], 0, &SBNames[2],
35:                             &size, NULL, &psprog->velbp );
36:   psprog->pcloc = glGetUniformLocation ( psprog->program_id[1], PCName );
37:   AttachUniformTransBlockToBP ( psprog->program_id[1] );
38:   for ( i = 0; i < 3; i++ )
39:     glDeleteShader ( shader_id[i] );
40:   ExitIfGLError ( "LoadParticleShaders" );
41: } /*LoadParticleShaders*/
42:
43: void DeleteParticleShaders ( PartSysPrograms *psprog )
44: {
45:   int i;
46:
47:   glUseProgram ( 0 );
48:   for ( i = 0; i < 2; i++ )
49:     glDeleteProgram ( psprog->program_id[i] );
50: } /*DeleteParticleShaders*/
```

24.3. Generatory liczb i wektorów pseudolosowych

Jak wspomniałem, biblioteka procedur GLSL-a nie zawiera generatora liczb losowych, w związku z czym odpowiednią pulę takich liczb, w razie potrzeby, musi wygenerować aplikacja. Procedura `RandomFloat` na listingu 24.5 realizuje algorytm wzięty z programu przykładowego opisanego w książce [24]. W kolejnych wywołaniach procedura ta generuje elementy ciągu liczb pseudolosowych o rozkładzie jednostajnym na odcinku $[0,1)$.[5]

Podstawą prostych generatorów liczb pseudolosowych jest zmienna statyczna zwana **ziarnem** (*seed*), której wartość za każdym razem jest poddawana przekształceniu — jej wartość jest mnożona przez pewną stałą (i ewentualnie powiększana o inną stałą), przy czym w razie nadmiaru stałopozycyjnego brane są najmniej znaczące 32 bity wyniku. W ten sposób powstają pseudolosowe liczby całkowite ze zbioru $\{0,\ldots,2^{32}-1\}$.

Sposób przetwarzania wartości ziarna na liczbę zmiennopozycyjną opiera się bezpośrednio na standardzie IEEE-754. Znormalizowana liczba pojedynczej precyzji (32-bitowa) jest dana wzorem

$$x = (-1)^s 2^{c-127}(1+m),$$

w którym bit znaku s jest na najbardziej znaczącej pozycji, kolejne 8 bitów reprezentuje cechę $c \in \{1,\ldots,254\}$, a mantysa m jest ułamkiem z przedziału $[0,1)$ reprezentowanym przez najmniej znaczące 23 bity. W linii 8 zmienna `tmp` otrzymuje wartość powstałą przez zastosowanie operacji xor do poprzesuwanych bitów ziarna. Przesunięcie o 9 w prawo produkuje ciąg 23 bitów, który staje się mantysą. Do tego dołączana jest cecha 127 (napisana w linii 9 liczba `0x3F800000` jest równa `127 << 23`), a bit znaku ma wartość 0. W ten sposób powstaje liczba zmiennopozycyjna z przedziału $[1,2)$, której mantysa ma pseudolosowe bity. Liczba przekazywana przez instrukcję w linii 10 jest o 1 mniejsza.

Listing 24.5. Procedury `RandomFloat` i `RandomVector`

C

```
1: GLfloat RandomFloat ( void )
2: {
3:   float               res;
4:   unsigned int        tmp;
5:   static unsigned int seed = 0xFFFF0C59;
6:
7:   seed *= 16807;  /* zawsze nieparzyste, czyli nigdy nie 0 */
8:   tmp = seed ^ (seed >> 4) ^ (seed << 15);
9:   *((unsigned int*)&res) = (tmp >> 9) | 0x3F800000;
10:  return res - 1.0;
11: } /*RandomFloat*/
12:
13: static unsigned int cnt = 0;
14:
```

[5] Ciąg ten jest okresowy, o bardzo długim okresie.

```
15: void RandomVector ( GLfloat rv[4] )
16: {
17:   int i;
18:
19:   for ( ; ; cnt++ ) {
20:     for ( i = 0; i < 3; i++ )
21:       rv[i] = 2.0*RandomFloat() - 1.0;
22:     if ( V3DotProductf ( rv, rv ) <= 1.0 ) {
23:       rv[3] = 0.0;
24:       return;
25:     }
26:   }
27: } /*RandomVector*/
```

Zadaniem procedury `RandomVector` jest wygenerowanie wektora (pseudo)losowego z rozkładem jednostajnym w kuli jednostkowej. W liniach 20–21 elementom tablicy są przypisywane liczby pseudolosowe z przedziału $[-1, 1]$, tak więc powstaje wektor, który leży w kostce standardowej. Jeśli jego długość nie przekracza 1, to następuje powrót, a jeśli jest większa, to wektor jest odrzucany i procedura generuje kolejny wektor[6]. Ponieważ objętość kuli jednostkowej $(4\pi/3)$ jest nieco większa niż połowa objętości kostki standardowej, średnio odrzucana jest nieco mniej niż połowa wektorów. Zmienna `cnt` jest licznikiem odrzuconych wektorów — jego końcową wartość można dla zaspokojenia ciekawości porównać z liczbą otrzymanych wektorów leżących w kuli jednostkowej.

24.4. Przygotowanie, symulacja i rysowanie układu cząsteczek

Układ cząsteczek, jako element animacji, będzie obiektem dołączonym do łańcucha kinematycznego. Przedstawione w podrozdziale 24.5 metody tego obiektu będą wywoływać opisane w tym podrozdziale procedury, których instrukcje nie mają żadnych zależności od implementacji łańcucha.

Układ cząsteczek w aplikacji jest reprezentowany przez zmienną `ps` typu `PartSystem`, zadeklarowaną na listingu 24.6. W polach `vao` i `sbo` pamiętane są identyfikatory obiektu tablicy wierzchołków (cząsteczek) i buforów magazynowych. W polach `time0` i `last_time` są pamiętane chwile rozpoczęcia symulacji układu i ostatniego wywołania szadera obliczeniowego; pola te są typu `double` z powodu przedstawionego na s. 77. Pozostałe pola odpowiadają polom zmiennej jednolitej `ps` szadera z listingu 24.1.

Makrodefinicja `PARTICLE_COUNT` określa liczbę częsteczek, 2^{20}, która jest też wielkością globalnej grupy roboczej szadera obliczeniowego z listingu 24.1.

Procedura na listingu 24.7 przygotowuje układ cząsteczek do pracy; przed jej wywołaniem należy przygotować programy szaderów i odczytać położenia zmiennych jednolitych za pomocą procedury `LoadParticleShaders`.

[6] Normalizowanie zbyt długich wektorów zamiast ich odrzucania spowodowałoby wygenerowanie ciągu wektorów o niejednostajnym rozkładzie w kuli.

Listing 24.6. Reprezentacja układu cząsteczek w pamięci CPU

C

```
1: #define PARTICLE_COUNT 1048576
2:
3: typedef struct {
4:     GLuint  vao, sbo[3], ubo;
5:     double  time0, last_time;
6:     GLint   tablength, iter;
7:     GLfloat dt, Tpart, vel, TC, acc, divv, divp;
8:     GLfloat pos0[3], pos1[3], vel0[3], vel1[3], conv[3];
9:   } PartSystem;
```

Procedura `InitParticleSystem` kolejno: w linii 9 tworzy obiekt tablicy wierzchołków, w którym zarejestruje (w liniach 21–24) bufor z tablicą położeń cząsteczek; jest on jednocześnie buforem magazynowym `PartPos` szadera z listingu 24.1. Bufor ten jest tworzony jednocześnie z dwoma pozostałymi buforami magazynowymi w linii 10. W linii 11 jest rezerwowana tablica w pamięci RAM CPU, w której procedura przygotowuje pulę wektorów pseudolosowych; jest ich tyle, ile cząsteczek. Kolejne czwórki liczb w tej tablicy zawierają trójkę współrzędnych kartezjańskich i „startowy" ujemny wiek cząsteczki (obliczany w linii 16). Sposób określania wieku startowego ma na celu zapewnienie stałego tempa emitowania cząsteczek z wylotu dziobka czajnika. W liniach 18–19 dane z tablicy `mbuf` są przesyłane do bufora magazynowego `RandomVecPool`. Te same dane są przesyłane w liniach 22–23 do bufora `PartPos`, przy czym dla każdej cząsteczki jest istotny tylko jej wiek startowy, ponieważ położenie początkowe zostanie nadane przez szader obliczeniowy w chwili, gdy cząsteczka „dojrzeje", tj. osiągnie nieujemny wiek. Do bufora magazynowego `PartVel` nie trzeba wpisywać żadnych danych, bo to będzie robił szader obliczeniowy we właściwym czasie, zatem wywołanie procedury `glBufferData` w liniach 28–29 ma na celu tylko ustalenie wielkości bufora.

Uwaga: Choć elementy tablicy w tym buforze są typu `vec3` (zobacz listing 24.1), rezerwowane jest miejsce jak dla tablicy elementów typu `vec4`, ponieważ kompilator GLSL-a wyrównuje wielkość elementów tablicy do wielkości zmiennych typu `vec4`.

Listing 24.7. Procedura `InitParticleSystem`

C

```
1: #define UNB GL_UNIFORM_BUFFER
2: #define SSB GL_SHADER_STORAGE_BUFFER
3:
4: void InitParticleSystem ( PartSysPrograms *psprog, PartSystem *ps )
5: {
6:   GLfloat *mbuf;
7:   int     i;
8:
9:   glGenVertexArrays ( 1, &ps->vao );
10:  glGenBuffers ( 3, ps->sbo );
```

```
11:   if ( !(mbuf = (GLfloat*)malloc ( PARTICLE_COUNT*4*sizeof(GLfloat) )) )
12:     ExitOnError ( "InitParticles" );
13:   glBindBuffer ( SSB, ps->sbo[0] );
14:   for ( i = 0; i < PARTICLE_COUNT; i++ ) {
15:     RandomVector ( &mbuf[4*i] );
16:     mbuf[4*i+3] = (float)i/(float)PARTICLE_COUNT - 1.0;
17:   }
18:   glBufferData ( SSB, PARTICLE_COUNT*4*sizeof(GLfloat),
19:                  mbuf, GL_STATIC_DRAW );
20:   glBindVertexArray ( ps->vao );
21:   glBindBuffer ( GL_ARRAY_BUFFER, ps->sbo[1] );
22:   glBufferData ( GL_ARRAY_BUFFER, PARTICLE_COUNT*4*sizeof(GLfloat),
23:                  mbuf, GL_DYNAMIC_COPY );
24:   glVertexAttribPointer ( 0, 4, GL_FLOAT, GL_FALSE, 0, NULL );
25:   glEnableVertexAttribArray ( 0 );
26:   glBindVertexArray ( 0 );
27:   glBindBuffer ( SSB, ps->sbo[2] );
28:   glBufferData ( SSB, PARTICLE_COUNT*4*sizeof(GLfloat),
29:                  NULL, GL_DYNAMIC_COPY );
30:   free ( mbuf );
31:   glGenBuffers ( 1, &ps->ubo );
32:   glBindBufferBase ( UNB, psprog->psbp, ps->ubo );
33:   glBufferData ( UNB, psprog->psbsize, NULL, GL_DYNAMIC_DRAW );
34:   ps->tablength = PARTICLE_COUNT;
35:   glBufferSubData ( UNB, psprog->psofs[0], sizeof(GLint), &ps->tablength );
36:   ps->iter = 0;
37:   glBufferSubData ( UNB, psprog->psofs[1], sizeof(GLint), &ps->iter );
38:   ps->dt = 0.0;
39:   glBufferSubData ( UNB, psprog->psofs[2], sizeof(GLfloat), &ps->dt );
40:   ps->Tpart = 1.5;
41:   glBufferSubData ( UNB, psprog->psofs[3], sizeof(GLfloat), &ps->Tpart );
42:   ps->TC = 0.4;
43:   glBufferSubData ( UNB, psprog->psofs[5], sizeof(GLfloat), &ps->TC );
44:   ps->acc = 0.6;
45:   glBufferSubData ( UNB, psprog->psofs[6], sizeof(GLfloat), &ps->acc );
46:   ps->divv = 0.2;
47:   glBufferSubData ( UNB, psprog->psofs[7], sizeof(GLfloat), &ps->divv );
48:   ps->divp = 0.04;
49:   glBufferSubData ( UNB, psprog->psofs[8], sizeof(GLfloat), &ps->divp );
50:   ps->conv[0] = ps->conv[1] = 0.0;  ps->conv[2] = 0.5;
51:   glBufferSubData ( UNB, psprog->psofs[13], 3*sizeof(GLfloat), &ps->conv );
52:   ExitIfGLError ( "InitPartSystem" );
53: } /*InitPartSystem*/
```

W liniach 31–51 procedura tworzy bufor dla bloku zmiennych jednolitych `PartSys` i nadaje wartości zmiennym w tym polu: liczbę $N = 2^{20}$, numer początkowej iteracji, pierwszy krok czasowy, czas aktywności cząsteczki $T = 1.5$ s, stałą czasową $C = 0.4$ s, parametr wiel-

kości losowych przyspieszeń $a = 0.6$, parametry rozrzutu początkowych prędkości i położeń cząsteczek $v_0 = 0.2$ i $d_0 = 0.04$ oraz współrzędne wektora prędkości konwekcji $\boldsymbol{v}_{\text{conv}} = (0.0, 0.0, 0.5)$.

Listing 24.8 przedstawia procedurę wywołującą szader obliczeniowy w celu wykonania jednego kroku całkowania ruchu cząsteczek. Parametr `time` podaje czas (w sekundach), który upłynął od początku działania aplikacji. W zmiennej `ps.last_time` jest zapamiętany czas do poprzedniego wywołania. Jeśli krok czasowy obliczony w linii 5 jest krótszy niż $\frac{1}{100}$ s, to następuje powrót. Gdyby zaś krok był dłuższy niż 1 s, to jest on ograniczany, aby zabezpieczyć układ cząsteczek przed „eksplozją".

Listing 24.8. Procedura `MoveParticles`

C

```
1: void MoveParticles ( PartSysPrograms *psprog, PartSystem *ps, double time )
2: {
3:   GLfloat deltat;
4:
5:   if ( (deltat = time-ps->last_time) < 0.01 )
6:     return;
7:   ps->last_time = time;
8:   if ( deltat > 1.0 )
9:     deltat = 1.0;
10:  glUseProgram ( psprog->program_id[0] );
11:  glBindBufferBase ( SSB, psprog->rvpbp, ps->sbo[0] );
12:  glBindBufferBase ( SSB, psprog->posbp, ps->sbo[1] );
13:  glBindBufferBase ( SSB, psprog->velbp, ps->sbo[2] );
14:  time -= ps->time0;
15:  glBindBufferBase ( UNB, psprog->psbp, ps->ubo );
16:  ps->iter = (ps->iter+1) % PARTICLE_COUNT;
17:  glBufferSubData ( UNB, psprog->psofs[1], sizeof(GLint), &ps->iter );
18:  ps->dt = deltat;
19:  glBufferSubData ( UNB, psprog->psofs[2], sizeof(GLfloat), &ps->dt );
20:  ps->vel = 1.0+0.1*sin ( 18.0*PI*time*(1.0+sin(time)) );
21:  glBufferSubData ( UNB, psprog->psofs[4], sizeof(GLfloat), &ps->vel );
22:  glBufferSubData ( UNB, psprog->psofs[9], 3*sizeof(GLfloat), ps->pos0 );
23:  glBufferSubData ( UNB, psprog->psofs[10], 3*sizeof(GLfloat), ps->pos1 );
24:  glBufferSubData ( UNB, psprog->psofs[11], 3*sizeof(GLfloat), ps->vel0 );
25:  glBufferSubData ( UNB, psprog->psofs[12], 3*sizeof(GLfloat), ps->vel1 );
26:  COMPUTE ( PARTICLE_COUNT, 1, 1 )
27:  ExitIfGLError ( "MoveParticles" );
28: } /*MoveParticles*/
```

W liniach 10–13 procedura wybiera program z szaderem obliczeniowym i przywiązuje bufory magazynowe do odpowiednich punktów dowiązania. W zmiennej `ps.time0` jest pamiętany czas od początku symulacji układu, tj. od ostatniego włączenia symulacji lub ostatniego wywołania procedury `ResetParticles`, zatem w linii 14 jest obliczany czas w sekundach od początku symulacji do chwili obecnej. Czas ten jest używany do modulowania

średniej prędkości początkowej cząsteczek przy użyciu (wyssanego, nie da się ukryć, z palca) wzoru zapisanego w linii 20. Zmiennym jednolitym `ps.pos0`, `ps.pos1`, `ps.vel0` i `ps.vel1` procedura nadaje wartości obliczone przez procedurę artykulacji łańcucha kinematycznego — to są położenia i wektory kierunków wylotu dziobka czajnika w chwili t_{i-1} (poprzedniej) i t_i (obecnej). Makrodefinicja `COMPUTE` wywołuje procedurę `glDispatchCompute`, która powoduje przystąpienie szadera obliczeniowego do pracy i procedurę `glMemoryBarrier`, która czeka na dokończenie tej pracy, aby można było rysować cząsteczki w nowych położeniach. Powrót z tej procedury następuje, gdy wyniki są gotowe.

Listing 24.9. Procedura `DrawParticles`

C

```
1: void DrawParticles ( PartSysPrograms *psprog, PartSystem *ps,
2:                      Camera *camera, char final )
3: {
4:   GLfloat c[4];
5:
6:   glDepthMask ( GL_FALSE );
7:   glUseProgram ( psprog->program_id[1] );
8:   if ( !final ) {
9:     c[0] = c[1] = c[2] = 0.7;
10:    c[3] = (float)(SHADOW_MAP_SIZE*SHADOW_MAP_SIZE);
11:  }
12:  else {
13:    c[0] = c[1] = c[2] = 0.85;
14:    c[3] = (float)(camera->win_height*camera->win_height);
15:  }
16:  c[3] = exp ( -0.001*c[3]/(float)PARTICLE_COUNT );
17:  glUniform4fv ( psprog->pcloc, 1, c );
18:  glBindVertexArray ( ps->vao );
19:  glEnable ( GL_DEPTH_CLAMP );
20:  glEnable ( GL_BLEND );
21:  glBlendFunc ( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
22:  glPointSize ( 1.0 );
23:  glDrawArrays ( GL_POINTS, 0, PARTICLE_COUNT );
24:  glDisable ( GL_BLEND );
25:  glDisable ( GL_DEPTH_CLAMP );
26:  glDepthMask ( GL_TRUE );
27:  glBindVertexArray ( 0 );
28:  ExitIfGLError ( "DrawParticles" );
29: } /*DrawParticles*/
```

Procedura `DrawParticles` (listing 24.9) rysuje cząsteczki. Używamy jej zarówno do narysowania cząsteczek (czyli mgły) na końcowym obrazie, jak i do wykonania pomocniczego obrazu, który służy do tego, aby na końcowym obrazie otrzymać cień rzucany przez mgłę. Algorytm cieni dla mgły jest opisany dalej, ale w tym miejscu trzeba wyjaśnić cele, dla których są wywoływane poszczególne procedury OpenGL-a. W linii 18 obiekt tablicy wierzchołków (VAO) z tablicą położeń cząsteczek jest czyniony bieżącym, aby procedura `glDrawArrays`

brała wierzchołki do narysowania z tej tablicy. Instrukcja w linii 19 powoduje zmianę sposobu obcinania. Jeśli położenie cząsteczki w układzie kostki standardowej ma współrzędną z poza przedziałem $[-1,1]$, to współrzędna ta zostanie zastąpiona przez bliższy koniec tego przedziału, dzięki czemu cząsteczki, które uciekną z bryły widzenia też zostaną narysowane[7]. Instrukcja w linii 25 przywraca domyślny sposób obcinania.

Cząsteczki są częściowo przezroczyste. To oznacza, że choć muszą być poddawane testowi widoczności (nie należy rysować cząsteczek zasłoniętych przez czajnik lub lustro), narysowanie widocznej cząsteczki nie może powodować zmiany zawartości bufora głębokości, ponieważ przetwarzana później inna cząsteczka, znajdująca się dalej od obserwatora, ale przed nieprzezroczystymi obiektami, też powinna przejść test widoczności i być narysowana. Dlatego w linii 6 jest wywoływana procedura `glDepthMask` z parametrem `GL_FALSE`, która blokuje przypisywanie nowej zawartości do bufora głębokości. Zauważmy, że nie jest dobrym pomysłem wyłączenie testu widoczności (instrukcją `glDisable ( GL_DEPTH_TEST );`) w ogóle — nie chcemy przecież oglądać na obrazie niewidocznych cząsteczek. Po narysowaniu cząsteczek, w linii 26, następuje odblokowanie zapisu do bufora głębokości, tj. przywrócenie domyślnego stanu początkowego.

W linii 20 włączane jest mieszanie kolorów. Kolor nadany pikselowi po przetworzeniu fragmentu ma być mieszaniną koloru dotychczasowego i koloru przypisanego przez szader fragmentów zmiennej wyjściowej `colour` (zobacz listing 24.3). **Funkcja mieszająca** kolor fragmentu, $\boldsymbol{c}_s$, z poprzednim kolorem piksela, $\boldsymbol{c}_d$, wybrana przez parametry procedury `glBlendFunc` w linii 21, jest dana wzorem

$$\boldsymbol{c} = (1-\alpha)\boldsymbol{c}_d + \alpha\boldsymbol{c}_s,$$

w którym α oznacza składową alfa zmiennej wyjściowej `colour`. Rzut oka na kod szadera fragmentów ujawnia wzór

$$\alpha(w) = a(1-w)^3 w$$

używany do obliczania tej składowej na podstawie wieku w cząsteczki i składowej alfa (współrzędnej a) zmiennej jednolitej `PartColour`. Przypomnę, że wiek aktywnej cząsteczki jest liczbą z przedziału $[0,1]$, a więc cząsteczki „bardzo młode" i „bardzo stare" są na podstawie tego wzoru prawie przezroczyste.

W linii 17 zmiennej jednolitej `PartColour` jest przypisywany kolor (wektor $\boldsymbol{c}_s$) wraz ze składową alfa (współczynnik a), która określa nieprzezroczystość cząsteczek. Kolor mgły, która na końcowym obrazie ma być widoczna na białym tle, jest jasny szary ($r = g = b = 0.85$), natomiast na obrazie pomocniczym dla algorymu cienia jest ciemniejszy, aby uwydatnić cień na końcowym obrazie. Objaśnienia wymaga sposób obliczania współczynnika a, od którego zależy składowa α koloru wyprowadzanego przez szader fragmentów. Celem jest spowodowanie, aby mgła na obrazie pozostała „tak samo gęsta" po zmianie wymiarów okna, a także po zmianie liczby cząsteczek w układzie. Przypuśćmy, że czynnik $\beta = 1 - \alpha$, który

[7] Określając bryłę widzenia dla algorytmu cieni, braliśmy pod uwagę tylko „stałe" elementy sceny, tj. czajnik, torus i lustro. Nie zmuszamy cząsteczek do pozostania w tej bryle.

określa, jaka część światła przechodzi przez cząsteczkę, jest dla wszystkich cząsteczek taki sam*. Wtedy kolor piksela na końcowym obrazie jest dany wzorem

$$\boldsymbol{c}_f = (1-\beta^n)\boldsymbol{c}_s + \beta^n \boldsymbol{c}_0,$$

w którym $\boldsymbol{c}_0$ oznacza kolor piksela na obrazie bez mgły, a n jest liczbą cząsteczek, które na obrazie „wpadły w ten piksel". Jest $n \approx CN/h^2$, gdzie C jest pewną stałą, N jest liczbą wszystkich cząsteczek, h oznacza wysokość obrazu w pikselach[8]. Aby zachować proporcję, w jakiej kolory $\boldsymbol{c}_0$ i $\boldsymbol{c}_s$ są zmieszane po zmianie wielkości okna lub po zmianie liczby N, należałoby przyjąć $\beta = \mathrm{e}^{Dh^2/N}$, dla pewnej stałej $D < 0$. *Zamiast tego* współczynnik a jest dobierany tak, aby w wyrażeniu $\alpha^n \boldsymbol{c}_s + (1-\alpha^n)\boldsymbol{c}_0$ liczba $\alpha^n = 1-\beta^n \approx \alpha^{CN/h^2}$ nie zależała od N ani od h*. To ma na celu zakodowany w liniach 10, 14 i 16 wzór

$$a = \mathrm{e}^{dh^2/N}$$

ze stałą $d < 0$, którą trzeba dobrać doświadczalnie. W programie przyjąłem $d = -0.001$. Zwracam uwagę, że to rozwiązanie nie zapewnia *dokładnego* zachowania gęstości mgły na całym obrazie po zmianie wysokości h lub liczby N, ponieważ w dwóch miejscach rozumowania oznaczonych gwiazdkami jest (rozmyślny, bo nieunikniony) błąd[9].

Pokazana na listingu 24.10 procedura `ResetParticles` jest wywoływana, gdy użytkownik włączy lub zrestartuje symulację układu (naciskając odpowiedni klawisz) lub gdy zmieni wymiary okna. Parametr `time` podaje czas od początku działania aplikacji. Czas ten jest zapamiętywany w zmiennych `ps.time0` i `ps.last_time`. W liniach 4 i 5 procedura `glBindBuffer` przywiązuje bufory magazynowe z pulą wektorów pseudolosowych i z położeniami cząsteczek do celów `GL_COPY_READ_BUFFER` i `GL_COPY_WRITE_BUFFER`. Cele te zostały wprowadzone specjalnie po to, aby operacja kopiowania danych z jednego bufora w pamięci GPU do drugiego nie musiała angażować innych celów w kontekście OpenGL-a, używanych w danej chwili do innych celów[10]. Kopiowania danych dokonuje procedura `glCopyBufferSubData`, przy czym znów, istotny w tym momencie jest tylko początkowy wiek każdej cząsteczki; jest on liczbą ujemną przechowywaną w polu `w` odpowiedniego elementu tablicy `rvpool.rv`. Ponadto procedura kasuje licznik iteracji (kroków całkowania) układu cząsteczek.

Sprzątanie układu cząsteczek polega na usunięciu utworzonych przez procedurę `InitParticleSystem` buforów i obiektu tablicy wierzchołków. Robi to procedura z listin-

[8] Zajrzyj do opisu konstrukcji bryły widzenia w rozdziale 7 i w szczególności zastanów się, jak dostosować algorytm obliczania współczynnika a do konstrukcji bryły widzenia zmodyfikowanej przez rozwiązanie ćwiczenia 7.8.

[9] Udało się osiągnąć tyle, że zmiana h lub N powoduje taką zmianę funkcji $\alpha(w)$, aby proporcje, w jakich kolory $\boldsymbol{c}_0$ i $\boldsymbol{c}_s$ są zmieszane na końcowym obrazie, pozostawały niezmienione tylko dla pewnej wartości zmiennej w. To musi wystarczyć.

[10] Aby skopiować dane między buforami, można te bufory przywiązać do dowolnych celów, które trzeba następnie podać jako parametry procedury `glCopyBufferSubData`. W OpenGL 4.5 i nowszych można też użyć procedury `glCopyNamedBufferSubData`, której pierwsze dwa parametry to identyfikatory buforów; wtedy nie trzeba buforów przywiązywać.

gu 24.11, wywoływana przez destruktor obiektu mgły (listing 24.13). Natomiast do procedury `DeleteMyWorld` trzeba dodać wywołanie procedury kasującej programy szaderów.

Listing 24.10. Procedura `ResetParticles`

```C
void ResetParticles ( PartSysPrograms *psprog, PartSystem *ps, double time )
{
  ps->time0 = ps->last_time = time;
  glBindBuffer ( GL_COPY_READ_BUFFER, ps->sbo[0] );
  glBindBuffer ( GL_COPY_WRITE_BUFFER, ps->sbo[1] );
  glCopyBufferSubData ( GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER,
                        0, 0, PARTICLE_COUNT*4*sizeof(GLfloat) );
  glBindBufferBase ( UNB, psprog->psbp, ps->ubo );
  ps->iter = 0;
  glBufferSubData ( UNB, psprog->psofs[1], sizeof(GLint), &ps->iter );
  ExitIfGLError ( "ResetParticles" );
} /*ResetParticles*/
```

Listing 24.11. Procedura sprzątania układu cząsteczek

```C
void DeleteParticleSystem ( PartSystem *ps )
{
  glDeleteVertexArrays ( 1, &ps->vao );
  glDeleteBuffers ( 3, ps->sbo );
  glDeleteBuffers ( 1, &ps->ubo );
} /*DeleteParticleSystem*/
```

24.5. Zmiany łańcucha kinematycznego

Do łańcucha kinematycznego pokazanego na rysunku 23.1 trzeba dodać tylko jeden nowy obiekt, związany z członem L_4, z którym wcześniej zostały związane punkty kontrolne płatów Béziera składających się na dziobek czajnika. Zmiany procedury tworzenia łańcucha są pokazane na listingu 24.12 (porównaj go z listingiem 23.4). Parametry procedury `kl_NewLinkage` określające liczby obiektów i ich referencji zostały zwiększone do 3 i 5. W liniach 22–24 doszło polecenie utworzenia nowego obiektu, który będzie określał miejsca emisji i początkowe prędkości cząsteczek. Metody tego obiektu są pokazane na listingu 24.13. Macierz E_2, która opisuje wstępne przekształcenie obiektu (zobacz podrozdz. 13.1), jest taka sama jak macierz E_0 związana z czajnikiem. Dla algorytmu widoczności istotna jest kolejność tworzenia obiektów (która będzie kolejnością ich rysowania) — obiekt mgły musi być ostatni.

Listing 24.13 przedstawia procedury, z których pierwsza jest konstruktorem obiektu mgły, a pozostałe to metody wirtualne `transform`, `postprocess`, `redraw` i `destroy` tego obiektu. Konstruktor wywołuje procedurę inicjalizacji układu cząsteczek i wprowadza referencję tego obiektu wiążącą go z członem L_4, tym samym, z którym związany jest dziobek czajnika. Liczba wierzchołków obiektu mgły jest równa 0.

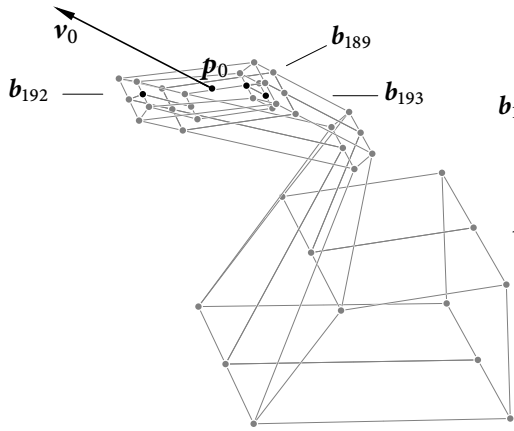
Listing 24.12. Zmiany procedury `ConstructMyLinkage`

C

```
kl_linkage *ConstructMyLinkage ( AppData *ad )
{
#define SCF (1.0/3.0)
  kl_linkage *lkg;
  int        l[10], j[9];
  int        i;
  GLfloat    tra[16], trb[16];

  if ( (lkg = kl_NewLinkage ( 3, 10, 5, 9, 9, (void*)ad )) ) {
    ad->linkage = lkg;
    for ( i = 0; i < 10; i++ )
      l[i] = kl_NewLink ( lkg );
    M4x4Scalef ( tra, SCF, SCF, SCF*4.0/3.0 );
    kl_NewObject ( lkg, 3,  TEAPOT_NPOINTS, tra, (void*)&ad->bezp[0],
                   ConstructMyTeapot, KLTransformBP, KLPostprocessBP,
                   KLRedrawBezPatches, KLDeleteBezPatches );
    M4x4RotateXf ( trb, 0.5*PI );
    M4x4MScalef ( trb, 0.1, 0.1, 0.1 );
    kl_NewObject ( lkg, 0, 4, 49, trb, (void*)&ad->bezp[1],
                    ConstructMyTorus, KLTransformBP, KLPostprocessBP,
                    KLRedrawBezPatches, KLDeleteBezPatches );
    kl_NewObject ( lkg, 0, 3, 0, tra, (void*)&ad->ps, ConstructParticleGun,
                    KLTransformParticles, KLPostprocessParticles,
                    KLRedrawParticles, KLDeleteParticles );
    .... /* dalej instrukcje bez zmian */
    glGenBuffers ( 1, &ad->lktrbuf );
    glBindBuffer ( GL_SHADER_STORAGE_BUFFER, ad->lktrbuf );
    glBufferData ( GL_SHADER_STORAGE_BUFFER, lkg->norefs*16*sizeof(GLfloat),
                    NULL, GL_DYNAMIC_DRAW );
  }
  else
    ExitOnError ( "ConstructMyLinkage" );
  return lkg;
#undef SCF
} /*ConstructMyLinkage*/
```

Procedura `KLTransformParticles`, wywoływana w celu dokonania artykulacji, poddaje przekształceniu opisanemu przez otrzymaną macierz punkt $\boldsymbol{p}_0$ i wektor $\boldsymbol{v}_0$, które opisują położenie i kierunek „działa" emitującego cząsteczki (rys. 24.1). Dane te są przypisywane polom `pos1` i `vel1` struktury łańcucha, po zapamiętaniu w polach `pos0` i `vel0` poprzedniego położenia obiektu. Dane te, później przypisane zmiennym o tych samych nazwach w bloku `PartSys`, służą do interpolacji mającej na celu obliczenie położenia i prędkości początkowej cząsteczki emitowanej z dziobka.

Procedura `KLPostprocessParticles` dokonuje całkowania ruchu cząsteczek przy użyciu szadera obliczeniowego. W zasadzie mogłoby jej nie być, gdyby jej instrukcje były prze-



$$b_{189} = \begin{bmatrix} 2.9 \\ 0 \\ 2.475 \end{bmatrix} \quad b_{192} = \begin{bmatrix} 3.45 \\ 0 \\ 2.5125 \end{bmatrix} \quad b_{193} = \begin{bmatrix} 2.8 \\ 0 \\ 2.4 \end{bmatrix}$$

$$p_0 = \tfrac{2}{3} b_{189} + \tfrac{1}{3} b_{192}$$
$$v_0 = 8(b_{189} - b_{193})$$

Rysunek 24.1. Konstrukcja położenia i prędkości początkowej cząsteczek

niesione na koniec procedury `KLTransformParticles`. Zmienna typu `AppData`, opisująca scenę do narysowania, ma dodatkowe pola, `ps` typu `PartSystem` i `psprog` typu `PartSysPrograms`. Adresy tych pól, opakowujących układ cząsteczek i programy jego przetwarzania, są przekazywane procedurze `MoveParticles` z listingu 24.8. Pole `particles` jest przełącznikiem — symulacja i rysowanie cząsteczek odbywa się, gdy ma ono wartość `true`.

Dodatkowe pole `hold_time` struktury `AppData` umożliwia wstrzymywanie na polecenie użytkownika *całej* animacji. Wartość tego pola jest to całkowity czas wstrzymania animacji. Opis realizacji tego elementu interakcji z użytkownikiem jest dalej.

Procedura `KLRedrawParticles` jest wywoływana w celu narysowania mgły. Jeśli pole `final` ma wartość `true`, to jest to rysowanie końcowego obrazu lub obrazu sceny odbitej w lustrze. Jeśli pole to ma wartość `false`, to rysowanie cząsteczek widzianych z punktu położenia źródła światła ma na celu znalezienie cienia rzucanego przez mgłę na pozostałe obiekty sceny. Rola procedury `glColorMask`, której wywołanie poprzedza wywołanie procedury rysującej, jest objaśniona w następnym podrozdziale.

Ostatnią metodą obiektu mgły jest destruktor, tj. procedura `KLDeleteParticles`. Jej działanie jest oczywiste.

Listing 24.13. Metody obiektu mgły

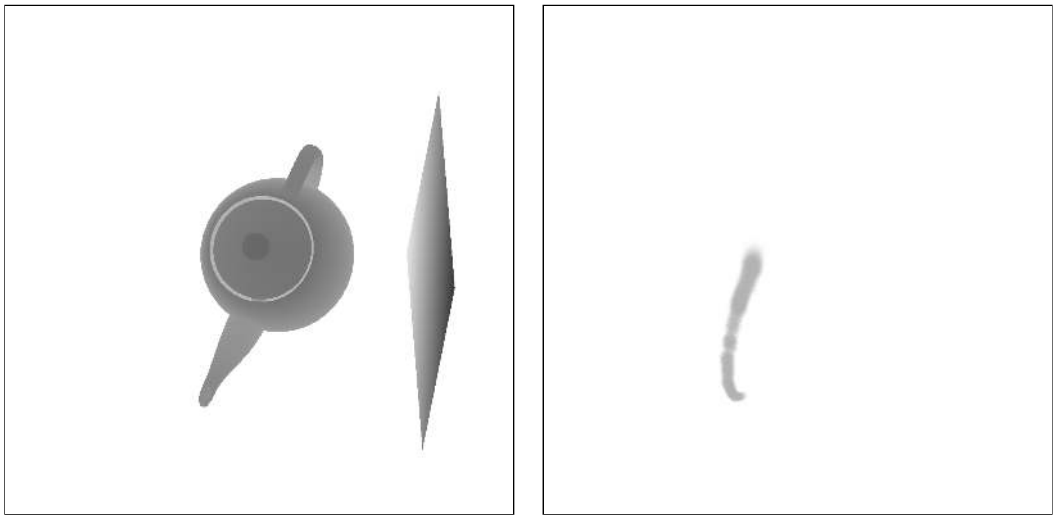
C

```
1: static char ConstructParticleGun ( kl_linkage *lkg, kl_object *obj )
2: {
3:   AppData    *ad;
4:   PartSystem *ps;
5:
6:   ad = (AppData*)lkg->usrdata;
7:   ps = (PartSystem*)obj->usrdata;
8:   InitParticleSystem ( &ad->psprog, ps );
9:   kl_NewObjRef ( lkg, 4, obj - lkg->obj, 0, NULL );
```

```
10:   return true;
11: } /*ConstructParticleGun*/
12:
13: static void KLTransformParticles ( kl_linkage *lkg, kl_object *obj,
14:                                     int refn, GLfloat *tr, int nv, int *vn )
15: {
16:   const GLfloat p0[3] = {3.083333,0.0,2.4875},
17:                 v0[3] = {0.8,0.0,0.6};
18:   PartSystem *ps;
19:
20:   ps = (PartSystem*)obj->usrdata;
21:   memcpy ( ps->pos0, ps->pos1, 3*sizeof(GLfloat) );
22:   M4x4MultMP3f ( ps->pos1, tr, p0 );
23:   memcpy ( ps->vel0, ps->vel1, 3*sizeof(GLfloat) );
24:   M4x4MultMV3f ( ps->vel1, tr, v0 );
25:   V3Normalisef ( ps->vel1 );
26: } /*KLTransformParticles*/
27:
28: static void KLPostprocessParticles ( kl_linkage *lkg, kl_object *obj )
29: {
30:   AppData *ad;
31:
32:   ad = (AppData*)lkg->usrdata;
33:   if ( ad->particles )
34:     MoveParticles ( &ad->psprog, &ad->ps, app_time-ad->hold_time );
35: } /*KLPostprocessParticles*/
36:
37: static void KLRedrawParticles ( kl_linkage *lkg, kl_object *obj )
38: {
39:   AppData *ad;
40:
41:   ad = (AppData*)lkg->usrdata;
42:   if ( ad->particles ) {
43:     if ( !ad->final )
44:       glColorMask ( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE );
45:     DrawParticles ( &ad->psprog, &ad->ps, &ad->camera, ad->final );
46:   }
47: } /*KLRedrawParticles*/
48:
49: static void KLDeleteParticles ( kl_linkage *lkg, kl_object *obj )
50: {
51:   AppData *ad;
52:
53:   ad = (AppData*)lkg->usrdata;
54:   DeleteParticleSystem ( &ad->ps );
55: } /*KLDeleteParticles*/
```

24.6. Algorytm cieni dla mgły

Kolejnym problemem do rozwiązania jest uzyskanie na obrazie czajnika cienia spowodowanego obecnością mgły między czajnikiem a źródłem światła. Ten efekt można otrzymać przy użyciu dodatkowej tekstury uzupełniającej teksturę obszaru cienia wprowadzoną w aplikacji drugiej G (rozdz. 22). Konstruując reprezentację obszaru cienia w postaci tekstury, rysowaliśmy scenę widzianą z kierunku dochodzenia światła (lub z punku położenia źródła światła), ale nie wykonywaliśmy obrazu, bo był niepotrzebny. Teraz utworzymy obraz, rysując na nim te cząsteczki, które są dla źródła światła widoczne. Obraz będzie wykonany na białym tle. Najpierw narysujemy czajnik, torus i lustro, dzięki czemu uzyskamy odpowiednią teksturę obszaru cienia (rys. 24.2 lewy), ale zablokujemy przypisywanie wartości pikselom na obrazie, aby tło pozostało białe. Potem narysujemy mgłę, ale tym razem włączymy przypisywanie wartości pikselom, a za to zablokujemy pisanie do bufora głębokości. Cząsteczki mają na tym obrazie kolor szary. Rysując końcowy obraz, użyjemy tekstury stworzonej z tego obrazu (**tekstury mgły**, rys. 24.2 prawy) do określenia stopnia stłumienia padającego na powierzchnię światła, które przeszło przez mgłę.



Rysunek 24.2. Tekstura obszaru cienia i tekstura mgły

Listing 24.14 przedstawia zmieniony typ struktury opisującej źródło światła. Tu zmiana polega na przerobieniu pola `shadow_txt` na tablicę, bo teraz potrzebujemy dwóch tekstur.

Zmienione i nowe instrukcje w procedurach związanych z buforem ramki do znajdowania obszaru cienia są pokazane na listingu 24.15. Zmodyfikowany szader fragmentów (listing 24.17) ma dwie tablice ewaluatorów tekstur używanych do otrzymania cieni. Do tablicy `shtex`, której elementy są typu `sampler2DShadow`, doszła nowa tablica `parttex` elementów „zwykłego" typu `sampler2D`. Długość obu tablic jest równa maksymalnej liczbie źródeł światła (czyli 8, bo taka jest treść makra `MAX_NLIGHTS`). Tekstury z l-tej pozycji w tablicach

Listing 24.14. Zmieniona struktura `LSPar`

C

```
1: typedef struct LSPar {
2:     GLfloat position[4];
3:     GLfloat ambient[3];
4:     GLfloat direct[3];
5:     GLfloat attenuation[3];
6:     GLuint  shadow_fbo, shadow_txt[2];
7:     GLfloat shadow_view[16], shadow_proj[16];
8:   } LSPar;
```

tworzą parę dla jednego źródła światła, a ich punkty dowiązania mają numery różniące się o `MAX_NLIGHTS` (porównaj linie 13 i 17 na listingu 24.15 oraz 6 i 7 na listingu 24.17).

Wystarczy, aby tekstura mgły miała teksele o jednej składowej (formalnie czerwonej, bo współrzędna r jest pierwsza w układzie r, g, b, a). Tu jest potrzebna dosyć dokładna reprezentacja tekseli, dlatego użyłem 32-bitowych liczb zmiennopozycyjnych[11]. Wymiary tekstury podane w liniach 18–19 są takie same jak wymiary chwilę wcześniej utworzonej tekstury obszaru cienia. Siódmy i ósmy parametr, `GL_RED` i `GL_FLOAT`, opisują reprezentację danych w pamięci CPU, które procedura `glTexImage2D` miałaby przesłać, gdyby istniały. Ale ostatni parametr jest wskaźnikiem pustym, zatem wspomniane parametry są tu nieistotne. Wywołanie procedury `glTexImage2D` ma na celu zarezerwowanie pamięci GPU na teksturę i określenie postaci tekseli przez pierwsze pięć parametrów.

W liniach 20–23 są określane parametry dla ewaluatora tekstury, wybierające sposób filtrowania podczas wykonywania końcowego obrazu (będzie stosowana interpolacja liniowa, mamy tu tylko jeden poziom tekstury, zatem nie będzie mipmapingu) i sposób postępowania, gdy parametr ewaluatora opisuje punkt poza dziedziną tekstury. Instrukcja w liniach 27–28 czyni z rozważanej tu tekstury załącznik obrazu (`GL_COLOR_ATTACHMENT0`) bufora ramki używanego przez algorytm cieni.

Listing 24.15. Zmiany w procedurze `ConstructShadowTxtFBO`

C

```
1: void ConstructShadowTxtFBO ( LightBl *light, int l )
2: {
3:   GLuint fbo, txt[2];
4:
5:   if ( l < 0 || l >= MAX_NLIGHTS )
6:     return;
7:   glGenTextures ( 2, txt );
8:   glGenFramebuffers ( 1, &fbo );
9:   if ( (light->ls[l].shadow_txt[0] = txt[0]) &&
10:       (light->ls[l].shadow_txt[1] = txt[1]) &&
11:       (light->ls[l].shadow_fbo = fbo) ) {
12:     glActiveTexture ( GL_TEXTURE2+l );
```

[11] Można rozważyć użycie typu całkowitego 16- lub 32-bitowego. W tym celu należałoby podać jako trzeci parametr (`internalFormat`) procedury `glTexImage2D` stałą `GL_R16` albo `GL_R32UI`. Dokładność zapewniona przez 8 bitów w tym przypadku jest niewystarczająca.

```
13:     glBindTexture ( GL_TEXTURE_2D, txt[0] );
14:     .... /* ustawianie parametrów tekstury obszaru cienia bez zmian */
15:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
16:     glActiveTexture ( GL_TEXTURE2+MAX_NLIGHTS+l );
17:     glBindTexture ( GL_TEXTURE_2D, txt[1] );
18:     glTexImage2D ( GL_TEXTURE_2D, 0, GL_R32F, SHADOW_MAP_SIZE,
19:                    SHADOW_MAP_SIZE, 0, GL_RED, GL_FLOAT, NULL );
20:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
21:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
22:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
23:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
24:     glBindTexture ( GL_TEXTURE_2D, 0 );
25:     glBindFramebuffer ( GL_FRAMEBUFFER, fbo );
26:     glFramebufferTexture ( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, txt[0], 0 );
27:     glFramebufferTexture ( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
28:                            txt[1], 0 );
29:     if ( glCheckFramebufferStatus ( GL_DRAW_FRAMEBUFFER ) !=
30:          GL_FRAMEBUFFER_COMPLETE )
31:       ExitOnError ( "Framebuffer incomplete" );
32:     glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
33:     .... /* końcowe instrukcje bez zmian */
34:     ExitIfGLError ( "ConstructShadowTxtFBO" );
35:   }
36: } /*ConstructShadowTxtFBO*/
37:
38: void BindShadowTxtFBO ( TransBl *trans, LightBl *light, int l )
39: {
40:   if ( l < 0 || l >= MAX_NLIGHTS )
41:     return;
42:   if ( (light->ls[l].shadow_txt[0] == 0) )
43:     return;
44:     .... /* linie 60-63 z listingu 22.8 bez zmian */
45: } /*BindShadowTxtFBO*/
46:
47: void DeleteShadowFBO ( LightBl *light )
48: {
49:   int l;
50:
51:   glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
52:   for ( l = 0; l < MAX_NLIGHTS; l++ ) {
53:     if ( light->ls[l].shadow_fbo != 0 )
54:       glDeleteFramebuffers ( 1, &light->ls[l].shadow_fbo );
55:     if ( light->ls[l].shadow_txt[0] != 0 )
56:       glDeleteTextures ( 2, light->ls[l].shadow_txt );
57:   }
58:   ExitIfGLError ( "DeleteShadowFBO" );
59: } /*DeleteShadowFBO*/
```

Procedurę DeleteShadowFBO (listing 22.8), wywoływaną podczas sprzątania, trzeba zmienić tak, aby likwidowała obie tekstury utworzone dla każdego źródła światła.

Na listingu 24.16 są pokazane zmiany procedury DrawSceneToShadows, której zadaniem jest znalezienie obszarów cienia dla włączonych źródeł światła. Do tego doszło zadanie znalezienia tekstur reprezentujących cienie rzucane przez mgłę na pozostałe obiekty sceny. Struktura typu AppData została rozbudowana o pole particles typu char, która ma wartość true, gdy aplikacja ma symulować ruch cząsteczek i wykonywać ich obrazy. Przed rysowaniem trzeba skasować *oba* załączniki bufora ramki, tj. teksturę obszaru cienia i teksturę mgły. Wszystkie piksele tej ostatniej otrzymują kolor maksymalnie jasny (tu piksele mają tylko jedną składową, czerwoną, zatem tylko pierwszy parametr procedury wywołanej w linii 6 ma istotną wartość). Po narysowaniu obrazów reprezentujących cienie dla wszystkich źródeł światła trzeba podłączyć wszystkie tekstury obszarów cienia i mgły do odpowiednich elementów tablic zadeklarowanych w szaderze fragmentów, przy użyciu którego powstanie właściwy obraz (do nałożenia na lustro i końcowy obraz w oknie). Tekstura mgły jest podłączana w liniach 25–26.

Listing 24.16. Procedura znajdowania cieni z mgłą

C

```
1: void DrawSceneToShadows ( AppData *ad )
2: {
3:   int    l;
4:   GLuint mask;
5:
6:   glClearColor ( 1.0, 1.0, 1.0, 1.0 );
7:   glViewport ( 0, 0, SHADOW_MAP_SIZE, SHADOW_MAP_SIZE );
8:   glEnable ( GL_POLYGON_OFFSET_FILL );
9:   glPolygonOffset ( 2.0f, 4.0f );
10:  for ( l = 0, mask = 0x00000001;  l < ad->light.nls;  l++, mask <<= 1 )
11:    if ( ad->light.shmask & mask ) {
12:      BindShadowTxtFBO ( &ad->trans, &ad->light, l );
13:      glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
14:      glColorMask ( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );
15:      DrawMirror ( ad, false );
16:      DrawMyLinkage ( ad, false );
17:    }
18:  glColorMask ( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE );
19:  glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
20:  glDisable ( GL_POLYGON_OFFSET_FILL );
21:  for ( l = 0, mask = 0x00000001;  l < ad->light.nls;  l++, mask <<= 1 )
22:    if ( ad->light.shmask & mask ) {
23:      glActiveTexture ( GL_TEXTURE2+l );
24:      glBindTexture ( GL_TEXTURE_2D, ad->light.ls[l].shadow_txt[0] );
25:      glActiveTexture ( GL_TEXTURE2+MAX_NLIGHTS+l );
26:      glBindTexture ( GL_TEXTURE_2D, ad->light.ls[l].shadow_txt[1] );
27:    }
28: } /*DrawSceneToShadows*/
```

Przed rysowaniem jest wywołana procedura `glColorMask`, aby wyłączyć przypisywanie wartości pikselom powstającego obrazu. Cztery parametry tej procedury służą do indywidualnego sterowania nadawaniem wartości składowych *r*, *g*, *b*, *a* pikseli. Pamiętamy, że to rysowanie ma na celu otrzymanie tekstury obszaru cienia, cały zaś obraz (tekstura mgły) ma przedstawiać tylko widoczne cząsteczki mgły. Dlatego rysowanie obiektów — czajnika, torusa i lustra — pozostawia reprezentację obszaru cienia w buforze głębokości, ale zostawia nietknięte tło. Mgła, jako ostatni obiekt w łańcuchu, jest rysowana na końcu, dzięki czemu cząsteczki zasłonięte przez pozostałe obiekty są odrzucane przez test widoczności. Przed wywołaniem `DrawParticles` procedura `KLRedrawParticles` włącza przypisywanie koloru pikselom, wywołując procedurę `glColorMask` z czterema parametrami `GL_TRUE` (listing 24.13, linia 44). Jeśli mgła nie jest rysowana, to stan domyślny, w którym kolory pikselom są przypisywane, przywraca procedura wywołana w linii 18.

Listing 24.17. Zmiany szadera fragmentów w programie rysowania powierzchni

GLSL

```
1: #version 450 core
2:
3: #define MAX_NLIGHTS 8
4: .... /* początkowe deklaracje bez zmian */
5: layout(binding=0) uniform sampler2D tex;
6: layout(binding=2) uniform sampler2DShadow shtex[MAX_NLIGHTS];
7: layout(binding=2+MAX_NLIGHTS) uniform sampler2D parttex[MAX_NLIGHTS];
8: uniform int ColourSource = 0, LightingModel = 0, NormalSource = 0;
9: .... /* dalsze deklaracje bez zmian */
10:
11: .... /* procedury bez zmian */
12: vec3 NormalTex ( float dx, float dy ) { .... }
13: float attFactor ( vec3 att, float dist ) { .... }
14:
15: float IsEnlighted ( uint l )
16: {
17:   float s;
18:
19:   s = textureProj ( shtex[l], In.ShadowPos[l] );
20:   if ( s > 0.0 )
21:     return s*textureProj ( parttex[l], In.ShadowPos[l] ).r;
22:   else
23:     return 0.0;
24: } /*IsEnlighted*/
25:
26: .... /* wszystkie pozostałe procedury bez zmian */
```

Na listingu 24.17 są przedstawione zmiany wprowadzone do szadera z listingu 22.5. W linii 7 jest nowa tablica ewaluatorów, dla tekstur mgły.

Aby dostosować szader fragmentów do rysowania cienia mgły na końcowym obrazie, wystarczyło zmienić funkcję `IsEnlighted`. Jeśli na podstawie tekstury cienia przetwarzany

fragment jest zasłonięty od światła przez powierzchnię jakiegoś obiektu, to jest w cieniu i już. W przeciwnym razie brana jest pod uwagę wartość tekstury mgły. Im mgła jest gęstsza lub im jej warstwa jest grubsza, tym więcej światła pochłania, a więc tym słabiej jest oświetlona powierzchnia w danym punkcie. Wartość tekstury mgły wystarczy zatem uwzględnić jako czynnik wartości funkcji `IsEnlighted`.

24.7. Pozostałe zmiany w aplikacji

Do struktury `AppData` aplikacji 2H wystarczy dodać tylko dwa pola opisujące układ cząsteczek i programy jego symulacji i rysowania (listing 24.18) oraz pole `particles`. Gdy pole to ma wartość 0, to animacja wygląda tak, jak w aplikacji 2H, z torusem wskakującym do i wyskakującym z czajnika. Gdy ma ono wartość 1, torus pozostaje w czajniku, a z dziobka wylatuje mgła.

Listing 24.18. Zmiany w strukturze `AppData`

C

```
typedef struct {
    Camera                camera;
    kl_linkage            *linkage;
    KLBezPatches          bezp[2];
    PartSystem            ps;
    Mirror                mirror;
    TransBl               trans;
    LightBl               light;
    MatBl                 mat;
    GLuint                lktrbuf;
    GLint                 BezNormals, TessLevel;
    char                  cnet, skeleton, shadows, particles,
                          animate, hold, final;
    double                hold_time, hold_time0;
    double                teapot_rot_angle;
    BPRenderPrograms      brprog;
    GLuint                miprog[2];
    KLArticulationProgram artprog;
    PartSysPrograms       psprog;
  } AppData;
```

Ponieważ inicjalizacja układu cząsteczek jest dokonywana przez opisaną wcześniej procedurę — konstruktor obiektu przywiązanego do łańcucha kinematycznego, do procedury inicjalizacji danych opisujących obiekty do rysowania trzeba tylko dodać wywołanie procedury kompilującej szadery przeznaczone do symulacji i rysowania cząsteczek (zob. listing 24.19). Instrukcja ta musi być dodana przed wywołaniem procedury `ConstructMyLinkage`. Ponadto polom `particles` i `hold` trzeba nadać wartość początkową `false`, a polu `hold_time` wartość 0.

Listing 24.19. Procedura `InitMyWorld`

```C
void InitMyWorld ( int argc, char *argv[], int width, int height )
{
  GLfloat ident_matrix[16];

  memset ( &appdata, 0, sizeof(AppData) );
  LoadBPShaders ( &appdata.bprprog );
  LoadMirrorShaders ( appdata.miprog );
  LoadLinkageArticulationProgram ( &appdata.artprog );
  LoadParticleShaders ( &appdata.psprog );
  appdata.trans.trbuf = NewUniformTransBlock ();
  ....
  appdata.hold_time = 0.0;
  appdata.cnet = appdata.skeleton = appdata.particles =
    appdata.animate = appdata.hold = false;
  ....
  if ( !ConstructMyLinkage ( &appdata ) )
    ExitOnError ( "InitMyWorld" );
  ArticulateMyLinkage ( appdata.linkage );
} /*InitMyWorld*/
```

Jeśli symulacja układu cząsteczek jest włączona, to lepiej, aby torus pozostawał we wnętrzu czajnika, bo jego oddziaływania na strumień pary wylatującej z dziobka nasz model nie jest w stanie dobrze naśladować. Zatem, gdy zmienna `appdata.particles` ma wartość `true`, wówczas funkcja `TorusRotAngle1` ma stałą wartość 1.84 i torus kręci się wewnątrz czajnika.

Listing 24.20. Reakcja na kolejne klawisze

```C
char ProcessCharCommand ( char charcode )
{
  switch ( toupper ( charcode ) ) {
.... /* wszystko, co było wcześniej, zostało niezmienione */
case 'P':
    if ( (appdata.particles = !appdata.particles) )
      ResetParticles ( &appdata.psprog, &appdata.ps, app_time );
    return true;
case 'R':
    if ( appdata.particles )
      ResetParticles ( &appdata.psprog, &appdata.ps, app_time );
    return appdata.particles;
case 'X':
    return HoldOnOff ( &appdata );
default:
    return false;
  }
} /*ProcessCharCommand*/
```

Do procedury `ProcessCharCommand` zostały dodane instrukcje reagowania na kolejne znaki napisane na klawiaturze (listing 24.20). Napisanie litery `P` powoduje zmianę wartości zmiennej `appdata.particles`, co skutkuje włączeniem lub wyłączeniem symulacji układu cząsteczek i rysowania mgły. Jeśli symulacja została włączona, to wywoływana jest procedura `ResetParticles` z parametrem podającym czas, który upłynął od początku działania aplikacji; czas ten został zmierzony przez procedurę artykulacji łańcucha kinematycznego podczas ostatniego wywołania procedury `MoveOn`. Napisanie litery `R`, gdy symulacja układu cząsteczek jest włączona, powoduje restart, czyli nadanie wszystkim cząsteczkom początkowego ujemnego wieku (innego dla każdej cząsteczki), dzięki czemu można obserwować wylatywanie mgły z dziobka od początku.

Listingi 24.21 i 24.22 przedstawiają procedury realizujące wstrzymywanie i wznawianie animacji w reakcji na napisanie litery `X`. Wywoływana wtedy procedura `HoldOnOff`, gdy animacja jest wstrzymywana (czyli gdy zmienna `hold` otrzymuje wartość `true`), zapamiętuje w polu `hold_time0` bieżący odczyt zegara, tj. wartość zmiennej `app_time` nadaną przez procedurę `TimerToc`. Wartość powrotna `false` zawiadamia część okienkową, że nie trzeba wykonywać nowego obrazu (gdy animacja jest wstrzymana, również procedura `MoveOn` przekazuje wartość `false`). Wznowienie animacji powoduje odczytanie zegara, a potem obliczenie i dodanie do zmiennej `hold_time` czasu, przez który animacja była wstrzymana.

Listing 24.21. Procedury realizujące wstrzymywanie animacji

C

```
char HoldOnOff ( AppData *ad )
{
  if ( (ad->hold = !ad->hold) ) {
    TimerToc ();
    ad->hold_time0 = app_time;
    return false;
  }
  else {
    TimerTic ();
    ad->hold_time += app_time - ad->hold_time0;
    return true;
  }
} /*HoldOnOff*/

void RedrawMyWorld ( void )
{
  if ( !appdata.hold )
    ArticulateMyLinkage ( appdata.linkage );
  glEnable ( GL_DEPTH_TEST );
  DrawSceneToShadows ( &appdata );
  DrawSceneToMirror ( &appdata );
  DrawSceneToWindow ( &appdata );
} /*RedrawMyWorld*/

```

```
25: char MoveOn ( void )
26: {
27:   return !appdata.hold;
28: } /*MoveOn*/
```

Wywołanie procedury `ArticulateMyLinkage` przed rysowaniem (wykonywanym zawsze po zmianie położenia obserwatora, po zmianie wielkości okna lub po zmianie modelu oświetlenia albo któregoś parametru wpływającego na wygląd obrazu) następuje pod warunkiem, że animacja nie jest wstrzymana (linia 17). W samej tej procedurze parametry artykulacji są obliczane na podstawie czasu od uruchomienia aplikacji pomniejszonego o sumę długości przedziałów czasowych, w których animacja była wstrzymana.

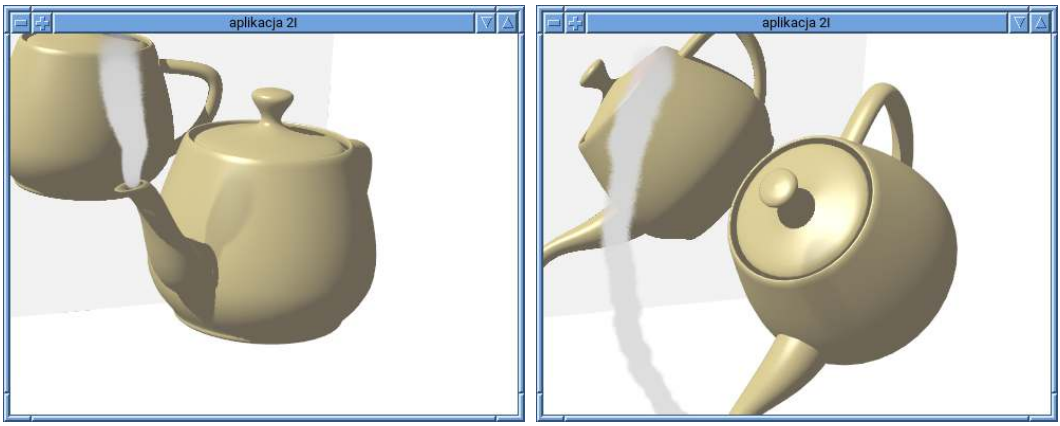
Listing 24.22. Procedura `ArticulateMyLinkage`

C

```
1: void ArticulateMyLinkage ( kl_linkage *lk )
2: {
3:   AppData *ad;
4:   double  at, dt, par[9];
5:
6:   ad = (AppData*)lk->usrdata;
7:   dt = TimerTocTic ();
8:   at = app_time - ad->hold_time;
9:   if ( ad->animate ) {
10:    if ( (ad->teapot_rot_angle += ANGULAR_VELOCITY1 * dt) >= PI )
11:      ad->teapot_rot_angle -= 2.0*PI;
12:  }
13:  par[0] = ad->teapot_rot_angle;
14:  par[1] = TeapotRotAngle2 ( at );
15:  par[2] = SpoutAngle ( at );
16:  par[3] = -0.5*par[2];
17:  par[4] = LidAngle1 ( at );
18:  par[5] = LidAngle2 ( at );
19:  par[7] = -(par[6] = TorusRotAngle1 ( at , ad->particles ) );
20:  par[8] = -2.0*PI*at ;
21:  kl_SetArtParam ( lk, 0, 9, par );
22:  kl_Articulate ( lk );
23: } /*ArticulateMyLinkage*/
```

24.8. Ćwiczenia

1. Zmodyfikuj szader z listingu 24.3, aby zależnie od tego, czy cząsteczka jest w obszarze cienia, czy nie, narysować ją w odpowiednim (jaśniejszym lub ciemniejszym) kolorze. W tym celu szader musi mieć dostęp do tekstur obszarów cienia.



Rysunek 24.3. Okno aplikacji drugiej I

2. Rozbuduj model ruchu cząsteczek o wiry. Wir jest nowym rodzajem cząsteczek wylatujących z dziobka czajnika, przy czym oprócz położenia i prędkości ma też oś (określoną przez pewien niezerowy wektor) i siłę. Wirów powinno być kilkanaście lub kilkadziesiąt i powinny wylatywać z dziobka co pewien czas. Wiry nie mają być rysowane, ale mają wpływać na ruch znajdujących się w pobliżu cząsteczek, dodając odpowiednią prędkość i przyspieszenie, aby otrzymać ruch obrotowy wokół osi wiru. Układ cząsteczek z wirami może dać bardziej realistyczny efekt symulacji.
3. Wymyśl sposób, zaprogramuj i uruchom wysyłanie cząsteczek spod pokrywki czajnika zamiast z dziobka, gdy pokrywka jest uchylona.
4. *Zaimplementuj równoległy generator liczb pseudolosowych działający na GPU i wytwarzający ciąg o długości M w $\lceil \log_2 M \rceil + 2$ krokach. Korzystając z łatwego do sprawdzenia faktu, że dla dowolnych liczb naturalnych a, b oraz N ma miejsce równość

   $$(a \bmod N)(b \bmod N) = (ab) \bmod N,$$

   z której wynika, że mnożenie modulo N jest działaniem łącznym, zmodyfikuj algorytm obliczania sum prefiksowych opisany w podrozdziale G.2, aby otrzymać w buforze magazynowym ciąg liczb $a_i = s_0 t^i \bmod 2^{32}$, określony przez ziarno początkowe s_0 i mnożnik t takie jak w podrozdziale 24.3. Następnie zamień liczby a_i na ciągi bitów reprezentujące liczby zmiennopozycyjne z przedziału $[1, 2)$.

24.9. *Uzupełnienia

24.9.1. Funkcje mieszające

Kolor przekazany na wyjście szadera fragmentów może być (po przejściu fragmentu przez testy widoczności, nożyczek i maski) przypisany pikselowi lub zmieszany w pewnych pro-

porcjach z poprzednim kolorem piksela — najczęściej przydaje się to do otrzymania efektów takich jak przezroczystość (nie tylko mgły, ale także powierzchni). Mieszanie kolorów jest wykonywane osobno dla każdej ze składowych r, g, b i a, zgodnie ze wzorem

$$C = sS + dD,$$

w którym symbole S, D i C oznaczają odpowiednio składową koloru podanego przez szader fragmentów (*source*), składową dotychczasowego koloru piksela (*destination*) i składową koloru, który ma pikselowi być przypisany (*colour*). Współczynniki s i d proporcji, w jakiej mieszane są poszczególne składowe, są liczbami z przedziału $[0,1]$, przy czym mogą one być ustalane dla każdej składowej osobno lub takie same dla wszystkich składowych. Określa się je, wywołując procedurę `glBlendFunc`, której pierwszy parametr ustala sposób wyboru współczynnika s, a drugi d. Repertuar dla obu współczynników jest taki sam: jest kilkanaście dopuszczalnych wartości każdego z tych parametrów. Mają one nazwy symboliczne nadane przez odpowiednie makrodefinicje w pliku nagłówkowym OpenGL-a.

Na przykład parametry `GL_ZERO` i `GL_ONE` ustalają wartość 0 i 1 współczynnika s lub d dla wszystkich czterech składowych. Parametry `GL_SRC_COLOR` i `GL_DST_COLOR` określają indywidualny współczynnik dla każdej składowej równy odpowiedniej składowej (czyli S albo D), poddanej normalizacji[12]. Parametry `GL_SRC_ALPHA` i `GL_DST_ALPHA` określają ten sam współczynnik dla wszystkich czterech składowych na podstawie składowej A odpowiedniego koloru. Parametry `GL_ONE_MINUS_SRC_COLOR`, `GL_ONE_MINUS_DST_COLOR`, `GL_ONE_MINUS_SRC_ALPHA` i `GL_ONE_MINUS_DST_ALPHA` odejmują od liczby 1 współczynnik odpowiadający jednemu z czterech wyborów opisanych wcześniej. Dokładniejszy opis tych i pozostałych dopuszczalnych parametrów procedury `glBlendFunc` można znaleźć w specyfikacji [1] lub na stronie [7].

Każda składowa koloru wynikowego po obliczeniu jest obcinana do przedziału $[0,1]$, a następnie konwertowana do postaci, w jakiej będzie przechowywana w pikselu (czyli np. mnożona przez 255 i zamieniana na liczbę ośmiobitową, jeśli tyle bitów w pikselu jest dla niej przewidziane). Zwracam uwagę, że to wprowadza błędy (zaokrągleń), które mogą mieć istotny wpływ na końcowy obraz i dlatego może okazać się potrzebne wykonywanie obrazu w pozaekranowym buforze ramki z załącznikiem — buforem obrazu — o większej dokładności (np. o składowych reprezentowanych jako 32-bitowe liczby zmiennopozycyjne). Obraz można potem przesłać do okna za pomocą procedury `glBlitFramebuffer`

24.9.2. Zanurzanie buforów w przestrzeń adresową CPU

Bufor w pamięci GPU można (na chwilę) zanurzyć w przestrzeń adresową aplikacji działającej na CPU; wtedy ma ona do niego taki sam dostęp jak do dowolnego obszaru pamięci zarezerwowanego za pomocą procedury `malloc`. Temu służą procedury `glMapBuffer` i `glMapBufferRange`, podające adres początku znajdującego się w pamięci GPU bufora, do którego

[12] Składowe r, g, b, a pikseli najczęściej są reprezentowane przez liczby całkowite, na przykład od 0 do 255, jeśli jest 8 bitów na składową. Wtedy liczba odczytana z piksela jest dzielona przez 255, czego wynikiem jest współczynnik z przedziału $[0,1]$.

CPU uzyskuje dostęp. Po zapisaniu lub odczytaniu zawartości bufora aplikacja powinna bez zbędnej zwłoki zlikwidować to zanurzenie, wywołując procedurę `glUnmapBuffer`; wcześniej uzyskany adres staje się wtedy całkowicie nieaktualny. W czasie, gdy bufor jest widoczny w przestrzeni adresowej CPU, nie należy wykonywać programów szaderów, które się do niego odwołują.

Opisany wyżej mechanizm wydaje się wygodny, zwłaszcza jeśli bufor jest duży i alternatywne rozwiązanie, tj. użycie procedury `malloc` do zajęcia obszaru pamięci o takiej wielkości jak bufor, zapisanie w tym obszarze danych i użycie przesyłającej dane do bufora procedury `glBufferData` lub `glBufferSubData`, albo przesłanie danych w drugą stronę za pomocą procedury `glGetBufferSubData`, wydaje się zbyt skomplikowane. Osobiście jestem jednak skłonny polecać właśnie to rozwiązanie (jest ono używane w aplikacji opisanej w tym rozdziale) z prostego powodu: bezpośredni dostęp CPU do pamięci GPU jest znacznie (o rzędy wielkości) wolniejszy niż dostęp do RAM CPU, a także dostęp podsystemu komputera służącego do przesyłania danych przez magistralę między CPU a GPU do pamięci po obu stronach. W przeprowadzonym eksperymencie wypełniałem dwa bufory w pamięci GPU punktami o losowych współrzędnych; punktów tych było w każdym buforze 2^{23} (czyli ponad 8 milionów), przy czym każdy punkt (w $\mathbb{R}^4$) zajmował 16 bajtów. Do przesłania było zatem 256 MB danych.

Odpowiedni fragment programu napisałem w trzech wersjach. W pierwszym podejściu skorzystałem z procedur `glMapBufferRange` i `glUnmapBuffer` i wygenerowane dane wpisywałem bezpośrednio do buforów. W drugim sposobie generowałem w pętli po jednym punkcie i natychmiast przesyłałem opisujące ten punkt 16 bajtów do bufora za pomocą procedury `glBufferSubData`, która w związku z tym została wywołana 16777216 razy. Trzecie rozwiązanie polegało na zarezerwowaniu przez `malloc` tablicy o długości 128 MB, dwukrotnym wypełnieniu jej danymi i przesłaniu tych danych do buforów za pomocą `glBufferSubData`. Czasy, które zmierzyłem, były takie: w pierwszym przypadku 34.22 s, w drugim 3.09 s, a w trzecim 0.94 s, z czego 0.9 s zajęło generowanie danych przy użyciu generatora liczb losowych opisanego w podrozdziale 24.3. Wnioski proszę wyciągnąć samemu[13].

[13] W książce [64] można znaleźć opis sposobu przesyłania danych w aplikacjach Vulkana, z wykorzystaniem bufora pomocniczego (do którego CPU ma szybki dostęp) i procedury szybko kopiującej dane do bufora docelowego. Można to zrobić, bo standard Vulkan umożliwia określenie rodzaju pamięci rezerwowanej dla bufora (a dokładniej, zmusza autora aplikacji do zajmowania się takimi szczegółami). W aplikacji OpenGL-a można za pomocą procedury `glBufferStorage` utworzyć bufor pomocniczy znajdujący się w RAM CPU (po szczegóły odsyłam do specyfikacji [1]) i korzystać z procedury przesyłającej dane między buforami, ale to niweczy uproszczenie kodu aplikacji.

25

Aplikacja druga J

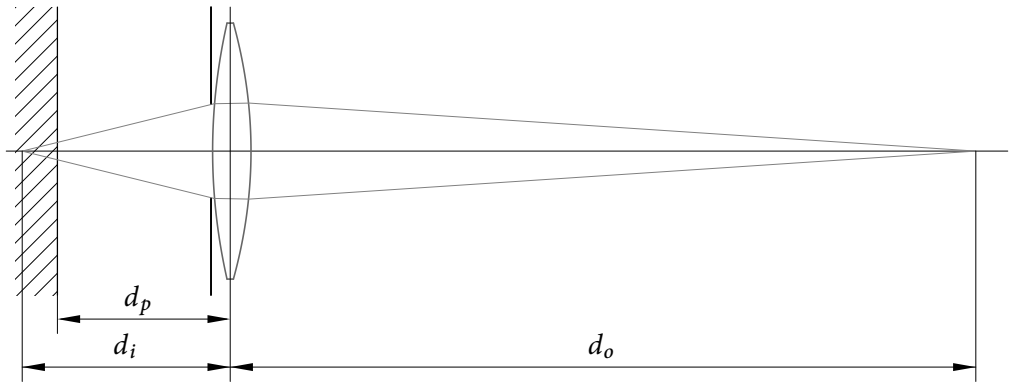
Bufor akumulacji (*accumulation buffer*) jest to zdeprecjonowany element starego OpenGL-a przeznaczony do mieszania w dowolnych proporcjach wielu obrazów. Można dzięki niemu osiągać rozmaite efekty, w tym antyaliasing (w nowym OpenGL-u tę rolę przejął mechanizm wielokrotnego próbkowania, zobacz rozdział 19), a także głębię ostrości (*depth of field*) i rozmycie obiektów w ruchu (*motion blur*), które również jest antyaliasingiem, choć w dziedzinie czasu, nie w przestrzeni. Funkcjonalność bufora akumulacji można osiągnąć przy użyciu pozaekranowego bufora ramki. Posługując się nim, aplikacja opisana w tym rozdziale wytwarza efekty głębi ostrości i rozmycia przedmiotów w ruchu.

Są dwa powody, dla których bufor akumulacji został zdeprecjonowany. Pierwszy to ten, że łatwo jest go zaimplementować środkami dostępnymi w nowym OpenGL-u, co tu zrobimy. Drugi powód jest taki, że efekt głębi ostrości może być osiągnięty przez **rysowanie na wielu warstwach** (*multilayer rendering*), przy czym ta technika umożliwia wykonanie obrazów w krótszym czasie. Użyjemy jej w następnym rozdziale[1].

25.1. Podstawy symulacji głębi ostrości

Fotografia dowolnego przedmiotu powinna uwagę osoby oglądającej skupiać na tym przedmiocie. Temu służy nastawienie odległości od obiektywu do tego przedmiotu. Obiekty znajdujące się bliżej lub dalej oraz tło *powinny* być nieostre — bo tak działa układ optyczny aparatu fotograficznego, i ludzkiego oka też. Aby osiągnąć ten efekt w grafice, można zmieszać pewną liczbę obrazów sceny wykonanych tak, aby tylko obrazy punktów znajdujących się w ustalonej odległości pokrywały się, dzięki czemu końcowe obrazy tych punktów będą ostre. W tym celu przed wykonaniem każdego obrazu będziemy modyfikować położenie obserwatora (tj. środek rzutowania perspektywicznego) i ostrosłup widzenia.

[1] Jeszcze inna możliwość, dająca jednak obrazy o gorszej jakości, to filtrowanie obrazu wykonanego w zwykły sposób za pomocą filtru określonego przez zawartość bufora głębokości. Niedoskonałości tej techniki są najbardziej widoczne na krawędziach sylwetek obiektów. Ponadto jeśli scena zawiera lustro, w którym są widoczne odbicia innych obiektów, to otrzymany przez filtrowanie efekt głębi ostrości w lustrze będzie niepoprawny.



Rysunek 25.1. Model obiektywu dla symulacji głębi ostrości

Na rysunku 25.1 jest przedstawiony schemat symulowanego „układu optycznego", którego obiektyw składa się z jednej soczewki skupiającej. Równanie soczewki jest następujące:

$$\frac{1}{d_o} + \frac{1}{d_i} = \frac{1}{F}.$$

Symbole d_o i d_i oznaczają odległości od obiektywu (umieszczonego w środku rzutowania) pewnego punktu i jego ostrego obrazu wytworzonego przez ten obiektyw. Symbol F oznacza długość ogniskową obiektywu.

Jeśli rzutnia jest umieszczona w odległości $d_p \neq d_i$ od obiektywu, to utworzony na niej obraz punktu położonego w odległości d_o jest plamką o średnicy

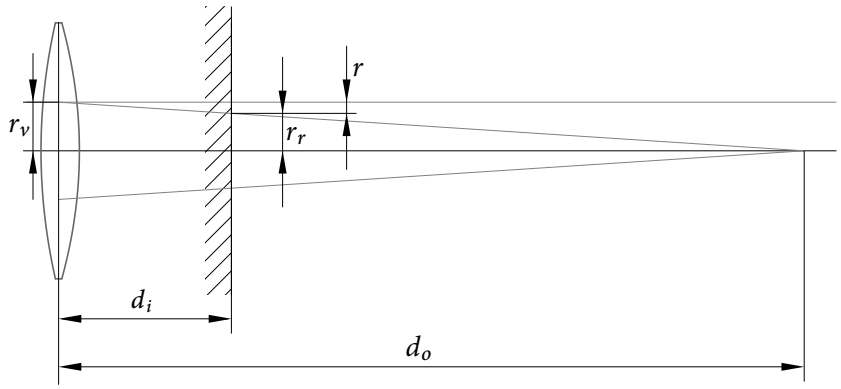
$$c_d = \left|1 - \frac{d_p}{d_i}\right| \frac{F}{N}, \tag{25.1}$$

gdzie N oznacza nastawienie przysłony[2]; średnica otworu przysłony jest równa $\frac{F}{N}$.

Uwaga: Opisany tu algorytm nie zapewnia otrzymania plamek o średnicy określonej wzorem (25.1) dla punktów w odległościach innych niż odległość nastawienia ostrości (dalej oznaczana symbolem d_o) i ∞, ale błąd aproksymacji jest w praktyce niezauważalny.

Wykonując prosty eksperyment myślowy, umieścimy rzutnię w odległości d_i *przed* obiektywem i będziemy rozważać tylko promienie światła przechodzące przez środek soczewki i przez punkt soczewki przesunięty względem środka równolegle do rzutni na odległość r_v. Promień z punktu osi optycznej położonego w odległości $d_o < \infty$ od soczewki przejdzie przez rzutnię w odległości r_r od osi optycznej. Jeśli chcemy na obrazach otrzymanych w rzutach perspektywicznych ze środka soczewki i z punktu w odległości r_v otrzymać ten sam punkt

[2]Gęstość mocy światła padającego na rzutnię (czyli na macierz CCD lub na błonę światłoczułą w aparacie fotograficznym) jest proporcjonalna do kwadratu bezwymiarowego współczynnika N, dlatego to ten współczynnik jest ustawiany na podstawie wskazań światłomierza.



Rysunek 25.2. Przesunięcia obserwatora i rzutni w symulacji głębi ostrości

(aby nastawić ostrość na odległość d_o), to rzutnię (razem z pierwszym obrazem) trzeba przesunąć na odległość r_r. Z podobieństwa trójkątów na rysunku 25.2 musimy przyjąć

$$\frac{r_r}{r_v} = \frac{d_o - d_i}{d_o}.$$

Stąd i z równania soczewki wynika

$$r_r = \left(1 - \frac{d_i}{d_o}\right) r_v = \left(1 - \frac{F}{d_o - F}\right) r_v.$$

Odległość r obrazów „punktu w nieskończoności", czyli takiego, z którego dochodzą równoległe promienie światła, na obrazach wyznaczonych przez promień oryginalny i przesunięty, jest różnicą wielkości przesunięć środka rzutowania i rzutni, czyli

$$r = r_v - r_r = r_v \frac{F}{d_o - F}.$$

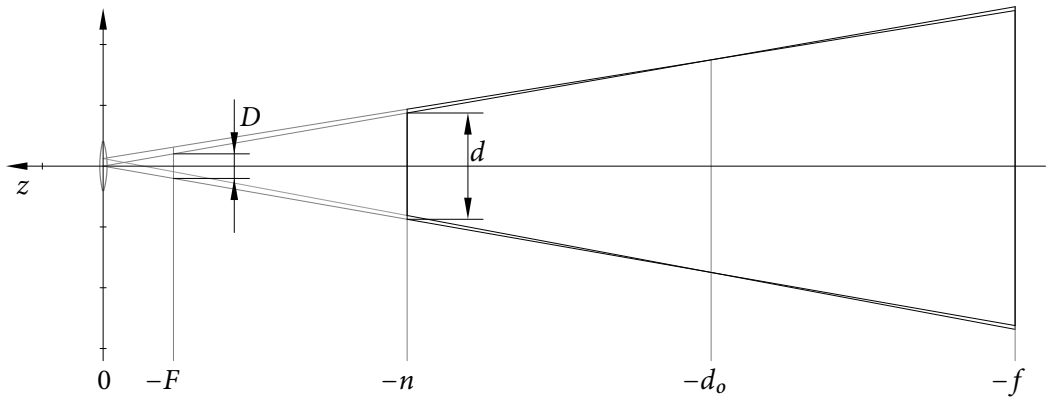
Przypuśćmy, że $r = r_{\max}$ jest promieniem plamki będącej obrazem punktu w nieskończoności (którego ostry obraz powstaje w odległości F od obiektywu), gdy otwór przysłony ma promień $\frac{F}{2N}$. Wtedy

$$r_{\max} = \left|1 - \frac{F}{d_i}\right| \frac{F}{2N} = \left|1 - \frac{d_o - F}{d_o}\right| \frac{F}{2N} = \frac{F^2}{2d_o N}.$$

Równość wyrażeń opisujacych odległość r i promień plamki $r_{\max}$ prowadzi do otrzymania wzoru na maksymalne przesunięcie r_v:

$$r_{v\max} = \frac{F}{2N} \frac{d_o - F}{d_o}. \tag{25.2}$$

Znając przesunięcie obserwatora, $r_v \in [0, r_{v\max}]$, potrzebujemy znaleźć przesunięcie przedniej ściany ostrosłupa widzenia dające efekt równoważny przesunięciu rzutni na odległość r_r. Dla wygody długość ogniskową F będziemy dalej podawać w milimetrach, odnosząc ją do wymiarów klatki w małoobrazkowym aparacie fotograficznym[3], ale we wzorach tu wyprowadzonych wszystkie wymiary są podane w jednostkach układu świata. Symbole l, r, b, t, n i f w tych wzorach oznaczają współrzędne wierzchołków ostrosłupa widzenia opisane w rozdziale 6.



Rysunek 25.3. Przekrój przez bryły widzenia dla dwóch położeń obserwatora

Rysunek 25.3 przedstawia przekrój ostrosłupa widzenia płaszczyzną zawierającą przekątną jego przedniej (i tylnej) ściany. Na osi z układu współrzędnych obserwatora są zaznaczone punkty $-n$ i $-f$ (wyznaczające płaszczyzny przedniej i tylnej ściany ostrosłupa) oraz punkt $-F$, do którego (w nowym eksperymencie myślowym) przesunęliśmy rzutnię[4] i punkt $-d_o$, na którego odległość od obiektywu jest nastawiona ostrość. Długości D i d przekątnych „klatki aparatu fotograficznego" i przedniej ściany ostrosłupa widzenia pozostają w proporcji

$$\frac{d}{D} = \frac{n}{F}.$$

Jeśli klatka w oknie ma szerokość w pikseli i wysokość h pikseli, to szerokość i wysokość przedniej ściany (w jednostkach układu obserwatora, takich samych jak w układzie świata), $r - l$ i $t - b$, pozostają w proporcji

$$\frac{r-l}{t-b} = \frac{aw}{h},$$

[3] Czyli 36 mm × 24 mm. Dzięki tej umowie możemy stosować tradycyjny podział obiektywów na standardowe, $F = 50$ mm, szerokokątne z $F < 50$ mm i długoogniskowe ($F > 50$ mm) oraz specjalny sprzęt dla *paparazzi* ($F \geqslant 400$ mm).

[4] W dokładniejszym modelu należałoby rzutnię przesunąć do punktu $-d_i$, czyli przyjąć dalej $\frac{d}{D} = \frac{n}{d_i}$, ale to nie wniosłoby żadnej nowej jakości i w szczególności nie pomogłoby odtworzyć powodowanych przez prawdziwe obiektywy ze szklanymi soczewkami dystorsji i aberracji, nie mówiąc już o astygmatyzmie.

gdzie a oznacza współczynnik aspektu ekranu. Stąd możemy wyznaczyć

$$d = \sqrt{(r-l)^2 + (t-b)^2} = (t-b)\sqrt{1 + (aw/h)^2} = n\frac{D}{F}$$

i obliczyć na tej podstawie

$$t - b = \frac{n}{\sqrt{1 + (aw/h)^2}}\frac{D}{F} \quad \text{oraz} \quad r - l = (t-b)\frac{aw}{h}.$$

Jeśli ostrosłup widzenia w położeniu wyjściowym ma być symetryczny, to przyjmiemy następnie $t = -b = (t-b)/2$ oraz $r = -l = (r-l)/2$. Jeśli teraz na potrzeby symulacji głębi ostrości mamy przesunąć środek rzutowania o wektor $(x_v, y_v, 0)$ (dany w układzie współrzędnych obserwatora w położeniu wyjściowym), to aby uzyskać ostrość obrazów punktów w płaszczyźnie $z = -d_o$, należy zmienne l i r zmniejszyć o $x_v n/d_o$, a od zmiennych b i t trzeba odjąć $y_v n/d_o$. Na listingu 25.1 jest przedstawiona procedura, która konstruuje macierze przekształceń potrzebnych do rzutowania na podstawie wyprowadzonych wyżej wzorów.

Parametry `w`, `h` podają wymiary klatki w pikselach, `aspect` określa współczynnik aspektu ekranu. Parametr `F` podaje długość ogniskową *w milimetrach*. Pozostałe parametry (z wyjątkiem `xp`, `yp`) są podawane w jednostkach układu świata. Parametr `dist` określa odległość nastawienia ostrości, parametry `near` i `far` wyznaczają zakres odległości bryły widzenia. Parametry `xv` i `yv` są współrzędnymi wektora przesunięcia środka rzutowania (prostopadle do osi optycznej kamery). Parametry `xp` i `yp` określają dodatkowe przesunięcie obrazu rastrowego, które może być wykorzystane do symulacji wielokrotnego próbkowania w celu przeprowadzenia antyaliasingu. Współrzędne tego przesunięcia podaje się w jednostkach szerokości i wysokości jednego piksela.

Macierz `vm` określa przejście od układu współrzędnych świata do nominalnego (tj. nieprzesuniętego) układu obserwatora. Pozostałe parametry są wyjściowe: `left`, `right`, `bottom`, `top` wskazują zmienne, którym należy przypisać współrzędne l, r, b i t odpowiednich wierzchołków ostrosłupa widzenia. Do tablicy `shvm` procedura ma wpisać macierz przejścia do przesuniętego układu obserwatora, przy czym jeśli parametr `vm` jest wskaźnikiem pustym, to do tablicy `shvm` jest wpisywana macierz samego przejścia od nominalnego do przesuniętego układu obserwatora. Jeśli wskaźniki `vm` i `eyepos` są niepuste, to procedura jeszcze obliczy i wpisze do tablicy `eyepos` współrzędne w układzie świata początku układu przesuniętego obserwatora. Do tablic `pm` oraz `pmi` procedura wpisuje macierz przejścia od układu przesuniętego obserwatora do związanego z nowym ostrosłupem widzenia układu kostki standardowej i jej odwrotność. Parametry wskaźnikowe `left`, `right`, `bottom`, `top`, `shvm`, `eyepos`, `pm` i `pmi` mogą mieć wartość `NULL`, jeśli wyprowadzana za pomocą któregoś z tych parametrów informacja jest aplikacji niepotrzebna.

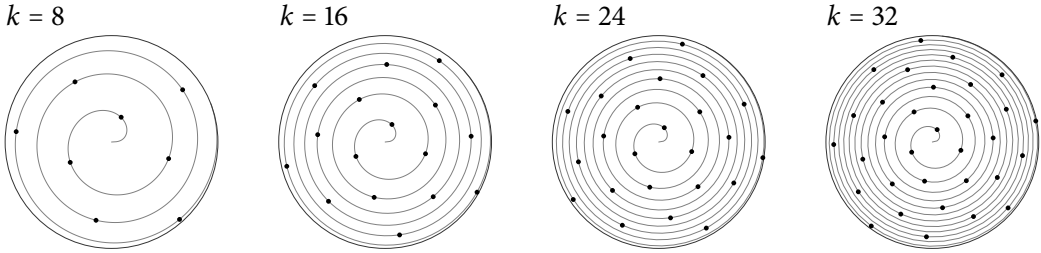
W linii 13 jest obliczany iloraz długości przekątnej klatki (36 mm×24 mm) i podanej w milimetrach długości ogniskowej obiektywu. Pozostałe instrukcje, realizujące wyprowadzone wcześniej wzory, powinny być oczywiste.

Listing 25.1. Procedura `M4x4SkewFrustumf`

C

```
void M4x4SkewFrustumf ( int w, int h, float aspect, float F, float dist,
                        float xv, float yv, float xp, float yp,
                        float near, float far,
                        float *left, float *right, float *bottom, float *top,
                        const GLfloat vm[16],
                        GLfloat shvm[16], GLfloat eyepos[4],
                        GLfloat pm[16], GLfloat pmi[16] )
{
  float   DF, awh, rl, tb, r, l, t, b, xr, yr;
  GLfloat am[16], v[4];
  int     p[3];

  DF = sqrt ( (double)(36*36 + 24*24) ) / F;
  awh = aspect*(float)w/(float)h;
  tb = DF*near/sqrt ( 1.0 + awh*awh );
  rl = tb*awh;
  xr = xv*near/dist - xp*rl/(float)w;
  l = -(r = 0.5*rl);  l -= xr;  r -= xr;
  yr = yv*near/dist - yp*tb/(float)h;
  b = -(t = 0.5*tb);  b -= yr;  t -= yr;
  if ( top ) *top = t;
  if ( left ) *left = l;
  if ( right ) *right = r;
  if ( bottom ) *bottom = b;
  if ( shvm ) {
    if ( vm ) {
      M4x4Translatef ( am, -xv, -yv, 0.0 );
      M4x4Multf ( shvm, am, vm );
      if ( eyepos ) {
        M4x4LUDecompf ( am, p, shvm );
        v[0] = v[1] = v[2] = 0.0;  v[3] = 1.0;
        M4x4LUSolvef ( eyepos, am, p, v );
      }
    }
    else
      M4x4Translatef ( shvm, xv, yv, 0.0 );
  }
  M4x4Frustumf ( pm, pmi, l, r, b, t, near, far );
} /*M4x4SkewFrustumf*/
```

Algorytm głębi ostrości wymaga wykonania pewnej liczby obrazów (rzędu kilku do kilkunastu), przy czym dla każdego z nich trzeba wybrać inne przesunięcie środka rzutowania. Przesunięte środki rzutowania powinny leżeć w kole o promieniu $r_{v\max}$ opisanym wzorem (25.2). Sposób wyboru punktów w kole powinien zapewniać ich rozmieszczenie z równomierną gęstością, przy czym niewskazane jest wybieranie wierzchołków regularnej



Rysunek 25.4. Rozmieszczenie przesuniętych środków rzutowania w kole

siatki. Wypróbowany przeze mnie (z dobrym skutkiem) sposób rozmieszczenia k punktów w kole (rys. 25.4) polega na użyciu wzorów

$$x_{vi} = r_i \cos\varphi_i, \quad y_{vi} = r_i \sin\varphi_i, \quad i = 0, \ldots, k-1, \tag{25.3}$$

w których $r_i = r_{v\max}\sqrt{(2i+1)/(2k)}$, $\varphi_i = (i+\frac{1}{2})2\pi\tau^2$, $\tau = (\sqrt{5}-1)/2$. Liczba τ jest proporcją złotego podziału (któremu poddawany jest kąt pełny). Otrzymane w ten sposób punkty są rozmieszczone zgodnie z zasadą filotaksji [30].

25.2. Implementacja bufora akumulacji

Reprezentacja obrazu przeznaczona do wyświetlania na ekranie jest nieodpowiednia do obliczeń wykonywanych w buforze akumulacji i w wielu innych zastosowaniach związanych z przetwarzaniem obrazów. Składowe *r*, *g*, *b* koloru piksela są w niej reprezentowane przez 8 (a znacznie rzadziej przez 10) bitów, zamienianych bezpośrednio na sygnał sterujący monitorem. Taka reprezentacja zajmuje stosunkowo mało pamięci, ale z powodów opisanych w podrozdziale C.3 potrzebna jest korekcja gamma, która jest nieliniowym odwzorowaniem jasności punktów obrazu na liczby reprezentujące składowe koloru. Do sumowania i innych działań na obrazach potrzebne jest odwzorowanie liniowe, większa dokładność i możliwość reprezentowania liczb (nawet znacznie) większych niż maksymalny poziom składowej, a czasami także ujemnych.

Bufory przechowujące obrazy w postaci gotowej do wyświetlenia na ekranie lub zapisania w pliku mają własność zwaną po angielsku *low dynamic range*, *LDR*, co moim zdaniem można przetłumaczyć na **wąski zakres dynamiczny**. Bufor akumulacji musi mieć **szeroki zakres dynamiczny** (*high dynamic range*, *HDR*), realizowany w praktyce za pomocą 32-bitowych liczb zmiennopozycyjnych. Wprawdzie zajmuje to czterokrotnie więcej miejsca (co obecnie nie jest problemem), ale zapewnia wystarczający zakres i dokładność reprezentacji.

Podstawowe działania na buforze akumulacji to skasowanie bufora, dodanie nowego obrazu, pomnożenie obrazu w buforze przez pewną stałą, dodanie nowego obrazu pomnożonego przez stałą, a także korekcję gamma; ten repertuar można oczywiście rozszerzać do woli. Można napisać osobny szader dla każdego działania, ale ja napisałem jeden szader, który wykona działanie wybrane przy użyciu służącej do tego zmiennej jednolitej.

Listing 25.2. Szader obliczeniowy bufora akumulacji

GLSL

```
#version 450 core

#define ACCBUF_NONE     0
#define ACCBUF_CLEAR    1
#define ACCBUF_ADD      2
#define ACCBUF_MULT     3
#define ACCBUF_MULT_ADD 4
#define ACCBUF_GAMMA    5

layout(local_size_x=1) in;
layout(rgba32f,binding=0) uniform image2D accb;
layout(rgba32f,binding=1) uniform image2D img;
uniform int   action;
uniform float fct;

#define AGamma(colour) pow ( colour, vec3(256.0/563.0) )

void main ( void )
{
  ivec2 xy;
  vec4  pix;

  xy = ivec2 ( gl_GlobalInvocationID.xy );
  switch ( action ) {
case ACCBUF_CLEAR:
    pix = vec4 ( 0.0, 0.0, 0.0, 0.0 );
    break;
case ACCBUF_ADD:
    pix = imageLoad ( accb, xy ) + imageLoad ( img, xy );
    break;
case ACCBUF_MULT:
    pix = fct * imageLoad ( accb, xy );
    break;
case ACCBUF_MULT_ADD:
    pix = imageLoad ( accb, xy ) + fct * imageLoad ( img, xy );
    break;
case ACCBUF_GAMMA:
    pix = vec4 ( AGamma ( imageLoad ( accb, xy ).rgb, 1.0 );
    break;
default:
    return;
  }
  imageStore ( accb, xy, pix );
} /*main*/
```

Pokazany na listingu 25.2 szader wykonuje działanie na dwuwymiarowym **obrazie** (*image*). Obraz jest prostokątną tablicą o ustalonych rozmiarach, której elementami są piksele o ustalonym formacie; z takich tablic (w najprostszym przypadku z jednej) składają się tekstury[5]. Dowolny obraz wchodzący w skład tekstury można przywiązać do odpowiedniej **jednostki obrazu** (*image unit*), co umożliwia szaderom dostęp do pikseli. Zmienne jednolite `accb` i `img` typu `image2D`, zadeklarowane w liniach 11 i 12, zgodnie z kwalifikatorami `binding=0` i `binding=1` są związane z jednostkami obrazu o numerach 0 i 1. Kwalifikator `rgba32f` określa, że piksele mają 4 składowe reprezentowane jako liczby zmiennopozycyjne. Obraz dostępny przez zmienną `accb` jest buforem akumulacji, natomiast zmienna `img` daje dostęp do drugiego obrazu, który można będzie dodać do bufora.

Opisane dalej procedury w C zapewniają, że oba obrazy mają identyczne wymiary. W linii 23 wątek szadera odczytuje swoje miejsce w globalnej grupie roboczej — to są współrzędne piksela, który ten wątek ma przetworzyć.

Zależnie od wybranego (za pomocą wartości zmiennej `action`) działania, w liniach 29, 32, 35 albo 38 szader pobiera z obrazów dane za pomocą funkcji `imageLoad` i wykonuje działanie na pikselu, po czym procedura `imageStore` zapisuje jego wynik w buforze akumulacji.

Na listingu 25.3 są pokazane procedury przygotowania i likwidacji programu szaderów z szaderem obliczeniowym z listingu 25.2. Po skompilowaniu i złączeniu programu procedura `LoadAccumBufShaders` wyciąga położenia jego zmiennych jednolitych `action` i `fct`. Działanie procedury `DeleteAccumBufShaders` nie wymaga objaśnień.

Listing 25.3. Tworzenie i likwidacja programu szaderów bufora akumulacji

C

```
1: static GLuint accprid;
2: static GLuint action_loc, fct_loc;
3:
4: char LoadAccumBufShaders ( void )
5: {
6:   static const char *filename[] = { "accbuf0.comp.glsl" };
7:   static const char *uvname[] = { "action", "fct" };
8:   GLuint shader_id;
9:
10:  shader_id = CompileShaderFiles ( GL_COMPUTE_SHADER, 1, &filename[0] );
11:  accprid = LinkShaderProgram ( 1, &shader_id, "acc" );
12:  action_loc = glGetUniformLocation ( accprid, uvname[0] );
13:  fct_loc = glGetUniformLocation ( accprid, uvname[1] );
14:  glDeleteShader ( shader_id );
15:  ExitIfGLError ( "LoadAccumBufShaders" );
16:  return true;
17: } /*LoadAccumBufShaders*/
18:
```

[5]Tu słowo to oznacza teksturę w rozumieniu OpenGL-a, nie zaś w znaczeniu matematycznym, tj. funkcję opisującą jakąś własność powierzchni wpływającą na wygląd tej powierzchni.

```
19: void DeleteAccumBufShaders ( void )
20: {
21:   glUseProgram ( 0 );
22:   glDeleteProgram ( accprid );
23:   ExitIfGLError ( "DeleteAccumBufShaders" );
24: } /*DeleteAccumBufShaders*/
```

Objaśnień za to wymagają przedstawione na listingu 25.4 procedury, które tworzą obiekt bufora akumulacji. Potrzebne będą dwa bufory ramki. Tekstura, która zawiera obraz używany jako bufor akumulacji, jest załącznikiem pierwszego bufora ramki. Po utworzeniu końcowego obrazu w buforze akumulacji obraz ten będzie przesłany do okna na ekranie; w OpenGL-u jest przeznaczona do tego procedura `glBlitFramebuffer`, która przesyła dane między buforami ramki. Drugi bufor ramki służy do tworzenia obrazów, które będą zbierane w buforze akumulacji.

Polem zmiennej typu `AccumBuf` jest tablica `fbo`, której elementy to identyfikatory obu buforów ramki. Tekstury, których identyfikatory są w tablicy `txt`, to: tekstura z obrazem używanym jako bufor akumulacji, tekstura, na której powstaje kolejny obraz, dodawany następnie do bufora akumulacji, i tekstura pełniąca rolę bufora głębokości. Pola `width` i `height` służą do przechowania szerokości i wysokości wszystkich tych tekstur.

Pomocnicza procedura `AllocAccTextures` jest wywoływana po utworzeniu buforów ramki (przez procedurę `NewAccumBuf`) i po zmianie wymiarów okna, gdy trzeba do niego dostosować wymiary obrazów. W linii 9 procedura rezerwuje identyfikatory trzech tekstur. W linii 11 pierwsza z tych tekstur otrzymuje konkretny kształt, tj. szerokość i wysokość oraz określenie wewnętrznej postaci tekseli — mają one po 4 składowe reprezentowane w postaci zmiennopozycyjnej.

Uwaga: Choć w specyfikacji [1] można przeczytać, że procedura `glTexImage2D` dokonuje rezerwacji pamięci GPU na obraz, to obraz wchodzący w skład tekstury utworzonej przez kolejno wywołane procedury `glGenTextures`, `glBindTexture` i `glTexImage2D` nie jest widoczny dla szadera. Zamiast `glTexImage2D` trzeba użyć procedury `glTexStorage2D`. Cały dzień mi zajęło odkrycie tej informacji[6].

W liniach 12 i 13 utworzona tekstura staje się załącznikiem pierwszego bufora ramki, po czym sprawdzana jest jego kompletność. W liniach 18–21 są tworzone pozostałe dwie tekstury, co w szczególności obejmuje rezerwację pamięci na ich obrazy, a następnie tekstury te stają się załącznikami drugiego bufora ramki — w pierwszej będą składowe *r*, *g*, *b*, *a* pikseli, a druga tekstura stanie się buforem głębokości. Na koniec trzeba odwiązać bufor ramki (co przywraca domyślny stan, w którym rysowanie odbywa się w buforze ramki związanym z oknem) i odwiązać teksturę od celu `GL_TEXTURE_2D`, a potem jeszcze zapamiętać wymiary tekstur w zmiennej wskazywanej przez pierwszy parametr procedury.

[6] Czyli zgadnięcie, co trzeba zrobić, żeby wreszcie obraz widziany przez szader miał szerokość i wysokość większe niż 0 pikseli.

Listing 25.4. Procedury tworzenia i likwidacji bufora akumulacji

C

```
typedef struct {
    GLuint fbo[2];
    GLuint txt[3];
    int    width, height;
  } AccumBuf;

static char AllocAccTextures ( AccumBuf *acb, int w, int h )
{
  glGenTextures ( 3, acb->txt );
  glBindTexture ( GL_TEXTURE_2D, acb->txt[0] );
  glTexStorage2D ( GL_TEXTURE_2D, 1, GL_RGBA32F, w, h );
  glBindFramebuffer ( GL_FRAMEBUFFER, acb->fbo[0] );
  glFramebufferTexture ( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                         acb->txt[0], 0 );
  if ( glCheckFramebufferStatus ( GL_FRAMEBUFFER ) !=
       GL_FRAMEBUFFER_COMPLETE )
    ExitOnError ( "AllocAccTextures 0" );
  glBindTexture ( GL_TEXTURE_2D, acb->txt[1] );
  glTexStorage2D ( GL_TEXTURE_2D, 1, GL_RGBA32F, w, h );
  glBindTexture ( GL_TEXTURE_2D, acb->txt[2] );
  glTexStorage2D ( GL_TEXTURE_2D, 1, GL_DEPTH_COMPONENT32F, w, h );
  glBindFramebuffer ( GL_FRAMEBUFFER, acb->fbo[1] );
  glFramebufferTexture ( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                         acb->txt[1], 0 );
  glFramebufferTexture ( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                         acb->txt[2], 0 );
  if ( glCheckFramebufferStatus ( GL_FRAMEBUFFER ) !=
       GL_FRAMEBUFFER_COMPLETE )
    ExitOnError ( "AllocAccTextures 1" );
  glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
  glBindTexture ( GL_TEXTURE_2D, 0 );
  acb->width = w;  acb->height = h;
  ExitIfGLError ( "AllocAccTextures" );
  return true;
} /*AllocAccTextures*/

AccumBuf *NewAccumBuf ( int w, int h )
{
  AccumBuf *acb;

  if ( (acb = malloc ( sizeof(AccumBuf) )) ) {
    glGenFramebuffers ( 2, acb->fbo );
    if ( AllocAccTextures ( acb, w, h ) )
      return acb;
    else {
```

```
46:       glDeleteFramebuffers ( 2, acb->fbo );
47:       free ( acb );
48:     }
49:   }
50:   return NULL;
51: } /*NewAccumBuf*/
52:
53: char SetAccumBufSize ( AccumBuf *acb, int w, int h )
54: {
55:   if ( w != acb->width || h != acb->height ) {
56:     glDeleteTextures ( 3, acb->txt );
57:     if ( !AllocAccTextures ( acb, w, h ) ) {
58:       glDeleteFramebuffers ( 2, acb->fbo );
59:       free ( acb );
60:       return false;
61:     }
62:   }
63:   return true;
64: } /*SetAccumBufSize*/
65:
66: void DeleteAccumBuf ( AccumBuf *acb )
67: {
68:   glDeleteFramebuffers ( 2, acb->fbo );
69:   glDeleteTextures ( 3, acb->txt );
70:   free ( acb );
71: } /*DeleteAccumBuf*/
```

Procedura `NewAccumBuf` rezerwuje pamięć na zmienną typu `AccumBuf`, tworzy bufory ramki i wywołuje opisaną wyżej procedurę `AllocAccTextures`. Procedura `SetAccumBufSize` likwiduje tekstury, aby na ich miejsce utworzyć nowe tekstury o wielkości określonej przez parametry. Procedura `DeleteAccumBuf` likwiduje tekstury i bufory ramki, a potem zwalnia pamięć zajmowaną przez zmienną typu `AccumBuf`.

Na listingu 25.5 są przedstawione procedury wykonujące działania na buforze akumulacji. Identyfikatory tych działań, wprowadzone w liniach 1–6, są identyczne z identyfikatorami na listingu 25.2. Każda z tych procedur przywiązuje bufor akumulacji (czyli obraz z tekstury `acb->txt[0]`) do zmiennej jednolitej `accb`, za pomocą procedury `glBindImageTexture`, której pierwszy parametr jest numerem jednostki obrazu — to jest numer podany w kwalifikatorze `layout` w deklaracji zmiennej `accb` szadera obliczeniowego, czyli `0`. Procedury `AccumBufAdd` i `AccumBufMultAdd` do zmiennej `img` (powiązanej z jednostką obrazu `1`) przywiązują teksturę `acb->txt[1]`, w której jest nowy obraz do dodania do bufora akumulacji. Wartości nadane zmiennym jednolitym `action` i `fct` wybierają działanie i określają stały czynnik, po czym szader z listingu 25.2 przystępuje do pracy. Każdy wątek szadera ma do przetworzenia jeden piksel, którego współrzędne są numerami wątku w globalnej grupie roboczej — dwuwymiarowej, o wymiarach takich jak obraz.

Listing 25.5. Procedury działań na buforze akumulacji

C

```
#define ACCBUF_NONE     0
#define ACCBUF_CLEAR    1
#define ACCBUF_ADD      2
#define ACCBUF_MULT     3
#define ACCBUF_MULT_ADD 4
#define ACCBUF_GAMMA    5

void AccumBufClear ( AccumBuf *acb )
{
  glUseProgram ( accprid );
  glBindImageTexture ( 0, acb->txt[0], 0, GL_FALSE, 0,
                       GL_WRITE_ONLY, GL_RGBA32F );
  glUniform1i ( action_loc, ACCBUF_CLEAR );
  glDispatchCompute ( acb->width, acb->height, 1 );
  glMemoryBarrier ( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
  ExitIfGLError ( "AccumBufClear" );
} /*AccumBufClear*/

void AccumBufAdd ( AccumBuf *acb )
{
  glUseProgram ( accprid );
  glBindImageTexture ( 0, acb->txt[0], 0, GL_FALSE, 0, GL_READ_WRITE,
                       GL_RGBA32F );
  glBindImageTexture ( 1, acb->txt[1], 0, GL_FALSE, 0, GL_READ_ONLY,
                       GL_RGBA32F );
  glUniform1i ( action_loc, ACCBUF_ADD );
  glDispatchCompute ( acb->width, acb->height, 1 );
  glMemoryBarrier ( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
  ExitIfGLError ( "AccumBufAdd" );
} /*AccumBufAdd*/

void AccumBufMult ( AccumBuf *acb, GLfloat a )
{
  glUseProgram ( accprid );
  glBindImageTexture ( 0, acb->txt[0], 0, GL_FALSE, 0, GL_READ_WRITE,
                       GL_RGBA32F );
  glUniform1i ( action_loc, ACCBUF_MULT );
  glUniform1f ( fct_loc, a );
  glDispatchCompute ( acb->width, acb->height, 1 );
  glMemoryBarrier ( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
  ExitIfGLError ( "AccumBufMult" );
} /*AccumBufMult*/

void AccumBufMultAdd ( AccumBuf *acb, GLfloat a )
{
```

```
46:   glUseProgram ( accprid );
47:   glBindImageTexture ( 0, acb->txt[0], 0, GL_FALSE, 0, GL_READ_WRITE,
48:                        GL_RGBA32F );
49:   glBindImageTexture ( 1, acb->txt[1], 0, GL_FALSE, 0, GL_READ_ONLY,
50:                        GL_RGBA32F );
51:   glUniform1i ( action_loc, ACCBUF_MULT_ADD );
52:   glUniform1f ( fct_loc, a );
53:   glDispatchCompute ( acb->width, acb->height, 1 );
54:   glMemoryBarrier ( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
55:   ExitIfGLError ( "AccumBufMultAdd" );
56: } /*AccumBufMultAdd*/
57:
58: void AccumBufGamma ( AccumBuf *acb )
59: {
60:   glUseProgram ( accprid );
61:   glBindImageTexture ( 0, acb->txt[0], 0, GL_FALSE, 0, GL_READ_WRITE,
62:                        GL_RGBA32F );
63:   glUniform1i ( action_loc, ACCBUF_GAMMA );
64:   glDispatchCompute ( acb->width, acb->height, 1 );
65:   glMemoryBarrier ( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
66:   ExitIfGLError ( "AccumBufGamma" );
67: } /*AccumBufGamma*/
```

Wywołanie procedury `glMemoryBarrier` ma na celu zaczekanie, aż wszystkie wątki szadera obliczeniowego dokończą swoją pracę i zapiszą wyniki w buforze akumulacji. Dopiero wtedy można wykonać następne działanie lub odczytać zawartość bufora akumulacji w celu przesłania jej na ekran.

25.3. Obliczanie parametrów rzutowania

Wprowadzenie modelu „aparatu fotograficznego" w aplikacji 2J jest dobrym powodem do zmiany sposobu obliczania parametrów ostrosłupa widzenia. W opisie sceny i jej odwzorowania na ekran są w użyciu trzy jednostki długości: milimetry, jednostki układu świata[7] i wymiary piksela. O te ostatnie zadbał producent monitora, natomiast ustalenie długości w milimetrach jednostki układu świata jest kwestią umowy, od której zależy wielkość obiektów na obrazie.

Jeśli jednostka układu świata ma ustaloną liczbę milimetrów (czyli także pikseli), to obiekty na obrazie (widziane z pewnego punktu) będą zachowywać wielkość podczas zmieniania wymiarów okna. Na przykład przyjęcie, że jednostka układu świata ma 108 mm odtwarza wielkość oryginalnego czajnika z porcelany, który ma 6 cali (152.4 mm) wysokości, a jego model ma w układzie świata (wszystkich wersji drugiej aplikacji) wysokość 1.4. Wtedy w małym oknie będzie można zobaczyć tylko fragment czajnika. Lepszym pomysłem wydaje się więc uzależnienie jednostki układu świata od wymiarów okna, bo wtedy ich zmiana

[7] Mowa o świecie, w którym są opisane obiekty do narysowania, nie o świecie, w którym znajduje się komputer, monitor, użytkownik i milimetry. W tej aplikacji jednostki układu świata są równe jednostkom układu obserwatora — przejścia między układami świata a obserwatora są izometriami.

spowoduje analogiczną zmianę wielkości obiektów na obrazie. Modyfikowanie tej zależności umożliwia wykonywanie najazdów i odjazdów, podobnie jak podczas używania kamery z obiektywem o zmiennej ogniskowej.

Makrodefinicje na listingu 25.6 opisują zakresy istotnych wymiarów i innych parametrów kamery. Założyłem, że użytkownik patrzy na monitor z odległości 80 cm, czyli 800 mm, przy czym przednia i tylna ściana ostrosłupa widzenia znajdują się w odległości 40 cm od płaszczyzny ekranu przed i za nią, tj. w odległościach 400 mm i 1200 mm od obserwatora; te wymiary są zapisane w makrodefinicjach `SCREEN_DIST`, `MIN_FOV_DIST` i `MAX_FOV_DIST`. Dwa ostatnie wymiary wyznaczają także zakres możliwych nastawień ostrości.

Listing 25.6. Typ `Camera` i związane z nim makrodefinicje oraz zmieniony typ `TransBl`

C

```
#define MIN_FOV_DIST          400.0
#define MAX_FOV_DIST         1200.0
#define FOCUS_FCT               1.05
#define MIN_FRAME_HEIGHT        0.1
#define MAX_FRAME_HEIGHT       20.0
#define INITIAL_FRAMEHEIGHT     2.2
#define ZOOM_FCT                1.05
#define MIN_APERTURE            2.0
#define MAX_APERTURE           32.0
#define INITIAL_APERTURE        8.0
#define SCREEN_DPI            118.0
#define SCREEN_DPMM           (SCREEN_DPI/25.4)
#define SCREEN_DIST           800.0

typedef struct Camera {
    int    win_width, win_height;
    float  left, right, bottom, top, near, far, rl, tb;
    float  frameheight, unitmm;
    float  F, N, dist;
    int    distcnt, heightcnt, Ncnt;
    float  viewer_pos0[4];
    float  viewer_rvec[3];
    double viewer_rangle;
  } Camera;

typedef struct TransBl {
    GLfloat mm[16], mmti[16], wvm0[16], wvm1[16], wpm1[16],
            mvm[16], mpm[16];
    GLfloat eyepos0[4], eyepos1[4], reyepos[4];
  } TransBl;
```

Nowa (porównaj z listingiem 15.13) struktura typu `Camera` używana do opisu położenie obserwatora i bryły widzenia ma dodatkowe pola `frameheight`, `unitmm`, `F`, `N`, `dist`, `distcnt`, `heightcnt` i `Ncnt`, w których są przechowywane odpowiednio: wysokość okna aplikacji, długość jednostki układu świata w milimetrach, długość ogniskowa obiektywu,

względny otwór przysłony, odległość nastawienia ostrości oraz wykładniki używane w sposób opisany dalej do obliczania odległości, wysokości klatki i względnego otworu przysłony.

Liczby 0.1 i 20.0, ukryte pod nazwami `MIN_FRAME_HEIGHT` i `MAX_FRAME_HEIGHT`, określają przedział wartości pola `frameheight`. Wartość tego pola jest wyrażoną w jednostkach układu świata wysokością prostokąta będącego przekrojem ostrosłupa widzenia płaszczyzną równoległą do jego przedniej i tylnej ściany, położonego w odległości `SCREEN_DIST` milimetrów od początku układu obserwatora. Ze wzrostem tego parametru obrazy obiektów maleją.

Parametry `MIN_APERTURE` i `MAX_APERTURE` określają zakres nastawień przysłony, tj. parametru N, który jest wartością pola `N`.

Pod nazwą `SCREEN_DPI` jest ukryta rozdzielczość ekranu[8], wyrażona w pikselach na cal. Makro `SCREEN_DPMM` ukrywa wyrażenie, którego wartością jest rozdzielczość ekranu w pikselach na milimetr. Przyjąłem tu jednakową rozdzielczość ekranu w poziomie i pionie, czyli współczynnik aspekt równy 1.

Pola zmienionej struktury typu `TransBl` służą do przechowywania reprezentacji przekształceń określonych przez nominalne i przesunięte położenia obserwatora. W tablicy `wvm0` jest przechowywana macierz widoku, tj. przejścia od układu współrzędnych świata do układu nieprzesuniętego obserwatora, a w tablicach `wvm1` i `wpm1` są macierze widoku i rzutowania dla obserwatora przesuniętego.

Procedury przedstawione na listingu 25.7 mają za zadanie obliczyć parametry i wymiary potrzebne do skonstruowania macierzy rzutowania. Wywoływana przez `InitMyWorld` procedura `InitCamera` nadaje wartości początkowe polom zmiennej opisującej kamerę. Wykładniki `distcnt`, `heightcnt` i `Ncnt` otrzymują wartość 0. Początkowa wysokość okna (w jednostkach układu świata) jest równa 2.2, ostrość jest nastawiana na `SCREEN_DIST` (tj. na 800 mm), a początkowy otwór przysłony $N = 8$. W linii 19 następuje obliczenie długości w milimetrach jednostki układu świata. Komentarza wymaga instrukcja w linii 21: dotychczas obserwator był umieszczony w odległości 10 jednostek układu świata od początku układu świata (i obracając się wokół sceny, pozostawał w tej odległości). Teraz obserwator jest w odległości 800 mm, która właśnie w tym miejscu jest wyrażana w (obowiązujących w danej chwili) jednostkach układu świata. Takie samo obliczenie wykonuje następnie procedura `AdjustDimensions` (w linii 74) przed każdym rysowaniem sceny.

Do procedury `ComputeEyePosition` zostały przeniesione końcowe instrukcje procedury `_RotateViewer`, ponieważ teraz te instrukcje są również częścią obliczenia wykonywanego przez procedurę `AdjustDimensions`. Procedura ta w linii 62 oblicza długość przekątnej okna w milimetrach. W linii 63 jest obliczana długość ogniskowa obiektywu, który dla sceny obserwowanej z odległości 800 mm utworzyłby obraz, na którym przekątna okna miałaby długość przekątnej klatki małoobrazkowej 36 mm × 24 mm. W liniach 64 i 65 są obliczane parametry n i f (w jednostkach układu świata) wyznaczające płaszczyzny przednią i tylną ostrosłupa widzenia[9].

[8] mojego monitora

[9] Do tego miejsca zawsze było $n = 5$, $f = 15$, ale teraz odległość obserwatora od sceny, wyrażona w jednostkach układu świata, już nie jest stała, a więc i te parametry będą się zmieniać.

Listing 25.7. Procedury obliczające parametry kamery

C

```
static void _ResizeMyWorld ( AppData *ad, int width, int height )
{
  SetAccumBufSize ( ad->acb, ad->camera.win_width = width,
                    ad->camera.win_height = height );
  if ( ad->particles )
    ResetParticles ( &ad->psprog, &ad->ps, app_time );
} /*_ResizeMyWorld*/

static void InitCamera ( AppData *ad, int width, int height )
{
  static const float viewer_rvec[3] = {1.0,0.0,0.0};
  static const float viewer_pos0[4] = {0.0,0.0,10.0,1.0};

  memcpy ( ad->camera.viewer_rvec, viewer_rvec, 3*sizeof(float) );
  memcpy ( ad->camera.viewer_pos0, viewer_pos0, 4*sizeof(float) );
  ad->camera.viewer_rangle = 0.0;
  ad->camera.distcnt = ad->camera.heightcnt = ad->camera.Ncnt = 0;
  ad->camera.frameheight = INITIAL_FRAMEHEIGHT;
  ad->camera.unitmm = (float)height/(SCREEN_DPMM*ad->camera.frameheight);
  ad->camera.dist = SCREEN_DIST;
  ad->camera.viewer_pos0[2] = ad->camera.dist / ad->camera.unitmm;
  ad->camera.N = INITIAL_APERTURE;
  memcpy ( ad->trans.eyepos0, ad->camera.viewer_pos0, 4*sizeof(GLfloat) );
  M4x4InvTranslatefv ( ad->trans.wvm0, ad->camera.viewer_pos0 );
  _ResizeMyWorld ( ad, width, height );
} /*InitCamera*/

static void ComputeEyePosition ( Camera *camera, TransBl *trans )
{
  M4x4RotateVfv ( trans->wvm0, camera->viewer_rvec,
                  -camera->viewer_rangle );
  M4x4MultMTVf ( trans->eyepos0, trans->wvm0, camera->viewer_pos0 );
  M4x4InvTranslateMfv ( trans->wvm0, camera->viewer_pos0 );
} /*ComputeEyePosition*/

static void _RotateViewer ( AppData *ad,
                            double delta_xi, double delta_eta )
{
  float   vi[3], lgt, vk[3];
  double  angi, angk;

  vi[0] = (float)delta_eta*ad->camera.rl/(float)ad->camera.win_height;
  vi[1] = (float)delta_xi*ad->camera.tb/(float)ad->camera.win_width;
  vi[2] = 0.0;
  lgt = sqrt ( V3DotProductf ( vi, vi ) );
```

```
46:   vi[0] /= lgt;  vi[1] /= lgt;
47:   angi = -0.052359878;  /* -3 stopnie */
48:   V3CompRotationsf ( vk, &angk, ad->camera.viewer_rvec,
49:                       ad->camera.viewer_rangle, vi, angi );
50:   memcpy ( ad->camera.viewer_rvec, vk, 3*sizeof(float) );
51:   ad->camera.viewer_rangle = angk;
52:   ComputeEyePosition ( &appdata.camera, &appdata.trans );
53: } /*_RotateViewer*/
54:
55: static void AdjustDimensions ( Camera *camera, TransBl *trans )
56: {
57:   float   w, h, windiag;
58:   GLfloat wvm[16], wpm[16];
59:
60:   w = (float)camera->win_width;  h = (float)camera->win_height;
61:   camera->unitmm = h/(SCREEN_DPMM*camera->frameheight);
62:   windiag = sqrt ( (float)(w*w + h*h) ) / SCREEN_DPMM;
63:   camera->F = SCREEN_DIST/windiag*sqrt ( 36.0*36.0+24.0*24.0 );
64:   camera->near = MIN_FOV_DIST / camera->unitmm;
65:   camera->far = MAX_FOV_DIST / camera->unitmm;
66:   if ( camera->far < 10.0 )
67:     camera->far = 10.0;
68:   M4x4SkewFrustumf ( camera->win_width, camera->win_height, 1.0,
69:         camera->F, camera->dist / camera->unitmm, 0.0, 0.0, 0.0, 0.0,
70:         camera->near, camera->far, &camera->left, &camera->right,
71:         &camera->bottom, &camera->top, trans->wvm0, wvm, NULL, wpm, NULL );
72:   camera->rl = camera->right - camera->left;
73:   camera->tb = camera->top - camera->bottom;
74:   camera->viewer_pos0[2] = SCREEN_DIST / camera->unitmm;
75:   ComputeEyePosition ( camera, trans );
76: } /*AdjustDimensions*/
```

W liniach 66–67 jest dokonywana korekta parametru f; powiększenie jednostki układu świata (przez powiększenie okna lub najazd kamerą, tj. zmniejszenie długości ogniskowej) powoduje przybliżenie obserwatora do środka sceny, bo teraz 800 mm jest równe mniejszej liczbie jednostek tego układu. Odległość 1200 mm (`MAX_FOV_DIST`) też maleje (w porównaniu z wielkością obiektów), co może spowodować obcięcie części sceny położonej najdalej od obserwatora. Problem (dla sceny rysowanej przez tę aplikację) rozwiązuje zadbanie o to, aby tylna ściana ostrosłupa widzenia była oddalona od obserwatora co najmniej o 10 jednostek układu świata[10].

Wywołanie procedury `M4x4SkewFrustumf` w liniach 68–71 ma na celu obliczenie wymiarów przedniej ściany ostrosłupa widzenia (w jednostkach układu świata) — dla nominalnego (tj. nieprzesuniętego) położenia obserwatora.

[10] Ten sam parametr f jest używany w konstrukcji macierzy rzutowania obrazu odbitego w lustrze.

25.4. Dalsze zmiany w aplikacji

Listing 25.8 przedstawia pola dodane do struktury typu `AppData`; jest w niej m.in. wskaźnik struktury opakowującej bufor akumulacji oraz tablica `blurpermut`, zawierająca pewną permutację ciągu liczb $0, \ldots, k-1$; długość k tego ciągu jest zapamiętaną w polu `acc_samples` liczbą obrazów, które mają być zmieszane w buforze akumulacji. Permutacja ta jest potrzebna do poprawnego rozmycia obiektów w ruchu. Opis sposobu jej otrzymania i użycia w generowaniu obrazów jest dalej. Pole `blur` jest przełącznikiem rozmycia obiektów w ruchu. Pole `part_time` służy do przekazywania czasu procedurze symulacji układu cząsteczek.

Listing 25.8. Zmieniony typ `AppData`

C

```
#define MIN_ACC_SAMPLES     8
#define MAX_ACC_SAMPLES    32

typedef struct {
    Camera                camera;
    kl_linkage            *linkage;
    KLBezPatches          bezp[2];
    PartSystem            ps;
    Mirror                mirror;
    TransBl               trans;
    LightBl               light;
    MatBl                 mat;
    GLuint                lktrbuf;
    GLint                 BezNormals, TessLevel;
    char                  cnet, skeleton, shadows, particles,
                          animate, hold, blur, final;
    double                hold_time, hold_time0, part_time;
    double                teapot_rot_angle;
    AccumBuf              *acb;
    int                   acc_samples, blurpermut[MAX_ACC_SAMPLES];
    BPRenderPrograms      brprog;
    GLuint                miprog[2];
    KLArticulationProgram artprog;
    PartSysPrograms       psprog;
  } AppData;
```

Na listingu 25.9 jest przedstawiona procedura `SetTransformations`, której zadaniem jest skonstruowanie na podstawie wzorów (25.2) i (25.3) macierzy widoku i macierzy rzutowania dla kolejnego przesuniętego położenia obserwatora; parametr `i` określa, które to jest położenie.

W liniach 16 i 17 procedura oblicza długość ogniskową i odległość nastawienia ostrości w jednostkach układu świata, a w linii 18 wielkość maksymalnego przesunięcia środka rzutowania, $r_{v\max}$. W linii 19 jest obliczana długość i-tego przesunięcia, a w liniach 22–23 współrzędne wektora przesunięcia w nominalnym układzie obserwatora. W liniach 24–28

jest wywołanie przedstawionej wcześniej procedury `M4x4SkewFrustumf`. Zwracam uwagę, że jej czwarty parametr, `camera.F`, podaje długość ogniskową w *milimetrach*.

Listing 25.9. Procedura konstrukcji macierzy przekształceń

C

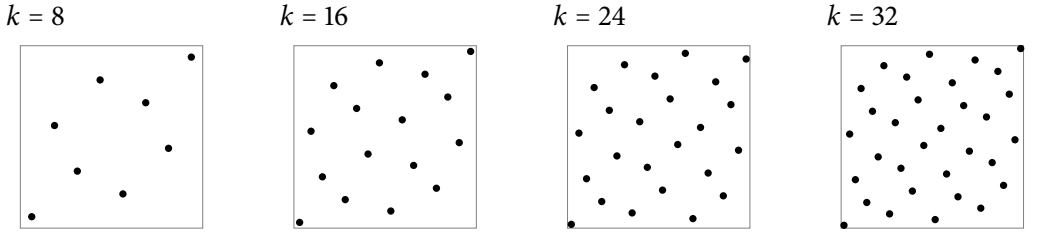
```
float RadicalInversion ( unsigned int i, unsigned int base )
{
  float x, b, c;

  for ( x = 0.0, c = b = 1.0/(float)base;  i > 0;  i /= base, c *= b )
    x += (float)(i % base) * c;
  return x;
} /*RadicalInversion*/

void SetTransformations ( Camera *camera, TransBl *trans,
                          int acc_samples, int i )
{
  float F, dist, rvmax, ri, tau, phii, xvi, yvi;
  float l, r, b, t;

  F = camera->F/camera->unitmm;
  dist = camera->dist/camera->unitmm;
  rvmax = F*(dist-F)/(2.0*camera->N*dist);
  ri = rvmax * sqrt ( (float)(i+i+1)/(float)(2*acc_samples) );
  tau = 0.5*(sqrt ( 5.0 ) - 1.0);
  phii = (float)(i+i+1)*PI*tau*tau;
  xvi = ri*cos ( phii );
  yvi = ri*sin ( phii );
  M4x4SkewFrustumf ( camera->win_width, camera->win_height, 1.0,
         camera->F, dist, xvi, yvi, ((float)i+0.5)/acc_samples-0.5,
         RadicalInversion ( i, 2 ) + 0.5*(1.0-1.0/(float)acc_samples),
         camera->near, camera->far, &l, &r, &b, &t, trans->wvm0,
         trans->wvm1, trans->eyepos1, trans->wpm1, NULL );
  SetupMirrorVPMatrices ( camera,
      trans->eyepos1, trans->reyepos, trans->mvm, trans->mpm, camera->far );
} /*SetTransformations*/
```

Ósmy i dziewiąty parametr procedury `M4x4SkewFrustumf` określają dodatkowe przesunięcie obrazu na rzutni. Ma ono na celu antyaliasing, a ściślej mówiąc, otrzymanie skutku równoważnego użyciu obrazu wielopróbkowego. Bez tego przesunięcia dla każdego obrazu otrzymanego przy użyciu konstruowanych tu macierzy kolor piksela byłby kolorem odpowiadającym punktowi w środku piksela. Dodatkowe (w stosunku do przesuwania na potrzeby algorytmu głębi ostrości) przesuwanie rzutni dla poszczególnych obrazów o ułamki wymiarów piksela spowoduje, że na obrazie otrzymanym w buforze akumulacji w każdym pikselu otrzymamy uśredniony kolor różnych (odpowiednio rozmieszczonych) punktów, a właśnie do tego sprowadza się filtrowanie obrazu podczas antyaliasingu. Punkty te

powinny być rozmieszczone w prostokącie o wielkości piksela z równomierną gęstością, ale nie powinny być punktami regularnej siatki. Dlatego współrzędne wektorów przesunięć poszczególnych obrazów są obliczane na podstawie wzorów

$$x_{p,i} = \frac{2i+1}{2k-1} - \frac{1}{2}, \quad y_{p,i} = \text{Inversion}(i;2) + \frac{k-1}{2k}, \quad i = 0, \ldots, k-1.$$

Funkcja Inversion jest określona następująco: jeśli $i = \sum_{j=0}^{m} d_j b^j$, $d_j \in \{0, \ldots, b-1\}$, czyli liczba naturalna i ma w układzie pozycyjnym o podstawie b zapis $d_m d_{m-1} \ldots d_1 d_0$, to $\text{Inversion}(i;b) = \sum_{j=0}^{m} d_j b^{-j-1}$; jest to więc ułamek, którego zapis w tym samym układzie pozycyjnym to $0.d_0 d_1 \ldots d_{m-1} d_m$. Na przykład $\text{Inversion}(2314;10) = 0.4132$. Do obliczania wartości tej funkcji służy podprogram `RadicalInversion`. Na rysunku 25.5 są pokazane położenia otrzymanych w ten sposób punktów obliczania koloru w pikselu dla różnych liczb k.



Rysunek 25.5. Przesunięcia rzutni o ułamki piksela

Procedura `SetupMirrorVPMatrices`, wywołana w liniach 29–30, oblicza macierze widoku i rzutowania dla obserwatora odbitego w lustrze oraz współrzędne w układzie świata położenia odbitego obserwatora. Odbiciu podlega obserwator przesunięty. Zapewnia to właściwą ostrość zarówno obrazu sceny (czajnika i torusa), jak i jej odbicia w lustrze. W samej procedurze `SetupMirrorVPMatrices` jest potrzebna tylko jedna drobna zmiana (listing 25.10). Dotąd była tam zakodowana na stałe wartość parametru `far` procedury `M4x4Frustumf`, ale teraz, w związku z możliwością zmiany długości jednostki układu świata (czyli zmiany wyrażonej w tych jednostkach odległości od sceny obserwatora, także odbitego w lustrze) trzeba dostosowywać ten parametr. Dlatego procedura ma dodatkowy parametr `far` typu `float`, któremu należy nadawać wartość zmiennej `camera->far`.

Listing 25.10. Zmieniona procedura `SetupMirrorVPMatrices`

C

```
1: void SetupMirrorVPMatrices ( GLfloat eyepos[4], GLfloat reyepos[4],
2:                              GLfloat mvm[16], GLfloat mpm[16], float far )
3: {
4:   .... /* początek bez zmian, zobacz listing 20.6 */
5:   M4x4Frustumf ( mpm, NULL, a[0], b[0], a[1], b[1], -a[2], far );
6: } /*SetupMirrorVPMatrices*/
```

Procedura na listingu 25.11 wywołuje jedną z dwóch procedur rysowania sceny w oknie. Obie procedury wykonują obrazy z głębią ostrości zależną od nastawień parametrów kamery. Procedura `DrawBlurredSceneToWindow` dodatkowo dokonuje rozmycia obiektów w ruchu.

Listing 25.11. Procedura `RedrawMyWorld`

```C
void RedrawMyWorld ( void )
{
  glEnable ( GL_DEPTH_TEST );
  if ( appdata.blur && !appdata.hold )
    DrawBlurredSceneToWindow ( &appdata );
  else
    DrawSceneToWindow ( &appdata );
  glFlush ();
} /*RedrawMyWorld*/
```

Procedura na listingu 25.12 wytwarza efekt głębi ostrości, ale nie wytwarza rozmycia obiektów w ruchu. Drugi i trzeci parametr procedury `ArticulateMyLinkage` to odpowiednio czas symulacji ruchu obiektów i przyrost czasu od poprzedniego wywołania. Ponieważ rysowana scena jest statyczna, artykulacja łańcucha kinematycznego (w tym całkowanie ruchu cząsteczek) jest wykonywana tylko raz, bezpośrednio po odczycie zegara w linii 7. Reprezentację obszaru cienia też wystarczy znaleźć tylko raz, wykonując instrukcję w linii 10.

Listing 25.12. Procedura `DrawSceneToWindow`

```C
1:  void DrawSceneToWindow ( AppData *ad )
2:  {
3:    int    i;
4:    double dt;
5:
6:    if ( !ad->hold ) {
7:      dt = TimerTocTic ();
8:      ArticulateMyLinkage ( ad->linkage, app_time-ad->hold_time, dt );
9:    }
10:   DrawSceneToShadows ( ad );
11:   AdjustDimensions ( &ad->camera, &ad->trans );
12:   AccumBufClear ( ad->acb );
13:   for ( i = 0; i < ad->acc_samples; i++ ) {
14:     SetTransformations ( &ad->camera, &ad->trans, ad->acc_samples, i );
15:     DrawSceneToMirror ( ad );
16:     glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, ad->acb->fbo[1] );
17:     glViewport ( 0, 0, ad->camera.win_width, ad->camera.win_height );
18:     LoadVPMatrix ( &ad->trans, false );
19:     glClearColor ( 1.0, 1.0, 1.0, 1.0 );
20:     glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
21:     LoadMMatrix ( &ad->trans, ad->mirror.mirror_matrix );
```

```
22:     DrawMirror ( &ad->mirror, ad->miprog[1], true );
23:     DrawScene ( ad, true );
24:     AccumBufAdd ( ad->acb );
25:   }
26:   AccumBufMult ( ad->acb, 1.0/(float)ad->acc_samples );
27:   AccumBufGamma ( ad->acb );
28:   glBindFramebuffer ( GL_READ_FRAMEBUFFER, ad->acb->fbo[0] );
29:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
30:   glBlitFramebuffer ( 0, 0, ad->camera.win_width, ad->camera.win_height,
31:                       0, 0, ad->camera.win_width, ad->camera.win_height,
32:                       GL_COLOR_BUFFER_BIT, GL_NEAREST );
33:   ExitIfGLError ( "DrawSceneToWindow" );
34: } /*DrawSceneToWindow*/
```

W linii 12 bufor akumulacji jest kasowany, a następnie w pętli są tworzone i zbierane w tym buforze kolejne obrazy. Instrukcja w linii 14 konstruuje macierze dla i-tego obrazu (tj. obrazu sceny widzianej z i-tego przesuniętego położenia obserwatora). W poprzedniej wersji procedury `DrawSceneToWindow` (listing 22.16, linia 51) jest wywołanie procedury `glBindFramebuffer` z drugim parametrem 0; ma to na celu włączenie rysowania w oknie[11]. W nowej wersji parametr procedury wywołanej w linii 16 ma wartość `acb->fbo[1]`, określającą pomocniczy bufor ramki utworzony jednocześnie z buforem akumulacji. Instrukcja w linii 18 przypisuje zmiennym jednolitym w bloku `TransBlock` dane odpowiadające i-temu przesuniętemu położeniu obserwatora, a potem kolejne instrukcje w liniach 19–23 rysują wszystkie elementy sceny. W linii 24 utworzony obraz jest dodawany do bufora akumulacji. W linii 26 zawartość bufora akumulacji jest mnożona przez czynnik $\frac{1}{k}$, dzięki czemu powstaje średnia arytmetyczna wszystkich zebranych obrazów[12]. Kolejna instrukcja powoduje wykonanie korekcji gamma obrazu w buforze akumulacji. Instrukcje w liniach 28–32 przesyłają gotowy obraz z bufora akumulacji do bufora ramki związanego z oknem.

Uwaga: Przeniesienie korekcji gamma do procedur działających na buforze akumulacji oznacza, że szadery fragmentów, które dostarczają wyniki trafiające do tego bufora, mają *nie* wykonywać tej korekcji, a zatem w szaderach napisanych dla wcześniejszych wersji aplikacji należy dokonać odpowiednich przeróbek. Ponieważ są one bardzo proste, nie zamieszczam przedstawiających je listingów.

Listing 25.13 przedstawia procedury, które realizują rozmycie obiektów w ruchu. Poszczególne obrazy, z których zmieszania powstaje obraz końcowy, są wykonywane dla różnych (takich jak poprzednio) przesunięć obserwatora i rzutni, a dodatkowo są „poprzesuwane w czasie".

W celu rozmycia obiektów w ruchu chwile „fotografowania" sceny powinny być w przedziale $[t_a, t_b]$, gdzie t_b jest bieżącym czasem animacji, przechowywanym w zmiennej glo-

[11] Procedury znajdowania obszaru cienia i tekstury na lustrze wykonują obrazy w buforach ramki poza oknem, zatem trzeba po powrocie z nich ustawić bufor ramki dla właściwego obrazu.

[12] Zamiast mnożyć sumę obrazów w tym miejscu, można w pętli używać procedury `AccumBufMultAdd`, co umożliwia wprowadzenie indywidualnych „wag" poszczególnych obrazów.

balnej app_time, a t_a jest czasem wykonania poprzedniego obrazu (czyli poprzedniej klatki animacji). Mając wykonać k „fotografii" w tym przedziale czasowym, wybrałem chwile równoodległe; czas animacji dla i-tego obrazu jest obliczany w linii 29.

Listing 25.13. Procedura DrawBlurredSceneToWindow

C

```
1: void ArticulateMyLinkage ( kl_linkage *lk, double at, double dt )
2: {
3:   AppData *ad;
4:   double  par[9];
5:
6:   ad = (AppData*)lk->usrdata;
7:   if ( ad->animate ) {
8:     if ( (ad->teapot_rot_angle += ANGULAR_VELOCITY1 * dt) >= PI )
9:       ad->teapot_rot_angle -= 2.0*PI;
10:  }
11:  par[0] = ad->teapot_rot_angle;
12:  par[1] = TeapotRotAngle2 ( at );
13:  .... /* instrukcje bez zmian */
14:  ad->part_time = at;
15:  kl_SetArtParam ( lk, 0, 9, par );
16:  kl_Articulate ( lk );
17: } /*ArticulateMyLinkage*/
18:
19: void DrawBlurredSceneToWindow ( AppData *ad )
20: {
21:   int    i;
22:   double at, dt, tt;
23:
24:   at = app_time - ad->hold_time;
25:   dt = TimerTocTic ();
26:   AdjustDimensions ( &ad->camera, &ad->trans );
27:   AccumBufClear ( ad->acb );
28:   for ( i = 0; i < ad->acc_samples; i++ ) {
29:     tt = at + (double)(i+1)*dt/(double)acc_samples;
30:     ArticulateMyLinkage ( ad->linkage, tt, dt/(double)ad->acc_samples );
31:     DrawSceneToShadows ( ad );
32:     SetTransformations ( &ad->camera, &ad->trans,
33:                          ad->acc_samples, ad->blurpermut[i] );
34:     DrawSceneToMirror ( ad );
35:     .... /* tu instrukcje takie jak w liniach 13-20 na listingu 25.12 */
36:     AccumBufAdd ( ad->acb );
37:   }
38:   .... /* dalsze instrukcje takie jak w liniach 23-29 na listingu 25.12 */
39:   ExitIfGLError ( "DrawBlurredSceneToWindow" );
40: } /*DrawBlurredSceneToWindow*/
```

Uwaga: Jeśli celem jest stworzenie filmu animowanego, to możemy chcieć dokładniej symulować działanie kamery filmowej. Migawka kamery („prawdziwej", z taśmą światłoczułą) jest dyskiem, z którego został wycięty sektor o rozwartości od 90° do 120°. Przesuwanie taśmy filmowej w celu naświetlenia następnej klatki filmu odbywa się w czasie, gdy obracająca się migawka zasłania całe okno (o wymiarach klatki), za którym jest taśma. Zatem czas ekspozycji klatki w kamerze zajmuje od 1/4 do 1/3 okresu T między naświetlaniem kolejnych klatek. W programie graficznym chwile „fotografowania" sceny należałoby wybrać w przedziale $[t_a + cT, t_b]$, gdzie $T = t_b - t_a$, $c \in [2/3, 3/4]$. Ten model migawki nie jest bardzo dokładny, ale umożliwia otrzymanie m.in. widocznego w wielu filmach efektu stroboskopowego.

Procedura `ArticulateMyLinkage` w linii 14 zapamiętuje „czas symulacyjny" w polu `part_time` struktury danych aplikacji. Procedura `KLPostProcessParticles` ma ten czas podać jako ostatni parametr w wywołaniu procedury `MoveParticles`, zamiast wartości zmiennej `app_time`.

W procedurze `DrawBlurredSceneToWindow` instrukcja w linii 24 zapamiętuje wynik ostatniego pomiaru czasu, a w linii 25 dokonuje nowego pomiaru czasu i przypisuje zmiennej `dt` przyrost czasu od poprzedniego pomiaru. Wywołania procedur artykulacji łańcucha kinematycznego i znajdowania obszaru cienia zostały przeniesione do pętli, w której za każdym razem jest dokonywana artykulacja w kolejnej chwili, a potem jest wykonywany obraz. Zauważmy, że kolejność wykonywania obrazów musi tu być zgodna z chronologią, bo symulacja układu cząsteczek odbywa się w jedną stronę (każdy krok czasowy całkowania ruchu cząsteczek musi być dodatni). Konstruując przekształcenia dla rzutowania sceny, trzeba poddać punkty określone wzorem (25.3) pewnej permutacji, ponieważ ich oryginalna kolejność daje ciąg przesunięć obserwatora na coraz większe odległości od położenia nominalnego, czego efekt raczej nie wygląda naturalnie. Lepsze wyniki daje „wymieszanie" tych punktów; zobaczmy konstrukcję odpowiedniej permutacji.

Procedura `InitBlurPermut` (listing 25.14) dla danej liczby k przemieszczeń obserwatora generuje odpowiednią permutację ciągu $0, 1, \ldots, k-1$, przy użyciu opisanej wcześniej funkcji Inversion. Procedura wpisuje do tablicy `blurpermut` ciąg $0, 1, \ldots, k-1$, jednocześnie wpisując do pomocniczej tablicy `inv` wartości funkcji Inversion z drugim argumentem (podstawą układu pozycyjnego) równym 3. Pomocnicza tablica jest następnie sortowana (algorytmem QuickSort). Nie widząc potrzeby drukowania całej procedury `QuickSort` (dopisanej do pliku `utilities.c`), na listingu zamieściłem tylko jej nagłówek. Jej parametry to: wskaźnik `data` do *dowolnych* danych, długość `n` sortowanego ciągu i dwie procedury, które (w sposób znany tylko aplikacji) porównują i przestawiają elementy ciągu. Procedura `QuickSort`, wywołując te procedury w miarę potrzeb, przekazuje im wskaźnik `data` jako pierwszy parametr. W tym fragmencie programu parametr `data` jest tablicą pomocniczą z dwoma wskaźnikami; pierwszy z nich wskazuje tablicę pomocniczą `inv`, a drugi tablicę `blurpermut`; procedura `blmLess` porównuje elementy pierwszej tablicy na wskazanych miejscach, a procedura `blmSwap` przestawia elementy w obu tablicach. Z końcem działania procedury `InitBlurPermut` tablica pomocnicza zostaje zapomniana, ale permutacja w tablicy `blurpermut` pozostaje.

Listing 25.14. Tworzenie permutacji w celu rozmycia obiektów w ruchu

C

```
void QuickSort ( void *data, int n,
                 char (*Less)(void *data, int i, int j),
                 void (*Swap)(void *data, int i, int j) );

static char blmLess ( void *data, int i, int j )
{
  return ((float*)((void**)data)[0])[i] < ((float*)((void**)data)[0])[j];
} /*blmLess*/

static void blmSwap ( void *data, int i, int j )
{
  float *inv, x;
  int   *blurpermut, y;

  if ( i == j ) return;
  inv = (float*)((void**)data)[0];
  blurpermut = (int*)((void**)data[1];
  x = inv[i];          inv[i] = inv[j];                  inv[j] = x;
  y = blurpermut[i];  blurpermut[i] = blurpermut[j];  blurpermut[j] = y;
} /*blmSwap*/

void InitBlurPermut ( AppData *ad, int k )
{
  float inv[MAX_ACC_SAMPLES];
  void  *data[2];
  int   i;

  ad->acc_samples = k;
  for ( i = 0; i < (acc_samples = k); i++ )
    ad->blurpermut[i] = i, inv[i] = RadicalInversion ( i, 3 );
  data[0] = (void*)inv;  data[1] = (void*)ad->blurpermut;
  QuickSort ( (void*)data, k, blmLess, blmSwap );
} /*InitBlurPermut*/
```

Listing 25.15 przedstawia procedurę `InitMyWorld`. Dodane do niej instrukcje nadają wartości początkowe parametrom kamery, kompilują szader obsługujący bufor akumulacji, tworzą obiekt bufora, inicjalizują zmienne potrzebne do animacji czajnika i układu cząsteczek i generują odpowiednią permutację w tablicy `blurpermut` dla początkowej liczby obrazów zbieranych w buforze akumulacji.

Listing 25.15. Procedura `InitMyWorld`

C

```
void InitMyWorld ( int argc, char *argv[], int width, int height )
{
  GLfloat ident_matrix[16];
```

```
4:
5:   memset ( &appdata, 0, sizeof(AppData) );
6:   LoadAccumBufShaders ();
7:   acb = NewAccumBuf ( ad->camera.win_width, ad->camera.win_height );
8:   LoadBPShaders ( &ad->brprog );
9:   /* .... dalsze instrukcje bez zmian */
10:  appdata.cnet = appdata.skeleton = appdata.particles = appdata.blur =
11:    appdata.animate = appdata.hold = false;
12:  ....
13:  InitLights ( &appdata );
14:  InitBlurPermut ( &appdata, MIN_ACC_SAMPLES );
15:  if ( !ConstructMyLinkage ( &appdata ) )
16:    ExitOnError ( "InitMyWorld" );
17:  ArticulateMyLinkage ( appdata.linkage, app_time, 0.0 );
18: } /*InitMyWorld*/
```

Procedury na listingu 25.16 służą do zmieniania nastawień odległości, długości ogniskowej obiektywu i przysłony. Stałe wyznaczające zakresy tych nastawień są podane na listingu 25.6. Parametr każdej procedury określa, czy parametr kamery ma być zwiększony, czy zmniejszony. Długość ogniskowa jest zmieniana na polecenie użytkownika, a także po zmianach długości jednostki układu świata i po zmianach wysokości okna. Czynnik $\sqrt{2}$ w procedurach nastawiania przysłony odpowiada standardowej podziałce na obiektywach aparatów fotograficznych, 2–2.8–4–5.6–8–11–16–22–32, która jest w przybliżeniu ciągiem geometrycznym o ilorazie $\sqrt{2}$.

Listing 25.16. Procedury zmieniania ustawień obiektywu

C

```
1: char ChangeFocus ( char incfd )
2: {
3:   int    e;
4:   double f;
5:
6:   e = appdata.camera.distcnt;
7:   if ( incfd ) {
8:     if ( (f = SCREEN_DIST*pow ( FOCUS_FCT, (double)(--e) )) < MIN_FOV_DIST )
9:       return false;
10:  }
11:  else {
12:    if ( (f = SCREEN_DIST*pow ( FOCUS_FCT, (double)(++e) )) > MAX_FOV_DIST )
13:      return false;
14:  }
15:  appdata.camera.distcnt = e;  appdata.camera.dist = f;
16:  return true;
17: } /*ChangeFocus*/
18:
19: char ChangeZoom ( char inczoom )
```

```
20: {
21:   int    e;
22:   double z;
23:
24:   e = appdata.camera.heightcnt;
25:   if ( inczoom ) {
26:     if ( (z = INITIAL_FRAMEHEIGHT*pow ( ZOOM_FCT, (double)(--e) )) <
27:          MIN_FRAME_HEIGHT )
28:       return false;
29:   }
30:   else {
31:     if ( (z = INITIAL_FRAMEHEIGHT*pow ( ZOOM_FCT, (double)(++e) )) >
32:          MAX_FRAME_HEIGHT )
33:       return false;
34:   }
35:   appdata.camera.heightcnt = e;  appdata.camera.frameheight = z;
36:   return true;
37: } /*ChangeZoom*/
38:
39: char ChangeAperture ( char incapr )
40: {
41:   int    e;
42:   double a;
43:
44:   e = appdata.camera.Ncnt;
45:   if ( incapr ) {
46:     if ( (a = INITIAL_APERTURE*pow ( 2.0, 0.5*(double)(++e) )) >
47:          MAX_APERTURE )
48:       return false;
49:   }
50:   else {
51:     if ( (a = INITIAL_APERTURE*pow ( 2.0, 0.5*(double)(--e) )) <
52:          MIN_APERTURE )
53:       return false;
54:   }
55:   appdata.camera.Ncnt = e;  appdata.camera.N = a;
56:   return true;
57: } /*ChangeAperture*/
```

Zmiany parametru kamery polegające na jego pomnożeniu lub podzieleniu przez pewien czynnik stały, podczas wielokrotnego zwiększania i zmniejszania go powodują kumulację błędów zaokrągleń — po zmianie parametru dokładne przywrócenie jego poprzedniej wartości może być niewykonalne. W praktyce te błędy są niezauważalne, ale to nie jest powód, by tego nie naprawić, zwłaszcza, że to jest łatwe; zawsze wartość parametru jest równa wartości początkowej pomnożonej przez czynnik stały podniesiony do pewnej potęgi o całkowitym wykładniku. Aby zmienić parametr, trzeba zwiększyć lub zmniejszyć wykładnik (początkowo równy 0), co odbywa się bez błędów zaokrągleń, a potem użyć funkcji pow; wtedy ob-

liczona wartość parametru nie zależy od błędu wcześniejszej wartości. Właśnie w ten sposób działają pokazane tu procedury. Wartość powrotna `false` oznacza, że postulowana zmiana spowodowałaby wyjście poza dopuszczalny zakres, więc nie została dokonana.

Listing 25.17 przedstawia zmiany w procedurze `ProcessCharCommand`. Interpretacja nowych znaków polega na wywołaniu procedury `ChangeFocus`, która zwiększa lub zmniejsza nastawienie odległości oraz zwiększenie lub zmniejszenie liczby obrazów mieszanych w buforze akumulacji. Napisanie litery `Z` powoduje włączenie lub wyłączenie rozmycia obiektów w ruchu.

Listing 25.17. Procedura obsługi poleceń z klawiatury

C

```
char ProcessCharCommand ( char charcode )
{
  switch ( toupper ( charcode ) ) {
    ....
case '>':
    return ChangeFocus ( true );
case '<':
    return ChangeFocus ( false );
case '^':
    if ( appdata.acc_samples >= MAX_ACC_SAMPLES )
      return false;
    InitBlurPermut ( &appdata, appdata.acc_samples+1 );
    return true;
case 'V':
    if ( appdata.acc_samples <= MIN_ACC_SAMPLES )
      return false;
    InitBlurPermut ( &appdata, appdata.acc_samples-1 );
    return true;
case 'Z':
    appdata.blur = !appdata.blur;
    return appdata.animate;
    ....
  }
} /*ProcessCharCommand*/
```

Procedury `ChangeZoom` i `ChangeAperture` są wywoływane przez procedurę `KeyFunc` z części okienkowej aplikacji, po naciśnięciu jednego z klawiszy strzałek. Procedury zmiany parametrów kamery są też wywoływane przez procedurę obsługi komunikatów rolki, co ułatwia użytkownikowi obsługę aplikacji (listing 25.18).

Listing 25.18. Procedury obsługi klawiszy specjalnych i komunikatów rolki

C

```
void KeyFunc ( GLFWwindow *win, int key, int code, int action, int mode )
{
  if ( action == GLFW_PRESS || action == GLFW_REPEAT ) {
```

```
4:     switch ( key ) {
5:   case GLFW_KEY_ESCAPE:
6:       glfwSetWindowShouldClose ( mywindow, 1 );
7:       break;
8:   case GLFW_KEY_RIGHT:
9:       redraw |= ChangeZoom ( true );
10:      break;
11:  case GLFW_KEY_LEFT:
12:      redraw |= ChangeZoom ( false );
13:      break;
14:  case GLFW_KEY_DOWN:
15:      redraw |= ChangeAperture ( true );
16:      break;
17:  case GLFW_KEY_UP:
18:      redraw |= ChangeAperture ( false );
19:      break;
20:  default:
21:      break;
22:    }
23:  }
24: } /*KeyFunc*/
25:
26: void ScrollFunc ( GLFWwindow *win, double x, double y )
27: {
28:  if ( y == 0.0 )
29:    return;
30:  if ( glfwGetMouseButton ( win, GLFW_MOUSE_BUTTON_LEFT ) == GLFW_PRESS )
31:    redraw |= ChangeFocus ( y < 0.0 );
32:  else if ( glfwGetMouseButton ( win, GLFW_MOUSE_BUTTON_RIGHT ) ==
33:            GLFW_PRESS )
34:    redraw |= ChangeAperture ( y < 0.0 );
35:  else
36:    redraw |= ChangeZoom ( y < 0.0 );
37: } /*ScrollFunc*/
```

Zmiany procedury inicjalizacji części okienkowej są pokazane na listingu 25.19. Wywołanie procedury `glfwWindowHint`, mające na celu utworzenie wielopróbkowego bufora obrazu dla okna aplikacji, można usunąć, ponieważ w tej aplikacji jego rolę przejął bufor akumulacji. W linii 15 jest rejestrowana dla okna procedura obsługi komunikatów o obracaniu rolki myszy z listingu 25.18.

Rysunek 25.6 przedstawia okno aplikacji 2J z różnymi ustawieniami parametrów kamery. We wszystkich przypadkach długość ogniskowa jest równa 268 mm, obserwator jest oddalony od początku układu współrzędnych świata o 800 mm, a jednostka układu świata ma 35.2 mm. Wszystkie obrazy powstały ze zmieszania $k = 12$ obrazów w buforze akumulacji.

Dwa obrazki na górze różnią się nastawieniem ostrości; po lewej stronie ostrość jest nastawiona na $d_o = 800$ mm, a po prawej $d_o = 882$ mm, przy czym w obu przypadkach nastawienie przysłony $N = 8$. Obrazy w środkowym wierszu powstały z wyłączonym i z włączonym

Rysunek 25.6. Okno aplikacji drugiej J

rozmyciem obiektów w ruchu. Obrazy na dole zostały wykonane z nastawieniem przysłony $N = 22$, co zwiększa zakres głębi ostrości, dzięki czemu można na obrazach wyraźniej zobaczyć rozmycie w ruchu dziobka czajnika i (znacznie słabsze) torusa.

Listing 25.19. Procedura `Initialise`

C

```
void Initialise ( int argc, char **argv )
{
  glfwSetErrorCallback ( myGLFWErrorHandler );
  if ( !glfwInit () )
    ExitOnError ( "glfwInit failed" );
/*  glfwWindowHint ( GLFW_SAMPLES, 8 );*/
  if ( !(mywindow = glfwCreateWindow ( 480, 360,
                                       "aplikacja 2J", NULL, NULL )) ) {
    glfwTerminate ();
    ExitOnError ( "glfwCreateWindow failed" );
  }
  glfwMakeContextCurrent ( mywindow );
  glGetProcAddresses ( APP2J_GL_MAJOR, APP2J_GL_MINOR );
  .... /* tu instrukcje bez zmian */
  glfwSetScrollCallback ( mywindow, ScrollFunc );
  InitMyWorld ( argc, argv, 480, 360 );
  redraw = true;
} /*Initialise*/
```

Jeśli rysowana scena jest skomplikowana, a maksymalna średnica plamki będącej obrazem punktu obiektu należącego do tej sceny jest duża (bo obraz jest duży, otwór przysłony jest duży i zakres odległości obiektów w scenie jest duży), lub jeśli obiekty poruszają się bardzo szybko, to dla dobrej jakości klatek animacji może być potrzebna duża liczba k obrazów mieszanych w buforze akumulacji. Taka animacja z klatkami tworzonymi sposobem opisanym w tym rozdziale może nie być wykonalna w czasie rzeczywistym przy użyciu nawet bardzo potężnej GPU.

25.5. Ćwiczenia

1. Zmodyfikuj aplikację tak, aby można było jawnie ustalać długość ogniskową i dostosowywać do niej długość w milimetrach jednostki układu świata.
2. Zmienianie położenia obserwatora jest również animacją, która powinna się wiązać z rozmywaniem przedmiotów w ruchu. Zaimplementuj algorytm, który to robi.
3. Zmodyfikuj aplikację tak, aby liczba obrazów zbieranych w buforze akumulacji była automatycznie dostosowywana do wymiarów okna oraz nastawienia przysłony, długości ogniskowej i odległości — dla oszczędności czasu.

26

Aplikacja druga K

Za każdym razem, gdy aplikacja 2J ma w oknie wyświetlić nowy obraz, rysuje kolejne rzuty sceny i sumuje obrazy w buforze akumulacji. Czas rysowania jest sumą czasów zużytych na wykonanie k kompletnych obrazów sceny, różniących się, jeśli nie ma rozmycia obiektów w ruchu, tylko położeniami obserwatora i związanymi z nimi macierzami potrzebnymi do rzutowania.

Aby oszczędzić czas, wszystkie te obrazy można wykonywać jednocześnie. W tym celu program szaderów powinien sprawić, że dane przekazane do etapu obcinania (i dalszych) w potoku przetwarzania grafiki trafią na różne **warstwy** (*layers*) obrazu końcowego. Wyboru warstwy może dokonać szader geometrii, a więc *wszystkie* programy szaderów, które rysują obrazy wielowarstwowe, *muszą* zawierać odpowiednie szadery geometrii. Wcześniejsze etapy potoku (w tym szadery wierzchołków i rozdrabniania) będą mogły wykonać swoją pracę tylko raz, ale w tym celu położenia wierzchołków przekazane szaderowi geometrii muszą być podane w tym samym układzie współrzędnych, najlepiej w układzie świata. Szader geometrii powinien mieć tablicę (lub tablice) z macierzami przejścia do układów obserwatora i kostki standardowej dla wszystkich warstw. Wtedy może on wyprodukować wyniki dla konkretnej warstwy i wprowadzić je do dalszych etapów potoku z podaniem numeru tej warstwy.

26.1. Rysowanie na wielu warstwach

Procedury OpenGL-a i elementy języka GLSL potrzebne do uruchomienia rysowania na wielu warstwach opiszę, przedstawiając je na przykładzie zmian aplikacji 2J (których wynikiem jest aplikacja 2K), ale pewne opisy pominę, aby uniknąć łopatologii. Pierwsza konieczna zmiana to (niestety) usunięcie kodu służącego do rozmycia obiektów w ruchu.

Na listingach 26.1 i 26.2 są pokazane zmiany, jakie należy wprowadzić do szaderów rozdrabniania i geometrii, aby płaty Béziera (czajnik i torus) rysować na wielu warstwach; opis tworzenia tych warstw i korzystania z nich będzie dalej. W treści wszystkich szaderów odwołujących się do bloku zmiennych jednolitych `TransBlock` konieczne jest zastąpienie w tym bloku pojedynczej macierzy `vpm` tablicą, której długość jest liczbą warstw. Każdy element tej

tablicy zawiera macierz przejścia od układu świata do układu kostki standardowej dla kolejnego przesuniętego położenia obserwatora, tj. iloczyn macierzy przejścia od układu świata do układu przesuniętego obserwatora i macierzy przejścia od tego układu do układu kostki standardowej skonstruowanej na podstawie parametrów odpowiedniego ostrosłupa widzenia. Ponadto wektor `eyepos`, który zawiera współrzędne położenia obserwatora, jest zastąpiony przez tablicę.

Dotychczas przekształcaniem wierzchołków do układu kostki standardowej zajmował się szader rozdrabniania z listingu 15.4 i 22.3. Obecnie ma on wyprowadzić (w zmiennej `gl_Position`) wektor współrzędnych jednorodnych wierzchołka w układzie świata, zatem w linii 26 wektor współrzędnych w układzie modelu (obliczony przez jedną z procedur znajdowania punktu płata) jest mnożony tylko przez macierz przejścia do układu świata.

Listing 26.1. Szader rozdrabniania płatów Béziera do rysowania na wielu warstwach

GLSL

```
1: #version 450 core
2:
3: #define NLAYERS 16
4:
5: uniform TransBlock {
6:     mat4 mm, mmti;
7:     mat4 vpm[NLAYERS];
8:     vec4 eyepos[NLAYERS];
9:   } trb;
10:
11: .... /* pozostałe zmienne i podprogramy prawie bez zmian */
12:
13: void main ( void )
14: {
15:   vec4 pos, wpos, pu, pv, nv;
16:   uint l, mask;
17:
18:   Out.instance = inst = In[0].instance;
19:   pos = nv = pu = pv = vec4 ( 0.0 );
20:   switch ( bezp.dim ) {
21: case 2: BPHorner2f ( .... );  break;
22: case 3: BPHorner3f ( .... );  break;
23: case 4: BPHorner4f ( .... );  break;
24:   }
25:   Out.PatchCoord = gl_TessCoord.xy;
26:   wpos = trb.mm * pos;
27:   for ( l = 0, mask = 0x00000001;  l < light.nls;  l++, mask <<= 1 )
28:     if ( (light.mask & mask) != 0 )
29:       Out.ShadowPos[l] = light.ls[l].shadow_vpm * wpos;
30:   gl_Position = wpos;
31:   .... /* wszystkie pozostałe instrukcje bez zmian */
32: } /*main*/
```

Instrukcje obliczające wektor normalny pozostały prawie bez zmian, ponieważ ten szader (i dalej szader geometrii) zawsze miał go podawać (i podawał) w układzie świata.

Listing 26.2. Szader geometrii do rysowania na wielu warstwach

GLSL

```
#version 450

#define NLAYERS     16
#define MAX_NLIGHTS  8

layout(invocations=NLAYERS,triangles) in;
layout(triangle_strip,max_vertices=3) out;

in GVertex { .... } In[];
out FVertex { .... } Out;
struct LSPar { .... };
uniform LSBlock { .... } light;
uniform BezPatch { .... } bezp;

uniform TransBlock {
    mat4 mm, mmti;
    mat4 vpm[NLAYERS];
    vec4 eyepos[NLAYERS];
  } trb;

void main ( void )
{
  uint i, l, mask;
  vec3 v1, v2, nv;

  v1 = In[1].Position - In[0].Position;
  v2 = In[2].Position - In[0].Position;
  nv = normalize ( cross ( v1, v2 ) );
  gl_Layer = gl_InvocationID;
  for ( i = 0; i < 3; i++ ) {
    gl_Position = trb.vpm[gl_InvocationID] * gl_in[i].gl_Position;
    Out.Position = In[i].Position;
    .... /* dalsze instrukcje bez zmian */
    EmitVertex ();
  }
  EndPrimitive ();
} /*main*/
```

Potrzebne zmiany szadera geometrii są również niewielkie. Szader, jak poprzednio, przyjmuje podany na wejście wektor normalny powierzchni lub oblicza wektor normalny płaszczyzny przetwarzanego trójkąta w układzie świata i przekazuje go na wyjście. Z kolei podany na wejście wektor współrzędnych jednorodnych wierzchołka jest teraz w układzie świata.

Szader geometrii mnoży ten wektor przez macierz z tablicy vpm z bloku TransBlock, przy czym indeks do tej tablicy, czyli numer warstwy, na której ma powstać obraz, jest wartością zmiennej wbudowanej gl_InvocationID. Kwalifikator invocations=NLAYERS w linii 6 określa liczbę k wątków szadera dla przetwarzanego (otrzymanego od szadera rozdrabniania) trójkąta. Wyjście każdego wątku trafi na warstwę o numerze przypisanym w linii 29 zmiennej wbudowanej gl_Layer (zatem numer wątku, od 0 do $k-1$, jest numerem docelowej warstwy).

Uwaga: Kwalifikator wejścia szadera geometrii z podaną liczbą wątków można zastąpić przez pętlę postaci

```
for ( int l = 0; l < n; l++ ) {
  gl_Layer = l;
  ....
  EndPrimitive ();
}
```

Maksymalna liczba wierzchołków podana w kwalifikatorze wyjścia musi być równa iloczynowi maksymalnej liczby warstw i liczby wierzchołków wyprowadzanych na jedną warstwę (np. NLAYERS*3, jeśli szader wyprowadza tylko jeden trójkąt). Obliczenia dla poszczególnych warstw zostaną wtedy przeprowadzone sekwencyjnie, ale za to (jeśli n jest zmienną jednolitą) jest możliwe zmienianie liczby warstw w kolejnych operacjach rysowania bez potrzeby ponownego kompilowania szadera.

Szader fragmentów ma dostęp do zmiennej wbudowanej gl_Layer, której wartość została nadana przez szader geometrii[1] i która jest numerem warstwy zawierającej piksel, któremu po przejściu testu widoczności zostanie przypisany kolor fragmentu. Dzięki temu szader fragmentów jest w stanie sięgnąć do odpowiedniego elementu tablicy eyepos w bloku TransBlock, aby otrzymać właściwe położenie obserwatora, które trzeba podstawić do modelu oświetlenia. W szaderach fragmentów używanych do rysowania płatów Béziera przez dotychczas opisane wersje drugiej aplikacji wystarczy w wywołaniu procedury posDifference zastąpić parametr trb.eyepos przez trb.eyepos[gl_Layer].

Listingi 26.3 i 26.4 przedstawiają szadery wchodzące w skład programu rysowania lustra z nałożoną teksturą opisującą obraz odbitej sceny. Szader wierzchołków tego programu jest przedstawiony na listingu 20.1. Problem do rozwiązania w tym miejscu polega na tym, że lustro narysowane na każdej warstwie ma mieć inną teksturę, a to dlatego, że przesunięcie obserwatora wiąże się z przesunięciem położenia obserwatora odbitego w lustrze. Dlatego konstruując bufor ramki do rysowania sceny odbitej w lustrze, utworzymy tekstury wielowarstwowe, które staną się załącznikami (buforem obrazu i buforem głębokości) tego bufora ramki. Gotowe obrazy w warstwach tekstury — bufora obrazu — trzeba nałożyć na obrazy lustra w warstwach końcowego obrazu. Szader wierzchołków dostarcza dla każdego

[1] Jeśli szader geometrii tego nie zrobił lub jeśli program szaderów nie zawiera szadera geometrii, to wyjście trafia na warstwę o numerze 0, ale wartość zmiennej gl_Layer szadera fragmentów jest nieokreślona.

wierzchołka wektor współrzędnych tekstury o dwóch współrzędnych, czyli typu vec2. Szader geometrii, wstawiony między szadery wierzchołków i fragmentów, przekazuje na wyjście wektor współrzędnych tekstury typu vec3; jego pierwsze dwie współrzędne są skopiowane ze zmiennej wejściowej, a trzecia współrzędna jest numerem warstwy. Wyjściowy wektor współrzędnych tekstury wybiera instrukcja w linii 18.

Listing 26.3. Szader geometrii do rysowania lustra na wielu warstwach

GLSL

```
1: #version 450 core
2:
3: #define NLAYERS 16
4:
5: layout(invocations=NLAYERS,triangles) in;
6: layout(triangle_strip,max_vertices=3) out;
7:
8: in GVertex { vec2 TexCoord; } In[];
9: out FVertex { vec3 TexCoord; } Out;
10: uniform TransBlock { .... } trb;   /* blok taki sam, jak na listingu 26.2 */
11:
12: void main ( void )
13: {
14:   int i;
15:
16:   for ( i = 0; i < 3; i++ ) {
17:     gl_Layer = gl_InvocationID;
18:     Out.TexCoord = vec3 ( In[i].TexCoord, float(gl_Layer) );
19:     gl_Position = trb.vpm[gl_InvocationID] * gl_in[i].gl_Position;
20:     EmitVertex ();
21:   }
22:   EndPrimitive ();
23: } /*main*/
```

Listing 26.4. Szader fragmentów do rysowania lustra na wielu warstwach

GLSL

```
1: #version 450 core
2:
3: in FVertex { vec3 TexCoord; } In;
4: out vec4 out_Colour;
5:
6: layout(binding=1) uniform sampler2DArray tex;
7:
8: void main ( void )
9: {
10:   out_Colour = gl_FrontFacing ? vec4(0.3) : texture ( tex, In.TexCoord );
11: } /*main*/
```

Dostęp do tekstury, która ma być nałożona na lustro, szader fragmentów uzyskuje za pośrednictwem zmiennej jednolitej `tex` typu `sampler2DArray`, która umożliwia dostęp do tekstury wielowarstwowej, będącej w istocie tablicą tekstur dwuwymiarowych. W linii 10 szader wywołuje funkcję `texture`. Jej nazwa jest przeciążona, tzn. jest wiele funkcji o tej nazwie, różniących się typami parametrów i wyniku (zobacz p. 9.13.8). Jeśli pierwszy parametr funkcji `texture` jest typu `sampler2DArray`, to funkcja oblicza wartość tekstury wielowarstwowej. Pierwsze dwie współrzędne drugiego parametru (typu `vec3`) określają punkt w dziedzinie tekstury (tj. w kwadracie jednostkowym), a trzecia współrzędna (typu `float`, ale będąca liczbą całkowitą) wybiera odpowiednią warstwę.

Zajmijmy się teraz kodem aplikacji na CPU. Aby obraz lustra na każdej warstwie miał nałożoną właściwą teksturę, wystarczy przed użyciem szaderów z listingów 26.3 i 26.4 teksturę wielowarstwową otrzymaną w wyniku działania procedury `DrawSceneToMirror` przywiązać do celu `GL_TEXTURE_2D_ARRAY`. Listing 26.5 przedstawia procedurę `DrawMirror` wywoływaną przez opisaną dalej procedurę `DrawSceneToWindow`. Jedyna zmiana w porównaniu z procedurą na listingu 22.15 polega na podaniu parametru `GL_TEXTURE_2D_ARRAY` w linii 8, aby na obrazy lustra na poszczególnych warstwach nałożyć odpowiednie warstwy tekstury.

Listing 26.5. Procedura `DrawMirror`

C

```
1: void DrawMirror ( AppData *ad, char final )
2: {
3:   SetModelMatrix ( ad, mirror_matrix );
4:   glBindVertexArray ( mirror_vao );
5:   if ( final ) {
6:     glUseProgram ( ad->mrprog.progid[1] );
7:     glActiveTexture ( GL_TEXTURE1 );
8:     glBindTexture ( GL_TEXTURE_2D_ARRAY, mirror_txt[0] );
9:   }
10:  else
11:    glUseProgram ( ad->mrprog.progid[0] );
12:  glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
13:  glDrawArrays ( GL_TRIANGLE_FAN, 0, 4 );
14:  glBindVertexArray ( 0 );
15:  ExitIfGLError ( "DrawMirror" );
16: } /*DrawMirror*/
```

Pola `vwm1`, `vpm1`, `mvm` `mpm`, `eyepos` i `reyepos` struktury `TransBl`, przechowujące dotychczas pojedyncze macierze i wektory, trzeba zmienić w tablice. W zasadzie niepotrzebne są osobne macierze przejścia od układu świata do obserwatora i od układu obserwatora do kostki standardowej, a więc można przechowywać tylko ich iloczyny dla każdego przesuniętego położenia obserwatora (listing 26.6). Zatem, pola `wvpm1` i `eyepos1` są tablicami przechowującymi macierze przejścia od układu świata do układu kostki standardowej dla poprzesuwanych położeń obserwatora, a w tablicach `mvpm` i `reyepos` są analogiczne dane dla położeń odbitych w lustrze.

Listing 26.6. Zmieniony typ struktury `TransBL`

```C
typedef struct TransBl {
    GLfloat mm[16], mmti[16], wvm0[16],
            wvpm1[NLAYERS][16], mvpm[NLAYERS][16];
    GLfloat eyepos0[4], eyepos1[NLAYERS][4], reyepos[NLAYERS][4];
  } TransBl;
```

Po utworzeniu tekstur, które mają stać się załącznikami pomocniczego bufora ramki, procedura `AllocAccTextures` (listing 25.4, linie 18–21) powinna je przywiązać do celu `GL_TEXTURE_2D_ARRAY` (zamiast `GL_TEXTURE_2D`). Wywołania procedury `glTexStorage2D` w liniach 19 i 21 należy zastąpić wywołaniami procedury `glTexStorage3D`, która ma o jeden parametr więcej. Ten dodatkowy parametr określa trzeci wymiar tworzonej tekstury (trójwymiarowej), przy czym użycie celu `GL_TEXTURE_2D_ARRAY` oznacza, że ma być utworzona (jednowymiarowa) tablica tekstur dwuwymiarowych o długości określonej przez ostatni parametr procedury `glTexStorage3D`. W naszym przypadku powinien on być równy liczbie warstw obrazu, czyli 16 zgodnie z makrodefinicją `NLAYERS` w treści szaderów[2]. Zwracam uwagę, że tekstura używana jako bufor głębokości musi mieć tyle samo warstw co tekstura będąca buforem obrazu.

Podobnie w konstrukcji bufora ramki używanego do rysowania obrazów sceny odbitej w lustrze należy utworzyć i podać jako załączniki (tj. bufor obrazu i bufor głębokości) tekstury wielowarstwowe. Listing 26.7 przedstawia zmienioną procedurę `ConstructMirrorFBO`, którą proszę porównać z procedurą zamieszczoną na listingu 20.5.

Listing 26.7. Zmieniona procedura `ConstructMirror`

```C
void ConstructMirror ( void )
{
  GLenum status;

  glGenTextures ( 2, mirror_txt );
  glActiveTexture ( GL_TEXTURE1 );
  glBindTexture ( GL_TEXTURE_2D_ARRAY, mirror_txt[0] );
  glTexParameteri ( GL_TEXTURE_2D_ARRAY GL_TEXTURE_MIN_FILTER, GL_LINEAR );
  glTexParameteri ( GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  glTexStorage3D ( GL_TEXTURE_2D_ARRAY, 1, GL_RGBA32F,
                   MIRRORTXT_W, MIRRORTXT_H, NLAYERS );
  glBindTexture ( GL_TEXTURE_2D_ARRAY, mirror_txt[1] );
  glTexStorage3D ( GL_TEXTURE_2D_ARRAY, 1, GL_DEPTH_COMPONENT32F,
                   MIRRORTXT_W, MIRRORTXT_H, NLAYERS );
  glGenFramebuffers ( 1, &mirror_fbo );
  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, mirror_fbo );
  glFramebufferTexture ( GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                         mirror_txt[0], 0 );
  glFramebufferTexture ( GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
```

[2] Jeśli chcemy to zmienić, to pamiętajmy, aby *wszędzie* było jednakowo.

```
                          mirror_txt[1], 0 );
  if ( (status = glCheckFramebufferStatus ( GL_DRAW_FRAMEBUFFER )) !=
       GL_FRAMEBUFFER_COMPLETE )
    ExitOnError ( "ConstructMirror" );
  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
  glBindTexture ( GL_TEXTURE_2D_ARRAY, 0 );
  ExitIfGLError ( "ConstructMirror" );
} /*ConstructMirror*/
```

Na listingu 26.8 są przedstawione procedury rysowania sceny na teksturach wielowarstwowych. Procedura `LoadVPMatrices`, która zastąpiła wcześniej używaną procedurę `LoadVPMatrix`, przypisuje wartości początkowym *k* elementom tablic `vpm` i `eyepos` w bloku zmiennych jednolitych `TransBlock`.

Procedury `DrawSceneToMirror` i `DrawSceneToWindow` przed wydaniem GPU poleceń rysowania wywołują procedurę `LoadVPMatrices`, podając jej jako parametry tablice macierzy przekształceń i poprzesuwanych położeń obserwatora odbitych (`mvpm` i `reyepos`) albo nieodbitych (`wvpm` i `eyepos`) w lustrze.

Listing 26.8. Procedury rysowania sceny

C

```
void LoadVPMatrices ( TransBl *trans, char mirror )
{
  GLfloat *vpm, *ep;

  if ( mirror )
    { vpm = trans->mvpm[0];  ep = trans->reyepos[0]; }
  else
    { vpm = trans->wvpm1[0];  ep = trans->eyepos1[0]; }
  glBindBuffer ( GL_UNIFORM_BUFFER, trans->trbuf );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[2],
                    NLAYERS*16*sizeof(GLfloat), vpm );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[3],
                    NLAYERS*4*sizeof(GLfloat), ep );
  ExitIfGLError ( "LoadVPMatrices" );
} /*LoadVPMatrices*/

void DrawSceneToMirror ( AppData *ad )
{
  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, ad->mirror_fbo );
  glViewport ( 0, 0, MIRRORTXT_W, MIRRORTXT_H );
  LoadVPMatrices ( &trans, true );
  glClearColor ( 0.95, 0.95, 0.95, 1.0 );
  glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  DrawScene ( ad, true );
  glFlush ();
} /*DrawSceneToMirror*/
```

```
27:
28: void DrawSceneToWindow ( AppData *ad )
29: {
30:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, ad->acb->fbo[1] );
31:   glViewport ( 0, 0, ad->camera.win_width, ad->camera.win_height );
32:   LoadVPMatrices ( &ad->trans, false );
33:   glClearColor ( 1.0, 1.0, 1.0, 1.0 );
34:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
35:   LoadMMatrix ( &ad->trans, ad->mirror.mirror_matrix );
36:   DrawMirror ( &ad->mirror, ad->miprog[1], true );
37:   DrawScene ( ad, true );
38:   AccumBufAverage ( ad->acb, ad->stereo );
39:   glBindFramebuffer ( GL_READ_FRAMEBUFFER, ad->acb->fbo[0] );
40:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
41:   glBlitFramebuffer ( 0, 0, ad->camera.win_width, ad->camera.win_height,
42:                       0, 0, ad->camera.win_width, ad->camera.win_height,
43:                       GL_COLOR_BUFFER_BIT, GL_NEAREST );
44:   glFlush ();
45:   ExitIfGLError ( "DrawSceneToWindow" );
46: } /*DrawSceneToWindow*/
47:
48: void RedrawMyWorld ( void )
49: {
50:   int i;
51:
52:   ArticulateMyLinkage ( appdata.linkage );
53:   glEnable ( GL_DEPTH_TEST );
54:   DrawSceneToShadows ( &appdata );
55:   AdjustDimensions ( &appdata.camera, &appdata.trans );
56:   for ( i = 0; i < NLAYERS; i++ ) {
57:     SetTransformations ( &appdata.camera, &appdata.trans,
58:                          appdata.stereo, i );
59:     SetupMirrorVPMatrices ( appdata.trans.eyepos1[i],
60:               appdata.trans.reyepos[i], appdata.trans.mvpm[i],
61:               appdata.camera.far );
62:   }
63:   DrawSceneToMirror ( &appdata );
64:   DrawSceneToWindow ( &appdata );
65: } /*RedrawMyWorld*/
```

Procedura `DrawSceneToShadows` nie wymaga żadnych zmian; nadal wykonuje ona obraz jednowarstwowy, ponieważ scena jest statyczna, a zatem nieruchome są zarówno obiekty tej sceny, jak i źródła światła. Oczywiście, program szaderów używany do znalezienia reprezentacji obszaru cienia (szader na listingu 22.2) trzeba zmodyfikować, dostosowując go do nowej postaci bloku zmiennych jednolitych `TransBlock`. Wystarczy w tym celu dopisać indeks `[0]` do instrukcji sięgających do zmiennej `trb.vpm`, która stała się tablicą, i przed uruchomieniem obliczeń znajdujących reprezentację cienia przypisać pierwszemu elemen-

towi tej tablicy współczynniki odpowiedniej macierzy; to zadanie wykonuje (niewymagająca zmian) procedura `LoadShTrans` wywoływana przez procedurę `BindShadowFBO`.

Po nadaniu odpowiednich wartości zmiennym jednolitym (w szczególności elementom tablic w bloku `TransBlock`) podczas tworzenia obrazu zarówno odbitego w lustrze, jak i końcowego trzeba narysować scenę tylko raz, dlatego pętla w procedurze `DrawSceneToWindow` służy tylko do obliczenia potrzebnych macierzy. Po otrzymaniu warstw końcowego obrazu trzeba uśrednić obrazy z tych warstw i dokonać korekcji gamma. Szader obliczeniowy, który to robi, jest opisany w następnym podrozdziale (listing 26.10). Procedury przygotowujące do pracy i uruchamiające program z tym szaderem są tak proste, że nie zamieszczam w książce ich treści.

Opisane wyżej (oraz pominięte, aby Czytelnik miał co przećwiczyć) modyfikacje dają elegancką implementację algorytmu głębi ostrości, ale nie umożliwiają rozmywania obiektów w ruchu. Gdyby scena składała się tylko z brył sztywnych poruszających się względem siebie i obserwatora, to można byłoby osiągnąć także ten efekt za pomocą tablicy macierzy przekształceń modelu (każda macierz w tablicy odpowiadałaby innej chwili, każdy obiekt miałby swoją tablicę). Jednak scena rysowana przez naszą aplikację zawiera czajnik z odkształcanym dziobkiem i układ cząsteczek, których rozmycie w ruchu wymagałoby utworzenia reprezentacji tych obiektów we wszystkich chwilach, których fotografie mają być zmieszane na końcowym obrazie. To znaczy, że *wszystkie* etapy potoku przetwarzania grafiki (a nie tylko szader geometrii i etapy części tylnej potoku) muszą wykonać *całą pracę* dla poszczególnych chwil, także związaną ze znajdowaniem zmieniających się w czasie obszarów cienia. Rysowanie na wielu warstwach nie dałoby w tym przypadku oszczędności czasu, za to ogromnie by wzrosło zapotrzebowanie aplikacji na pamięć GPU. Dlatego do rozmycia obiektów w ruchu znacznie lepiej i prościej jest użyć bufora akumulacji zrealizowanego w poprzednim rozdziale w stylu starego OpenGL-a.

26.2. Stereoskopia

Obrazy powstające w lewym i prawym oku osoby oglądającej trójwymiarową scenę są nieco inne, co umożliwia **widzenie stereoskopowe**, czyli odtwarzanie przez tę osobę informacji o odległości obiektów na podstawie różnic między tymi obrazami. Stereoskopia w grafice komputerowej wymaga rozwiązania dwóch problemów: utworzenia odpowiedniej pary obrazów i spowodowania, aby każde oko obserwatora widziało tylko obraz dla tego oka przeznaczony.

Są dwa sposoby rozwiązania drugiego problemu: drogi i tani. **Sposób drogi** wymaga zaopatrzenia się w stosowny sprzęt, na który składają się okulary z migawkami z ciekłych kryształów i nadajnik przesyłający w podczerwieni sygnał sterujący tymi migawkami[3]. Podczas pracy w trybie stereo monitor wyświetla obrazy z podwójną częstotliwością (np. 120 Hz zamiast 60 Hz), przy czym co drugi obraz jest przeznaczony dla oka lewego, a drugi co drugi dla oka prawego; sygnał sprawia, że migawki na przemian są nieprzezroczyste. Aplikacja,

[3] I być może specjalny monitor. Nadajnik może być wbudowany w monitor lub stać obok niego, istnieją też okulary podłączane przewodem do odpowiedniego wyjścia karty graficznej.

tworząc kontekst OpenGL-a dla obrazów stereo, musi go odpowiednio skonfigurować, a potem wyświetlać w oknie odpowiednie pary obrazów.

Od razu przyznam się, że nie dysponuję takim sprzętem, zatem podane niżej informacje na temat drogiego sposobu nie zostały przeze mnie praktycznie sprawdzone. Aplikacja FreeGLUT-a, w celu użycia trybu stereo, powinna podczas inicjalizacji wykonać instrukcję

```
glutInitDisplayMode ( GLUT_RGBA | .... | GLUT_STEREO );
```

Aplikacja biblioteki GLFW powinna przed utworzeniem okna, które ma pracować w tym trybie, wykonać instrukcję

```
glfwWindowHint ( GLFW_STEREO, GL_TRUE );
```

Z kolei aplikacja systemu X Window (korzystająca z biblioteki GLX) podczas tworzenia kontekstu OpenGL-a powinna wywołać procedurę `glXChooseFBConfig` z parametrem — tablicą atrybutów bufora ramki — zawierającą parę liczb całkowitych (`GLX_STEREO, True`). Jeśli jednak komputer, na którym została uruchomiona aplikacja, nie jest wyposażony we wspomniany sprzęt, to próba utworzenia stereoskopowego kontekstu OpenGL-a (przez aplikację korzystającą z dowolnej z tych bibliotek) zakończy się niepowodzeniem i tyle[4].

Bufor ramki związany z oknem działającym w trybie stereo ma cztery bufory obrazu, które są jego załącznikami: lewy przedni, lewy tylny, prawy przedni i prawy tylny[5]. Zawartość buforów przednich jest wyświetlana na ekranie, podczas gdy w tylnych buforach odbywa się rysowanie. Identyfikatory tych buforów mają nazwy symboliczne `GL_FRONT_LEFT`, `GL_BACK_LEFT`, `GL_FRONT_RIGHT` i `GL_BACK_RIGHT`. Do określania buforów obrazu (będących załącznikami bieżąco aktywnego bufora ramki), w których ma powstawać obraz, służy procedura `glDrawBuffers`. Jej drugi parametr jest tablicą identyfikatorów buforów obrazu, a pierwszy parametr jest długością tej tablicy. Można wykonać obrazy po kolei, za pomocą instrukcji

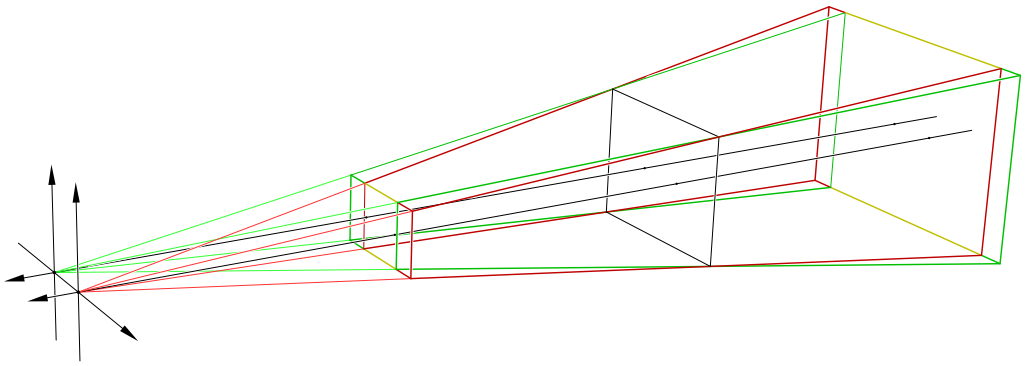
```
GLenum b;
b = GL_BACK_LEFT;  glDrawBuffers ( 1, &b );
....  /* narysuj obraz dla lewego oka */
b = GL_BACK_RIGHT;  glDrawBuffers ( 1, &b );
....  /* narysuj obraz dla prawego oka */
glFlush ();  glfwSwapBuffers ();
```

Można też oba obrazy wykonać jednocześnie. Najlepiej[6] jest użyć do tego rysowania na dwóch warstwach poza ekranem. Po otrzymaniu obrazów w pozaekranowym buforze ramki wystarczy przesłać je za pomocą procedury `glBlitFramebuffer` do buforów obrazu wyświetlanych w oknie. Nic nie stoi na przeszkodzie, aby obrazy dla lewego i prawego oka powstały przez zmieszanie większej liczby obrazów w celu uzyskania głębi ostrości i rozmycia obiektów w ruchu (ale, aby otrzymać efekt rozmycia w ruchu obiektów zmieniających kształt, trzeba osobno wykonać obrazy odpowiadające poszczególnym chwilom).

[4] To akurat sprawdziłem.

[5] Jeśli okno nie pracuje w trybie stereo, to obecne w buforze ramki okna bufory obrazu są formalnie lewe.

[6] moim skromnym zdaniem



Rysunek 26.1. Ostrosłupy widzenia dla stereoskopii

Teraz zajmijmy się **sposobem tanim**. Bez specjalnego sprzętu się nie obejdzie, ale mogą nim być względnie łatwe do zrobienia samemu[7] okulary z kolorowymi filtrami przepuszczającymi tylko światło czerwone i tylko zielone (na te dwa z trzech „czystych" kolorów światła emitowanego przez monitor ludzkie oczy są najbardziej czułe). Pary nakładających się obrazów wykonanych w dwóch kolorach i przeznaczonych do oglądania przez kolorowe filtry są znane pod nazwą **anaglify**. Podobnie, jak w drogim sposobie, trzeba wykonać oba obrazy, w dwóch pozaekranowych buforach ramki lub w osobnych warstwach tekstury będącej załącznikiem bufora ramki. Następnie trzeba je zmieszać w buforze akumulacji i wyświetlić na ekranie. Każdy obraz z pary musi być zamieniony na obraz jednobarwny, w kolorze światła przepuszczanego przez jeden filtr z pary i tłumionego przez drugi. Zamianą obrazów kolorowych na jednobarwne i ich końcowym przetwarzaniem zajmiemy się dalej.

Macierze potrzebne do rzutowania sceny dla lewego i prawego oka mogą być skonstruowane przez procedurę `M4x4SkewFrustumf` z listingu 25.1. Aby móc jej skutecznie użyć, trzeba wyrazić w jednostkach długości układu świata cztery wymiary fizyczne (które można zmierzyć linijką)[8]. Są to wymiary (szerokość i wysokość) okna aplikacji, rozstaw źrenic oczu osoby oglądającej obraz i ich odległość od ekranu. Zakłada się, że widz znajduje się dokładnie na wprost okna, choć to założenie nieczęsto jest spełnione. Ale okazuje się, że nawet dość spore przemieszczenia obserwatora względem „idealnego" położenia, a także zmiana wymiarów obrazu, nie wpływają istotnie na zjawisko stereoskopii. Tę samą parę obrazów stereo można oglądać na ekranie niewielkiego laptopa i wyświetlaną przez rzutnik na dużym ekranie i w obu przypadkach tak samo napawać się wrażeniem trójwymiarowości sceny. Komputer wykonuje jednak obliczenia dla konkretnych wymiarów, które trzeba podać.

Przyjąłem, że rozstaw źrenic obserwatora jest równy 67 mm. Mając ustaloną długość jednostki układu świata, trzeba obliczyć rozstaw źrenic w tych jednostkach (oznaczmy go symbolem m). Wywołując procedurę `M4x4SkewFrustumf` dla oka lewego, trzeba obliczony

[7] Być może najtrudniejsze jest tu kupienie przezroczystej kolorowej folii w ilości niehurtowej.

[8] Sposób obliczania parametrów ostrosłupa widzenia wprowadzony w aplikacji 2J był w istocie spowodowany moim zamiarem zaimplementowania (później, czyli teraz) stereoskopii i to dlatego w obliczeniach występuje długość jednostki układu świata i odległość obserwatora od ekranu podana w milimetrach.

opisanym wcześniej sposobem parametr xv zmniejszyć o $m/2$, a dla oka prawego o tyle samo zwiększyć[9]. Procedura M4x4SkewFrustumf dostosuje do tego właściwe przesunięcia przedniej (i tylnej) ściany ostrosłupa widzenia.

Listing 26.9 przedstawia procedurę SetTransformations dostosowaną do użycia w aplikacji, która (korzystając z rysowania wielowarstwowego) tworzy obrazy z głębią ostrości i może rysować stereoskopowe pary obrazów. Gdy stereoskopia jest włączona, obrazy z pierwszych k = NLAYERS/2 warstw, po zmieszaniu, utworzą końcowy obraz dla lewego oka, a obrazy z pozostałych warstw złożą się na obraz dla oka prawego. Liczba $m/2$, czyli połowa rozstawu źrenic obserwatora, wyrażona w jednostkach układu świata, powinna być obliczona przez odpowiednią instrukcję dodaną do procedury AdjustDimensions (listing 25.7) i zapamiętana w dodatkowym polu eyeshift zmiennej strukturalnej camera. Jeśli tryb stereo jest włączony, to procedura SetTransformations nadaje zmiennej es wartość przesunięcia oka obserwatora, a wartością zmiennej j staje się numer przemieszczenia potrzebnego do otrzymania głębi ostrości — dla oka lewego i prawego odpowiednio w liniach 11 i 12.

Listing 26.9. Nowa procedura SetTransformations

C

```
1:  void SetTransformations ( Camera *camera, TransBl *trans,
2:                            char stereo, int i )
3:  {
4:    float   F, dist, rvmax, ri, tau, phii, xvi, yvi;
5:    float   l, r, b, t, es;
6:    GLfloat wvm[16], wpm[16];
7:    int     j, k;
8:
9:    if ( stereo ) {
10:     k = NLAYERS/2;
11:     if ( i < k ) { es = -camera->eyeshift;  j = i; }
12:     else         { es = camera->eyeshift;   j = i-k; }
13:   }
14:   else { k = NLAYERS;  j = i;  es = 0.0; }
15:   F = camera->F/camera->unitmm;
16:   dist = camera->dist/camera->unitmm;
17:   rvmax = F*(dist-F)/(2.0*camera->N*dist);
18:   ri = rvmax * sqrt ( (float)(j+j+1)/(float)(k+k) );
19:   tau = 0.5*(sqrt ( 5.0 ) - 1.0);
20:   phii = (float)(j+j+1)*PI*tau*tau;
21:   xvi = ri*cos ( phii );
22:   yvi = ri*sin ( phii );
23:   M4x4SkewFrustumf ( camera->win_width, camera->win_height, 1.0, camera->F,
24:                      dist, xvi + es, yvi, ((float)j+0.5)/k-0.5,
25:                      RadicalInversion ( j, 2 ) + 0.5*(1.0-1.0/(float)k),
```

[9] Jeśli nie wprowadzamy głębi ostrości, to parametr xv dla lewego oka ma być równy $-m/2$, a dla prawego $+m/2$. Dla obrazów z głębią ostrości połowę rozstawu źrenic odejmujemy lub dodajemy do przesunięć obliczonych tak jak w procedurze z listingu 25.9.

```
                    camera->near, camera->far, &l, &r, &b, &t, trans->wvm0,
                    wvm, trans->eyepos1[i], wpm, NULL );
  M4x4Multf ( trans->wvpm1[i], wpm, wvm );
} /*SetTransformations*/
```

Do włączania i wyłączania stereoskopii może służyć klawisz F1 oraz instrukcja

```
case GLFW_KEY_F1:  stereo = !stereo;  break;
```

dopisana do instrukcji przełącznika w procedurze KeyFunc.

Zamiana obrazu kolorowego na jednobarwny polega na zmieszaniu składowych r, g, b pikseli w proporcjach odpowiadających względnej czułości receptorów w oku na światło czerwone, zielone i niebieskie, za pomocą wzoru

$$l = c_r r + c_g g + c_b b.$$

Symbol l oznacza poziom szarości, który daje wrażenie takiej samej luminancji (zobacz podrozdz. 28.1 i dodatek C) jak światło barwne o składowych r, g, b. Współczynniki c_r, c_g i c_b najlepiej jest przyjąć na podstawie jednego ze standardów telewizyjnych (zobacz podrozdz. C.4), na przykład $c_r = 0.2126$, $c_g = 0.7152$, $c_b = 0.0722$ lub $c_r = 0.299$, $c_g = 0.587$, $c_b = 0.114$. Chcąc zamienić obraz kolorowy na obraz w skali szarości reprezentowany jak kolorowy, należy dla każdego piksela obliczyć liczbę l i przypisać ją wszystkim trzem składowym r, g, b odpowiedniego piksela obrazu wynikowego. Aby otrzymać obraz anaglifowy, należy obliczoną wartość l piksela jednego obrazu (np. lewego) przypisać składowej g, a dla drugiego obrazu (prawego) składowej r. „Nieużywana" składowa b piksela w obrazie wynikowym powinna mieć wartość 0.

Listing 26.10 przedstawia szader obliczeniowy, którego zadaniem jest uśrednienie obrazów w buforze akumulacji. Jeśli zmienna jednolita action ma wartość ACCBUF_AVERAGE, to obraz wynikowy jest średnią arytmetyczną obrazów ze wszystkich warstw. Alternatywą jest obliczenie dwóch średnich, które są zamieniane na obrazy jednobarwne, z których powstaje obraz anaglifowy. W obu przypadkach szader dokonuje na końcu korekcji gamma.

Listing 26.10. Szader obsługi bufora akumulacji dla obrazu wielowarstwowego

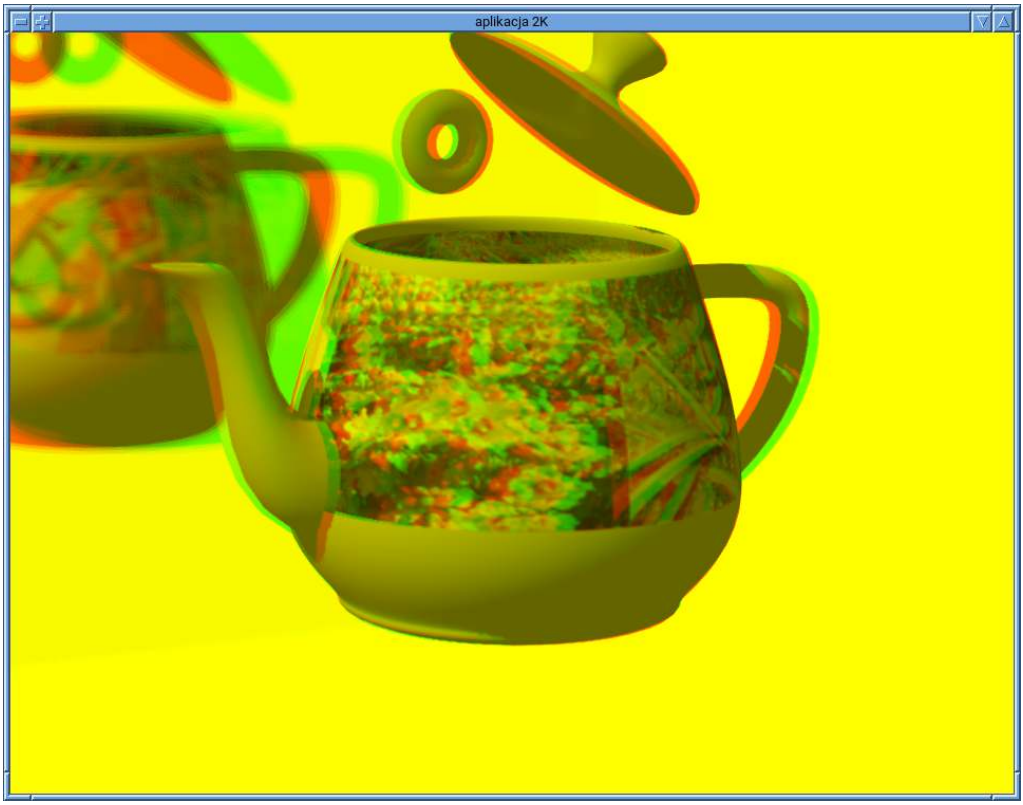
GLSL

```
#version 450 core

#define NLAYERS        16
#define ACCBUF_AVERAGE  0
#define ACCBUF_ANAGLYPH 1

layout(local_size_x=1) in;
layout(rgba32f,binding=0) uniform image2D accb;
layout(rgba32f,binding=1) uniform image2DArray img;
layout(location=0) uniform int action;

const vec3 c = vec3(0.299,0.587,0.114);
```

```
13:
14: #define AGamma(VEC,colour) pow ( colour, VEC(256.0/563.0) )
15:
16: void main ( void )
17: {
18:   ivec2 xy;
19:   vec3  pix, rpix;
20:   float lp, rp;
21:
22:   xy = ivec2 ( gl_GlobalInvocationID.xy );
23:   switch ( action ) {
24: case ACCBUF_AVERAGE:
25:     pix = vec3 ( 0.0 );
26:     for ( int i = 0; i < NLAYERS; i++ )
27:       pix += imageLoad ( img, ivec3 ( xy, i ) ).rgb;
28:     imageStore ( accb, xy,
29:                  vec4 ( AGamma ( vec3, pix/float(NLAYERS) ), 1.0 ) );
30:     break;
31: case ACCBUF_ANAGLYPH:
32:     pix = rpix = vec3 ( 0.0 );
33:     for ( int i = 0; i < NLAYERS/2; i++ ) {
34:       pix += imageLoad ( img, ivec3 ( xy, i ) ).rgb;
35:       rpix += imageLoad ( img, ivec3 ( xy, i + NLAYERS/2 ) ).rgb;
36:     }
37:     lp = dot ( pix, c )/(0.5*float(NLAYERS));
38:     rp = dot ( rpix, c )/(0.5*float(NLAYERS));
39:     imageStore ( accb, xy,
40:                  vec4 ( AGamma ( vec2, vec2 ( rp, lp ) ), 0.0, 1.0 ) );
41:     break;
42: default:
43:     return;
44:   }
45: } /*main*/
```

26.3. Ćwiczenia

1. Dodaj możliwość zmieniania liczby obrazów zbieranych w buforze akumulacji. Liczba wywołań szaderów geometrii ma być równa liczbie warstw tekstury, w której powstaje obraz (czyli maksymalnej liczbie obrazów), natomiast liczba faktycznie wykonywanych obrazów powinna być sterowana za pomocą zmiennej jednolitej. Jeśli zmienna `gl_InvocationID` ma wartość większą lub równą wartości tej zmiennej jednolitej, to szader geometrii powinien natychmiast zakończyć działanie (bez wyprowadzania wyników).
2. Dodaj do aplikacji 2J możliwość wyświetlania obrazów stereoskopowych (np. anaglifów). Umożliwi to otrzymanie rozmycia obiektów w ruchu na takich obrazach.



Rysunek 26.2. Okno aplikacji drugiej K z obrazem anaglifowym

3. Napisz program, który tworzy reprezentacje obszarów cienia jednocześnie dla wielu źródeł światła, korzystając z rysowania na wielu warstwach. Obszary cienia powinny być wyznaczane tylko dla źródeł światła włączonych w danej chwili.
4. Zmodyfikuj aplikację tak, aby dostosować gęstość mgły (sterowaną przez składową alfa koloru cząsteczki) do nastawienia parametrów kamery; zmieniając je użytkownik powinien mieć wrażenie, że mgła wylatująca z dziobka czajnika pozostaje mniej więcej tak samo przejrzysta.

26.4. Uzupełnienia

26.4.1. Tekstury i obrazy

Podsumujmy różnice między teksturą a obrazem w OpenGL-u. Tekstura jest reprezentacją (skalarnej lub wektorowej) funkcji w postaci tablicy lub zbioru tablic tekseli na jednym lub wielu poziomach (mipmapingu, zobacz rozdz. 19), wyposażoną w ewaluator tekstury, którego zadaniem jest obliczanie wartości tej funkcji, w tym interpolacja danych przecho-

wywanych w tekselach[10]. Dziedzina tekstury jest przedziałem $[0,1]$, kwadratem $[0,1]^2$ lub kostką jednostkową $[0,1]^3$,[11] przy czym sposób obliczania tekstury w punktach spoza dziedziny zależy od ustawionych przez aplikację parametrów ewaluatora. Obraz jest tylko jedno-, dwu- lub trójwymiarową tablicą (pikseli albo tekseli), która może wchodzić w skład tekstury, a odczyt lub zapis elementu tablicy wymaga podania odpowiednich indeksów. Szadery mogą (za pośrednictwem ewaluatora) tylko czytać dane z tekstury, natomiast w przypadku obrazu możliwy jest zarówno odczyt, jak i zapis elementów tablicy (ewentualna interpolacja danych odczytanych z obrazu może być wykonana przez szader). Zarówno tekstura, jak i obraz mogą być wielowarstwowe; wielowarstwowa tekstura składa się z wielowarstwowych obrazów.

Ewaluator tekstury należy zadeklarować jako zmienną jednolitą jednego z typów zamkniętych[12] przewidzianych dla tekstur. Dostęp do obrazu odbywa się za pośrednictwem jednostki obrazu; w treści szadera trzeba zadeklarować zmienną jednolitą typu zamkniętego przewidzianego dla obrazów. Nazwy tych typów są słowami kluczowymi zawierającymi rdzeń `sampler` (dla tekstury) albo `image` (dla obrazu). Rdzeń może być poprzedzony przedrostkiem `i` lub `u`; brak przedrostka oznacza, że szader „widzi" wartości tekstury lub pikseli obrazu reprezentowane przez liczby zmiennopozycyjne (funkcja `texture` lub `imageLoad` ma wynik typu `vec4`). Przedrostek `i` oznacza liczby całkowite ze znakiem (funkcja `texture` lub `imageLoad` podaje wynik typu `ivec4`), a przedrostek `u` oznacza wartości typu całkowitego bez znaku (wynik jest typu `uvec4`)[13]. Jest też garść przyrostków, które określają dziedzinę tekstury lub wymiary tablic pikseli obrazu; są one zebrane w tabeli 9.2 (s. 206), a niżej opisuję je trochę dokładniej niż w tabeli:

`1D`, `2D`, `3D` — tekstury i obrazy jedno-, dwu- i trójwymiarowe jednowarstwowe.

`1DArray`, `2DArray` — tekstury i obrazy wielowarstwowe.

`2DMS` — tekstury i obrazy dwuwymiarowe wielopróbkowe (*multisample*), umożliwiające antyaliasing. Jednowymiarowych tekstur wielopróbkowych nie ma, bo są zbędne, a trójwymiarowych nie ma, bo byłyby zbyt kosztowne do realizacji.

`2DMSArray` — wielowarstwowe wielopróbkowe tekstury i obrazy.

`Cube`, `CubeArray` — tekstury i obrazy, których warstwy są związane ze ścianami sześcianu. Dowolny obiekt można „obudować" takim sześcianem i nałożyć na ten obiekt teksturę, rzutując ją ze wszystkich stron. Jeśli powierzchnia obiektu jest (zakrzywionym) lustrem, to można sprawić, że na gotowym obrazie zobaczymy obraz odbitego w tym lustrze otoczenia obiektu, wcześniej narysowanego na ścianach tego sześcianu.

`1DShadow`, `2DShadow`, `1DArrayShadow`, `2DArrayShadow` — tekstury jedno- i dwuwymiarowe przeznaczone do reprezentowania obszarów cienia; wartość funkcji `texture` lub

[10] Szader może odczytywać dane z tablic bez interpolacji, służy do tego funkcja `texelFetch`.

[11] Dwa wyjątki to tekstury prostokątne i opisane dalej tekstury kostkowe.

[12] Zobacz przypis 10 na s. 483.

[13] Typ liczb przechowywanych w tablicy tekseli nie musi być identyczny z typem widocznym dla szadera; jeśli to są liczby całkowite (np. ośmiobitowe, czyli bajty, bez znaku), a ewaluator tekstury jest typu `sampler2D`, to funkcja `texture` dokona konwersji współrzędnych teksela na typ zmiennopozycyjny i dokona tzw. **normalizacji** (w przypadku bajtów bez znaku podzieli każdą liczbę przez 255), aby otrzymać liczby z przedziału $[0,1]$.

`textureProj` jest wynikiem porównania głębokości punktu podanego jako parametr z wartością tekstury, do zastosowania w algorytmie cieni.

`CubeShadow`, `CubeArrayShadow` — tekstury, których warstwy są związane ze ścianami sześcianu. Można je wykorzystać do reprezentowania obszaru cienia wokół źródła światła umieszczonego w scenie, na przykład lampy w pomieszczeniu.

`Buffer` — tekstury lub obrazy jednowymiarowe używane do przechowywania danych w dowolnych obliczeniach, jakie szadery wykonują w tylko sobie (i ich autorowi) znanych celach.

Przyrostki ze słowem `Shadow` występują tylko z rdzeniem `sampler` i bez przedrostka. Na podstawie powyższego opisu Czytelnik może już wyliczyć słowa kluczowe języka GLSL będące nazwami typów zamkniętych przeznaczonych do reprezentowania identyfikatorów ewaluatorów tekstur i obrazów oraz może odgadnąć własności i zastosowanie takich typów jak `isampler2DArray`, `usamplerCube`, `samplerCubeArrayShadow` lub `iimageBuffer`[14].

Zajmijmy się teraz kwalifikatorami zmiennych dających dostęp do obrazów. Ich identyfikatory składają się z trzech części[15]. Pierwsza część to jedna, dwie lub cztery litery, `r`, `rg`, `rgba`, druga to cyfry 8, 16 lub 32, a część ostatnia, jeśli jest obecna, jest literą `f` lub `i` albo literami `ui`. Określają one sposób reprezentowania pikseli; pierwsza część określa liczbę składowych (1, 2 albo 4, dopuszczalne są tu tylko całkowite potęgi 2), druga część to liczba bitów składowej, a trzecia określa reprezentację składowych: `f` — zmiennopozycyjną, `i` — całkowitą ze znakiem, `ui` lub pusta — całkowitą bez znaku. Kwalifikatorów z literą `f` (np. `rgba32f`) należy używać do obrazów z pikselami o składowych zmiennopozycyjnych (których nazwa typu nie ma przedrostka, np. `image2D`), kwalifikatorów z literą `i` do obrazów reprezentowanych przez liczby całkowite ze znakiem (typów z przedrostkiem `i`, np. `iimage2D`), a kwalifikatorów z literami `ui` do obrazów zawierających liczby bez znaku (np. `uimage2D`).

Reprezentacja zmiennopozycyjna ma 32 albo 16 bitów; w pierwszym przypadku jest używana standardowa reprezentacja pojedynczej precyzji, a w drugim tzw. **reprezentacja połówkowej precyzji**, w której cecha ma 5 bitów, a mantysa 10. Więcej informacji na ten temat jest w specyfikacji standardu OpenGL [1].

26.4.2. Tekstury kostkowe

Tekstury kostkowe służą do opisywania środowisk otaczających rysowane obiekty. Taka tekstura składa się z dwuwymiarowych obrazów nałożonych na ściany sześciennej kostki. Jeśli wszystkie obiekty sceny mieszczą się wewnątrz kostki, to każda półprosta wychodząca z do-

[14] Są jeszcze przyrostki `2DRect` i `2DRectShadow`. Dawno, dawno temu dwuwymiarowe tablice tekseli musiały być kwadratowe, tj. mieć jednakową liczbę wierszy i kolumn, możliwość zaś przetwarzania prostokątnych tablic tekseli była rozszerzeniem standardu. Obecnie tekstury dwuwymiarowe nie mają takiego ograniczenia. Tekstury dostępne przez ewaluatory z przyrostkami `2DRect` i `2DRectShadow` mają ograniczenia, na przykład nie mogą być wielopróbkowe i nie mogą być wielopoziomowe, tj. używane do mipmapingu. Dziedziną „zwykłych" tekstur dwuwymiarowych jest kwadrat jednostkowy, a dziedziną tekstury prostokątnej o wymiarach $w \times h$ tekseli jest prostokąt $[0, w] \times [0, h]$.

[15] Są jeszcze inne możliwe kwalifikatory, ich dokładny opis jest w specyfikacji [3].

wolnego punktu rysowanego obiektu przecina którąś ścianę kostki; temu punktowi można przyporządkować wartość tekstury w punkcie przecięcia. Trzeba tylko określić sposób wybierania kierunków odpowiednich półprostych.

Najczęściej tekstury kostkowe są używane do reprezentowania krajobrazów i wnętrz pomieszczeń. W pierwszym przypadku górna ściana kostki i części ścian bocznych reprezentują niebo z chmurami (stąd takie tekstury mają angielską nazwę *skybox*), a pozostałe części ścian bocznych i ściana dolna przedstawiają na przykład góry lub zabudowania i grunt lub inne podłoże. Rysując pokryte teksturą ściany kostki, otrzymamy tło dla sceny. Rysując obiekty, możemy otrzymać efekty związane z ich oświetleniem. Nałożenie tekstury kostkowej na obiekt daje efekt odbijania się otoczenia w lustrzanej powierzchni obiektu. Tekstura może też opisywać intensywność światła dochodzącego z różnych kierunków, co umożliwia uogólnienie rozważanego w p. 18.4.3 modelu hemisferycznego oświetlenia powierzchni. Przykłady takich zastosowań tekstur kostkowych są opisane w p. 28.3.2 i 28.4.3.

Tekstury kostkowe mają też zastosowanie w algorytmie cieni, gdy źródło światła (np. lampa lub świeca) znajduje się między obiektami sceny. Wtedy kostką zostaje otoczone źródło światła i reprezentacja obszaru cienia składa się z sześciu części reprezentowanych przez jeden obiekt tekstury (więcej o tym jest w p. 26.4.3).

Cel, do którego należy przywiązywać teksturę kostkową, aby ją utworzyć, a później używać, ma nazwę `GL_TEXTURE_CUBE_MAP`. Poszczególne ściany kostki są równoległe do płaszczyzn układu współrzędnych tekstury, a każda z nich jest identyfikowana przez jedną z sześciu nazw utworzonych przez doczepienie do tej nazwy celu przyrostków `_POSITIVE_X`, `_NEGATIVE_X`, `_POSITIVE_Y`, `_NEGATIVE_Y`, `_POSITIVE_Z`, `_NEGATIVE_Z`[16]. Argumentem tekstury kostkowej jest wektor o trzech współrzędnych (s, t, p), który nie może być wektorem zerowym. Współrzędne tego wektora są jednorodne, a zatem podanie zamiast (s, t, p) wektora (as, at, ap) (dla dowolnego $a > 0$) da w wyniku ten sam punkt na ścianie kostki[17].

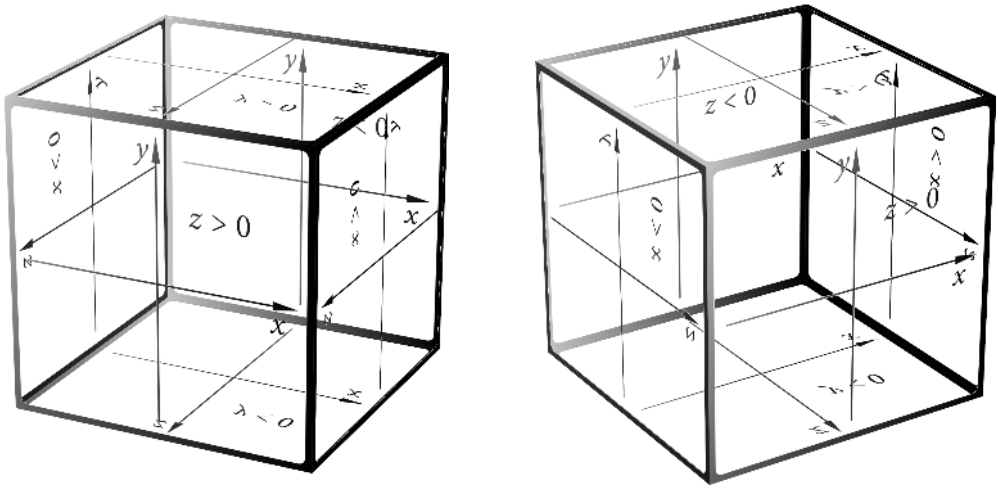
Na rysunku 26.3 jest pokazany sześcian z nałożoną teksturą (rys. 26.4) przedstawiającą odcinki równoległe do osi układu modelu (tożsamego z układem świata) i odpowiednie opisy osi. Aby otrzymać właściwy efekt, obrazy, które mają być teksturami, trzeba przed przesłaniem do pamięci GPU odpowiednio przekształcić (obrócić lub odbić). Określenie tych przekształceń zostawiam jako ćwiczenie.

Zobaczmy przykład użycia tekstury kostkowej; aby go otrzymać, dokonałem przeróbek w aplikacji 2C z rozdziału 18. Do sceny dodałem pomieszczenie, w którym znajduje się czajnik. Pomieszczenie to jest sześcianem o środku w początku układu współrzędnych świata i o krawędziach równoległych do osi tego układu. Krawędzie sześcianu mają długość 24, dzięki czemu mieści się w nim także obserwator, pozostający przez cały czas w odległości 10 od początku układu świata.

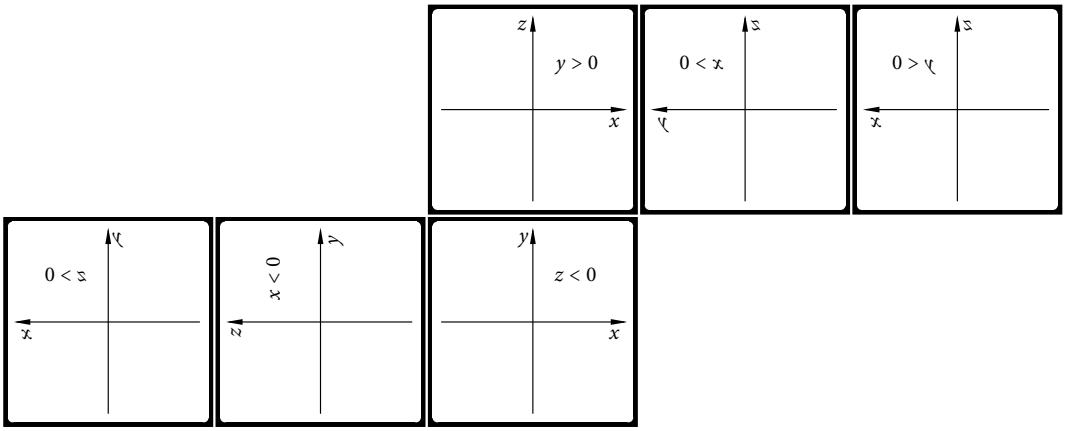
Przyjąłem, że podłoga, sufit i ściany pomieszczenia są białe, przy czym na ścianach znajdują się cztery obrazy — fotografie nakładane jako tekstura na czajnik w aplikacji 2D i dal-

[16] Pod otrzymanymi w ten sposób nazwami symbolicznymi są ukryte kolejne liczby całkowite, tak więc na przykład `GL_TEXTURE_CUBE_MAP_POSITIVE_Y == GL_TEXTURE_CUBE_MAP_POSITIVE_X+2`.

[17] Ewaluator tekstury określa właściwą ścianę kostki na podstawie tego, która współrzędna ma największą wartość bezwzględną i jaki jest znak tej współrzędnej.



Rysunek 26.3. Kostka sześcienna z nałożoną teksturą



Rysunek 26.4. Tekstura nałożona na ściany kostki na rysunku 26.3

szych. Uznałem też, że ściany odbijają światło zgodnie z modelem Lamberta, ale sufit i ściany są przezroczyste dla światła wpadającego do pomieszczenia.

Listing 26.11 przedstawia procedury użyte do utworzenia tekstury kostkowej. Podczas inicjalizacji aplikacja wywołuje procedurę `ConstructMyCubeTexture` (linie 54–78), która najpierw wywołuje procedurę `CreateCubeTexture`. Po przywiązaniu (w linii 8) identyfikatora tekstury do celu nowy obiekt tekstury staje się teksturą kostkową. Potrzebna aplikacji liczba poziomów mipmapingu jest wartością ostatniego parametru. Rezerwacji pamięci na tablice tekseli dokonuje procedura `glTexStorage2D` wywołana w linii 11.

Uwaga: Tablice tekseli dla tekstury kostkowej *muszą* być kwadratowe, tj. mieć tyle samo wierszy co kolumn.

Listing 26.11. Procedury tworzenia tekstury kostkowej

C

```
GLuint tex;

GLuint CreateCubeTexture ( int wh, GLenum format, GLint maxlevel )
{
  GLuint tex;

  glGenTextures ( 1, &tex );
  glBindTexture ( GL_TEXTURE_CUBE_MAP, tex );
  glTexParameteri ( GL_TEXTURE_CUBE_MAP,
                    GL_TEXTURE_MAX_LEVEL, maxlevel );
  glTexStorage2D ( GL_TEXTURE_CUBE_MAP, maxlevel+1, format, wh, wh );
  ExitIfGLError ( "CreateCubeTexture" );
  return tex;
} /*CreateCubeTexture*/

void FlipRotateImage ( int wh, GLubyte *image, int fliprot ) { .... }

char LoadMyTextureImage ( GLuint tex, GLuint target,
                          int txwidth, int txheight, int x, int y,
                          const char *filename, int fliprot )
{
  int     w, h;
  GLubyte *image;

  if ( !(image = ReadTiffImage ( filename, &w, &h )) )
    return false;
  if ( x+w <= txwidth && y+h <= txheight ) {
    glBindTexture ( GL_TEXTURE_CUBE_MAP, tex );
    FlipRotateImage ( w, image, fliprot );
    glTexSubImage2D ( target, 0, x, y, w, h,
                      GL_RGBA, GL_UNSIGNED_BYTE, image );
    free ( image );
    ExitIfGLError ( "LoadMyTextureImage" );
    return true;
  }
  else {
    free ( image );
    return false;
  }
} /*LoadMyTextureImage*/

char SetupMyTextureParameters ( GLuint tex )
{
  glBindTexture ( GL_TEXTURE_CUBE_MAP, tex );
  glGenerateMipmap ( GL_TEXTURE_CUBE_MAP );
```

```
46:   glTexParameteri ( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
47:   glTexParameteri ( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
48:                     GL_LINEAR_MIPMAP_LINEAR );
49:   glEnable ( GL_TEXTURE_CUBE_MAP_SEAMLESS );
50:   ExitIfGLError ( "SetupMyTextureParameters" );
51:   return 1;
52: } /*SetupMyTextureParameters*/
53:
54: void ConstructMyCubeTexture ( void )
55: {
56: #define TXTSIZE 768
57: #define TXTXY   128
58:   int     i;
59:   GLubyte *image;
60:
61:   tex = CreateCubeTexture ( TXTSIZE, GL_RGB8, 7 );
62:   image = malloc ( TXTSIZE*TXTSIZE*4 );
63:   memset ( image, 255, TXTSIZE*TXTSIZE*4 );
64:   glBindTexture ( GL_TEXTURE_CUBE_MAP, tex );
65:   for ( i = 0; i < 6; i++ )
66:     glTexSubImage2D ( GL_TEXTURE_CUBE_MAP_POSITIVE_X+i, 0, 0, 0,
67:                       TXTSIZE, TXTSIZE, GL_RGBA, GL_UNSIGNED_BYTE, image );
68:   free ( image );
69:   LoadMyTextureImage ( tex, GL_TEXTURE_CUBE_MAP_POSITIVE_X,
70:                        TXTSIZE, TXTSIZE, TXTXY, TXTXY, "jaszczur.tif", 1 );
71:   LoadMyTextureImage ( tex, GL_TEXTURE_CUBE_MAP_POSITIVE_Y,
72:                        TXTSIZE, TXTSIZE, TXTXY, TXTXY, "salamandra.tif", 0 );
73:   LoadMyTextureImage ( tex, GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
74:                        TXTSIZE, TXTSIZE, TXTXY, TXTXY, "kwiatki.tif", 3 );
75:   LoadMyTextureImage ( tex, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y,
76:                        TXTSIZE, TXTSIZE, TXTXY, TXTXY, "lis.tif", 2 );
77:   SetupMyTextureParameters ( tex );
78: } /*ConstructMyCubeTexture*/
```

W liniach 62–68 ściany, podłoga i sufit pomieszczenia są malowane na biało, po czym w liniach 69–76 następują wywołania procedury `LoadMyTextureImage`, której zadaniem jest przeczytanie plików z obrazkami i zawieszenie obrazków na ścianach pomieszczenia.

Do procedury `LoadMyImageTexture` z listingu 19.6 trzeba było wprowadzić pewne zmiany. Nowy parametr `target` określa, na którą ścianę kostki ma trafić przeczytany obraz. Nowy parametr `fliprot` określa potrzebny obrót lub odbicie obrazu, aby nie wisiał on na ścianie do góry nogami lub bokiem, albo nie był odbity w lustrze. Kąt obrotu jest iloczynem wartości tego parametru i miary kąta prostego, przy czym jeśli parametr jest większy niż 3, to przed obracaniem trzeba ustawić wiersze pikseli w odwrotnej kolejności, a odpowiednie przekształcenie obrazu (tj. przestawienie pikseli w tablicy) wykonuje procedura `FlipRotateImage`, której treść pominąłem.

Przesłaniem pikseli (które w tym momencie stają się tekselami) do pamięci GPU zajmuje się procedura `glTexSubImage2D`, przy czym pierwszy jej parametr nie jest celem, do którego tekstura jest przywiązana, tylko identyfikatorem ściany kostki, na której obraz ma się znaleźć. Obraz jest przesyłany do ściany tekstury przywiązanej do celu `GL_TEXTURE_CUBE_MAP`.

Ostatnią czynnością przygotowawczą jest wygenerowanie obrazów dla mipmapingu i nadanie wartości parametrom tekstury określającym sposób interpolacji; to wykonuje procedura `SetupMyTextureParameters`. Wywołanie procedury `glEnable` w linii 49 ma na celu włączenie tzw. **bezszwowego łączenia tekstury** na krawędziach kostki, czyli dokonywania interpolacji między tekselami z sąsiednich ścian, gdy zachodzi taka potrzeba.

Kolejne dwa listingi przedstawiają istotne fragmenty szaderów przeznaczonych do rysowania ścian pomieszczenia i lustrzanych płatów Béziera. W treści takich szaderów trzeba zadeklarować zmienną jednolitą typu `samplerCube`. Szader z listingu 26.12 po prostu podaje otrzymane w zmiennej `In.Position` współrzędne kartezjańskie punktu na ścianie rysowanego pomieszczenia jako parametr funkcji `texture`. Wartość tekstury jest dalej używana w modelu oświetlenia Lamberta jako kolor „farby” w tym punkcie.

Listing 26.12. Szader fragmentów do rysowania ścian pomieszczenia

GLSL

```
1:  .... pierwsze 29 linii jest identyczne, jak na listingu 10.4,
2:  .... dlatego tu jest tylko skrót dla przypomnienia
3: in FVertex { .... } In;
4: out vec4 out_Colour;
5: uniform TransBlock { .... } trb;
6: struct LSPar { .... };
7: uniform LSBlock { .... } light;
8: vec3 posDifference ( vec4 p, vec3 pos, out float dist ) { .... }
9: float attFactor ( vec3 att, float dist ) { .... }
10:
11: layout(binding=0) uniform samplerCube CubeMapTxt;
12:
13: vec3 LambertLighting ( void )
14: {
15:   vec4  cmtx;
16:   ....
17:   cmtx = texture ( CubeMapTxt, In.Position  );
18:   normal = normalize ( In.Normal );
19:   .... /* dalej instrukcje takie same jak w procedurze main na
20:       listingu 10.4, ale zamiast koloru ściany podanego w zmiennej
21:       In.Colour należy użyć koloru tekstury zapamiętanego w zmiennej cmtx.*/
22: } /*LambertLighting*/
```

Nieco bardziej skomplikowane obliczenie trzeba wykonać podczas obliczania koloru punktu powierzchni lustrzanej (listing 26.13). Instrukcje dodane do procedury `BlinnPhongLighting` z listingu 18.4 obliczają i dodają do koloru wyjściowego składnik reprezentują-

cy odbite od takiej powierzchni światło dochodzące od ścian kostki[18]. Większą część tego obliczenia wykonują dwie procedury pomocnicze.

Zadaniem procedury `ReflectTxtPoint` (linie 14–27) jest znalezienie odpowiedniego punktu na ścianie pomieszczenia. Rozważmy dwie półproste. Pierwsza z nich, zwana **promieniem pierwotnym**, ma początek w położeniu obserwatora i przechodzi przez punkt na rysowanej powierzchni. Druga półprosta, czyli **promień wtórny**, ma początek w tym punkcie, a jej kierunek wyznacza wektor otrzymany przez odbicie symetryczne wektora kierunkowego promienia pierwotnego w płaszczyźnie stycznej do powierzchni. Punkt, który należy znaleźć, jest punktem przecięcia promienia wtórnego ze ścianą pomieszczenia.

Parametry procedury `ReflectTxtPoint` opisują punkt powierzchni `p`, wektor kierunkowy promienia pierwotnego `vv` i wektor normalny powierzchni `nv` w układzie współrzędnych świata. Funkcja `reflect` w linii 20 oblicza wektor kierunkowy promienia wtórnego, po czym następuje obliczenie punktu przecięcia tego promienia ze ścianą. Wielkość pomieszczenia jest tu zakodowana „na twardo" za pomocą makrodefinicji `CUBESIZE`; ściany, podłoga i sufit leżą w płaszczyznach $x, y, z = \pm 12$, co daje kostkę, której krawędzie mają długość 24.

Listing 26.13. Szader fragmentów do rysowania płatów Béziera

GLSL

```
1:  .... pierwsze 29 linii jest identyczne, jak na listingu 10.4,
2:  .... dlatego tu jest tylko skrót dla przypomnienia
3:
4: in FVertex { .... } In;
5: out vec4 out_Colour;
6: uniform TransBlock { .... } trb;
7: struct LSPar { .... };
8: uniform LSBlock { .... } light;
9: vec3 posDifference ( vec4 p, vec3 pos, out float dist ) { .... }
10: float attFactor ( vec3 att, float dist ) { .... }
11:
12: layout(binding=0) uniform samplerCube CubeMapTxt;
13:
14: vec3 ReflectTxtPoint ( vec3 p, vec3 vv, vec3 nv )
15: {
16: #define CUBESIZE 12.0
17:   vec3  rv;
18:   float t, u;
19:
20:   rv = reflect ( vv, nv );
21:   t = 1.0e12;
22:   if ( rv.x > 0.0 ) t = (CUBESIZE-p.x)/rv.x;
23:   if ( rv.y > 0.0 ) { u = (CUBESIZE-p.y)/rv.y;  if ( u < t ) t = u; }
24:   if ( rv.z > 0.0 ) { u = (CUBESIZE-p.z)/rv.z;  if ( u < t ) t = u; }
25:   return p + t*rv;
```

[18] Nie ma wielkiego sensu dopisywanie tych instrukcji do procedury realizującej model Lamberta — trudno wymagać, aby powierzchnia matowa działała jak lustro.

```
#undef CUBESIZE
} /*ReflectTxtPoint*/

const vec3 e[6] = { vec3(-1,0,0), vec3(1,0,0), vec3(0,-1,0),
                    vec3(0,1,0), vec3(0,0,-1), vec3(0,0,1) };

vec4 TxtLight ( vec3 tp )
{
  vec4  tl;
  vec3  atp, lv;
  float a, b, dist;
  uint  i, cs, mask;

  atp = abs ( tp );
  a = tp.x, b = atp.x, cs = 0;
  if ( atp.y > b ) a = tp.y, b = atp.y, cs = 2;
  if ( atp.z > b ) a = tp.z, cs = 4;
  if ( a < 0 ) cs ++;
  atp = e[cs];
  tl = vec4 ( 0.0 );
  for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
    if ( (light.mask & mask) != 0 ) {
      tl += light.ls[i].ambient;
      lv = normalize ( posDifference ( light.ls[i].position, tp, dist ) );
      a = dot ( lv, atp );
      if ( a > 0.0 )
        tl += light.ls[i].direct*a;
    }
  return tl;
} /*TxtLight*/

vec3 BlinnPhongLighting ( void )
{
  vec3  normal, lv, vv, hv, Colour;
  ....

  normal = normalize ( In.Normal );
  vv = posDifference ( trb.eyepos, In.Position, dist );
  e = dot ( vv, normal );
  Colour = vec3(0.0);
  .... /* pętla przebiegająca po źródłach światła identyczna */
  .... /* jak na listingu 18.4 */
  lv = ReflectTxtPoint ( In.Position, -vv, normal );
  Colour += TxtLight ( lv ) * texture ( CubeMapTxt, lv ) * mm.specref;
  return clamp ( Colour, 0.0, 1.0 );
} /*BlinnPhongLighting*/
```

Do znalezienia punktu wspólnego promienia wtórnego ze ścianą jest użyte parametryczne przedstawienie promienia, $\boldsymbol{p}(t) = \boldsymbol{p}_0 + t\boldsymbol{v}$ dla $t \geqslant 0$. W liniach 22–24 obliczane są wartości parametru promienia odpowiadające przecięciom z tymi płaszczyznami ścian, które promień przecina, a w zmiennej `t` zostaje zapamiętana najmniejsza z tych wartości. W linii 25 jest obliczany odpowiadający jej punkt promienia.

Uwaga: W zastosowaniach, w których tekstura kostkowa reprezentuje krajobraz, tj. wielkość kostki jest „nieskończona", zamiast punktu przecięcia promienia ze ścianą należy podać wektor kierunkowy promienia wtórnego otrzymany za pomocą funkcji `reflect`.

Zadaniem procedury `TxtLight` jest obliczenie intensywności światła padającego na punkt ściany podany jako parametr. Instrukcje w liniach 39–44 rozpoznają, która to jest ściana, i przypisują zmiennej `atp` wektor normalny tej ściany. Wektor ten jest zorientowany do wnętrza pomieszczenia[19]. Pętla w liniach 46–53 przebiega po źródłach światła; dla każdego włączonego źródła światła do zmiennej `tl` jest dodawany składnik opisujący światło z tego źródła rozproszone w otoczeniu i jeśli światło oświetla wewnętrzną stronę ściany, to jest dodawany także składnik opisujący oświetlenie bezpośrednie.

Całkowita intensywność światła padającego na ścianę jest w linii 69 mnożona przez wartość tekstury w tym punkcie i przez czynnik opisujący zdolność powierzchni do odbijania światła w sposób lustrzany, a iloczyn jest dodawany do koloru wyprowadzanego na wyjście szadera.

Uwaga: Alternatywnym rozwiązaniem jest utworzenie tekstury, która zamiast koloru „farby" nałożonej na ściany opisuje „gotową" intensywność światła odbitego od tych ścian. Tekstura ta może uwzględniać cienie rzucane na ściany przez obiekty w pomieszczeniu.

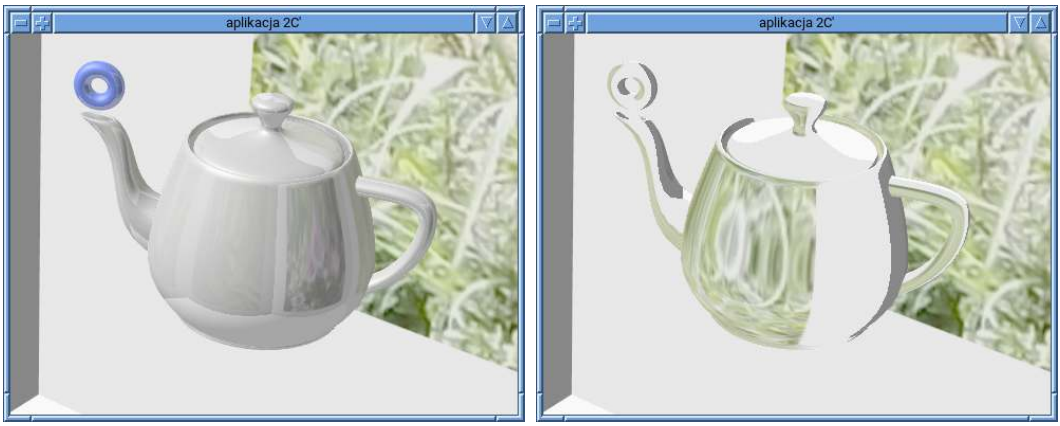
Obraz po lewej stronie rysunku 26.5 został otrzymany w opisany wyżej sposób; obraz ten nie jest całkiem poprawny, bo scena nie składa się z jednej bryły wypukłej i na przykład w powierzchni korpusu czajnika nie odbija się uchwyt, dziobek ani lewitujący nad dziobkiem torus. Aby otrzymać odbicia tych obiektów, należałoby znajdować przecięcia promieni wtórnych z innymi płatami Béziera w scenie. Rozwiązanie tego zadania, znacznie trudniejsze do zrealizowania niż przedstawiony przykład, to już „pełnowymiarowy" algorytm śledzenia promieni (*ray tracing*), pozostający poza zakresem tej książki[20].

Używając funkcji `refract` zamiast (lub oprócz) `reflect`, możemy otrzymać efekt załamania światła przez obiekt wykonany ze szkła lub wody[21]. Przykład jest pokazany na rysunku 26.5 z prawej strony. Jest tu uproszczenie na granicy oszustwa (lub nawet poza tą granicą): załamanie promienia zostało wyznaczone tylko na ścianie obiektu (czajnika lub torusa) położonej najbliżej obserwatora — obraz jest taki, jak gdyby promień wtórny znajdował się w całości (od swojego początku do punktu na ścianie pomieszczenia) w ośrodku gęstszym optycznie, na co zapewne większość osób oglądających obrazek nie zwróci uwagi. Poprawne

[19] Obserwator znajduje się w pomieszczeniu, nie trzeba zatem obliczać iloczynu skalarnego w celu zbadania, po której stronie ściany on jest.

[20] Największą trudność w implementacji śledzenia promieni sprawia potrzeba zachowania jednolitości obliczeń na GPU, zobacz podrozdział 9.9.

[21] ewentualnie zamrożonej



Rysunek 26.5. Obrazy odbitych i załamanych przez powierzchnie obiektów ścian pomieszczenia

geometrycznie rozwiązanie wymagałoby znalezienia przecięcia promienia wtórnego z tylną ścianą obiektu, wygenerowania przy użyciu funkcji `refract` promienia wtórnego drugiego rzędu, znalezienia jego przecięć z obiektami itd. Do tego jest również potrzebny algorytm śledzenia promieni, ale jak widać, nawet bez niego można otrzymać efektowne obrazy.

Uwaga: Dla funkcji `refract`, inaczej niż dla funkcji `reflect`, istotne znaczenie ma zwrot wektora normalnego (zobacz podrozdz. A.1). Do wygenerowania promienia wtórnego zastosowałem podprogram `RefractTxtPoint`, który powstał z `ReflectTxtPoint` przez zmianę nazwy i zastąpienie instrukcji w linii 20 instrukcją

```
rv = refract ( vv, dot ( nv, vv ) > 0.0 ? -nv : nv, eta );
```

Polecam ćwiczenie polegające na wymyśleniu i zaimplementowaniu sposobu nakładania tekstury kostkowej na ściany pomieszczenia będącego prostopadłościanem, którego krawędzie mogą mieć trzy różne długości.

26.4.3.*Cienie wokół źródeł światła

Algorytm cieni zrealizowany w aplikacji 2G można rozbudować tak, aby otrzymać cienie na obrazie sceny, w której źródła światła znajdują się pomiędzy obiektami. Do tego służą tekstury kostkowe, które po podłączeniu do pozaekranowego bufora ramki i narysowaniu sceny stają się pełną reprezentacją obszaru cienia wokół źródła światła. Warstwy tekstury kostkowej odpowiadają poszczególnym ścianom i można wykonać obrazy jednocześnie na nich wszystkich.

Zakładając, że wszystkie obiekty w scenie będą rysowane w taki sam sposób (np. przez wykonanie instrukcji `glDrawElements ( GL_TRIANGLES, ... );`), wystarczy mieć trzy różne programy szaderów; pierwszy służy do znajdowania obszarów cienia dla źródeł światła znajdujących się poza sceną, drugi do znajdowania obszarów cienia dla źródeł wewnątrz sceny i trzeci do wykonania końcowego obrazu. Jeśli reprezentacje obiektów wymagają

użycia różnych trybów rysowania (np. `GL_TRIANGLE_STRIP_ADJACENCY`, `GL_PATCHES`), to i programów trzeba więcej. Zobaczmy najważniejsze elementy przykładowej implementacji.

Listing 26.14. Zmieniony blok zmiennych jednolitych `LSBlock`

GLSL

```
struct LSPar { .... };  /* struktura identyczna jak na listingu 22.3 */

uniform LSBlock {
    uint  nls;
    uint  mask, shmask, cshmask;
    float max_depth;
    int   lightn;
    LSPar ls[MAX_NLIGHTS];
  } light;
```

Listing 26.14 przedstawia blok zmiennych jednolitych opisujących źródła światła. Są w nim trzy nowe pola: `cshmask` jest maską bitową, której bity odpowiadają poszczególnym źródłom światła. Mamy zatem trzy maski bitowe: i-ty bit pola `mask` określa, czy i-te źródło światła jest włączone. Jeśli i-ty bit pola `shmask` jest jedynką, to istnieje „płaska” tekstura reprezentująca obszar cienia. Niezerowy i-ty bit pola `cshmask` oznacza, że obszar cienia dla i-tego źródła światła jest reprezentowany przez teksturę kostkową i może całkowicie otaczać źródło światła. Dla każdego źródła odpowiedni bit może być jedynką co najwyżej w jednym z pól `shmask` i `cshmask`. Nowe pole `max_depth` jest wspólną dla wszystkich źródeł światła umieszczonych wewnątrz sceny wartością parametru f (`far`) wszystkich ostrosłupów widzenia. Pole `lightn` zawiera numer źródła światła, którego obszar cienia jest znajdowany. Do opisanych wyżej zmian trzeba dostosować procedury w C nadające wartości zmiennym w tym bloku zmiennych jednolitych (i w szczególności przenumerować elementy tablicy `lsbofs`), czego opis pomijam.

Na listingu 26.15 są pokazane zmiany struktur danych opisujących źródła światła w pamięci CPU; tu objaśnienia są zbędne.

Listing 26.15. Zmieniona struktura `LightBl`

C

```
typedef struct LSPar { .... } LSPar;  /* identycznie jak na listingu 22.1*/

typedef struct LightBl {
    GLuint  nls, mask, shmask, cshmask;
    GLfloat max_depth;
    LSPar   ls[MAX_NLIGHTS];
  } LightBl;
```

Procedura przygotowująca tekstury cienia jest pokazana na listingu 26.16; powstała ona przez modyfikację procedury z listingu 22.8. Pierwsza zmiana to dodanie parametru `cubesh`, którego wartość niezerowa oznacza, że źródło światła jest umieszczone wewnątrz sceny.

Listing 26.16. Przygotowanie tekstur cienia

C

```
void ConstructShadowTxtFBO ( LightBl *light,
                             int l, char cubesh )
{
  GLuint fbo, txt, mask;
  GLenum target;

  if ( l < 0 || l >= MAX_NLIGHTS )
    return;
  glGenTextures ( 1, &txt );
  glGenFramebuffers ( 1, &fbo );
  if ( (light->ls[l].shadow_txt = txt) &&
       (light->ls[l].shadow_fbo = fbo) ) {
    glBindFramebuffer ( GL_FRAMEBUFFER, fbo );
    glActiveTexture ( GL_TEXTURE2+l );
    glBindBuffer ( GL_UNIFORM_BUFFER, light->lsbuf );
    mask = 0x01 << l;
    if ( cubesh ) {
      light->cshmask |= mask;
      glBufferSubData ( GL_UNIFORM_BUFFER, lsbofs[3], sizeof(GLuint),
                        &light->cshmask );
      glBindTexture ( target = GL_TEXTURE_CUBE_MAP, txt );
    }
    else {
      light->shmask |= mask;
      glBufferSubData ( GL_UNIFORM_BUFFER, lsbofs[2], sizeof(GLuint),
                        &light->shmask );
      glBindTexture ( target = GL_TEXTURE_2D, txt );
      glTexParameteri ( target, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
      glTexParameteri ( target, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
    }
    glTexStorage2D ( target, 1, GL_DEPTH_COMPONENT32,
                     SHADOW_MAP_SIZE, SHADOW_MAP_SIZE );
    glTexParameteri ( target, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
    glTexParameteri ( target, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
    glBindTexture ( target, 0 );
    glFramebufferTexture ( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, txt, 0 );
    glDrawBuffer ( GL_NONE );
    if ( glCheckFramebufferStatus ( GL_FRAMEBUFFER ) !=
              GL_FRAMEBUFFER_COMPLETE )
      ExitOnError ( "ConstructShadowTxtFBO" );
    glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
    ExitIfGLError ( "ConstructShadowTxtFBO" );
  }
} /*ConstructShadowTxtFBO*/
```

W takim przypadku po przywiązaniu do celu GL_TEXTURE_CUBE_MAP, za pośrednictwem którego parametrom tekstury nadaje się odpowiednie wartości, zostaje utworzona kostkowa tekstura cienia. Jeśli źródło światła jest umieszczone poza sceną, to do utworzenia tekstury jest używany cel GL_TEXTURE_2D, tak samo jak w aplikacji 2G.

Jest jeszcze jedna zmiana: brak (zarówno dla tekstury kostkowej, jak i płaskiej) nadania wartości parametrom GL_TEXTURE_COMPARE_MODE i GL_TEXTURE_COMPARE_FUNC (zobacz listing 22.8, linie 24–26). W przedstawionej tu implementacji funkcja texture dla tekstur cienia ma podawać *wartość tekstury*, a nie wynik porównania tej wartości z głębokością punktu. Porównanie będzie dokonane jawnie przez szader. Jeszcze jeden szczegół: szader fragmentów używany do wykonania końcowego obrazu ma dwie tablice tekstur cienia, z teksturami płaskimi i kostkowymi. Jest dopuszczalne zadeklarowanie w treści szadera tekstur różnych rodzajów korzystających z tego samego punktu dowiązania, zatem tekstury w tych tablicach korzystają z punktów o numerach 2 do 9. Numer punktu jest określany w linii 14.

Procedura pokazana na listingu 26.17 jest rozszerzoną wersją procedury z listingu 22.1. Jeśli odpowiedni bit maski wskazuje na użycie tekstury kostkowej, to instrukcje w liniach 14–15 konstruują macierz przejścia od układu obserwatora do układu kostki standardowej dostosowaną do źródeł światła położonych wewnątrz sceny.

Parametr R dla źródeł światła położonych wewnątrz sceny określa maksymalną współrzędną z (głębokość) punktów powierzchni obiektów do narysowania w związanych z tymi źródłami układach obserwatora. Aby ją znaleźć, trzeba znaleźć prostopadłościan określony przez minimalne i maksymalne współrzędne x, y i z punktów sceny w układzie świata. Znając otaczający scenę prostopadłościan $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \times [z_{\min}, z_{\max}]$, dla źródła światła położonego w punkcie (x_i, y_i, z_i) należy obliczyć

$$R_i = \max\left(|x_i - x_{\min}|, |x_i - x_{\max}|, |y_i - y_{\min}|, |y_i - y_{\max}|, |z_i - z_{\min}|, |z_i - z_{\max}|\right),$$

a potem przypisać parametrowi R największą z tych liczb.

Wartość R parametru R, przekazana dalej jako ostatni parametr (far) procedury M4x4-Frustumf, zapewnia, że światło może dotrzeć z każdego umieszczonego w scenie źródła do jej najdalszych zakątków. Liczba R musi też zostać zapamiętana w polu max_depth bloku zmiennych jednolitych LSBlock (listing 26.14). W konstrukcji macierzy przejścia do układu kostki standardowej trzeba także podać parametr near, który określa (w układzie obserwatora dla źródła światła) minimalną głębokość obiektów, na które może być rzucany cień. W linii 14 jest przyjmowana liczba $\frac{1}{100}R$ — punkty o mniejszej głębokości będą zawsze oświetlone (co może być niepożądane), ale mniejsza wartość parametru near stwarza ryzyko nadmiernej utraty dokładności w obliczeniach cienia[22].

Przejście od układu świata do układu kostki standardowej, dla źródła światła wewnątrz sceny, składa się z trzech kroków. Pierwszym jest przesunięcie, które położenie źródła światła przeprowadza na początek układu. Drugie przekształcenie jest jednym z opisanych dalej sześciu obrotów, innym dla każdej ściany sześcianu. Macierze tych obrotów są opisane w treści szaderów. Trzecie przekształcenie jest przejściem do układu kostki standardowej, którego

[22] Zachęcam jednak do zbadania tego w eksperymentach.

Listing 26.17. Procedura `SetupShadowTransformations`

C

```
1: void SetupShadowTransformations ( int l, float sc[3], float R )
2: {
3:   GLfloat *lvm, *lpm;
4:   GLfloat lpos[3], v[3], d, n, s, t;
5:   int     i;
6:
7:   if ( l < 0 || l >= MAX_NLIGHTS )
8:     return;
9:   if ( light.ls[l].shadow_txt ) {
10:    lvm = light.ls[l].shadow_view;
11:    lpm = light.ls[l].shadow_proj;
12:    if ( light.ls[l].position[3] != 0.0 ) {
13:      if ( light.cshmask & (0x01 << l) ) {
14:        n = 0.01*R;
15:        M4x4Frustumf ( lpm, NULL, -n, n, -n, n, n, R );
16:      }
17:      else { /* obliczenie dla źródeł światła poza sceną */
18:        .... /* w skończonej odległości */
19:      }
20:    }
21:    else {   /* obliczenie dla źródeł światła położonych */
22:      ....   /* nieskończenie daleko */
23:    }
24:  }
25: } /*SetupShadowTransformations*/
```

macierz, P, jest konstruowana w linii 15. Parametry `left`, `right`, `bottom` i `top` są dobrane tak, że opisane przez nie i odpowiednio poobracane ostrosłupy widzenia dają w sumie sześcian o boku $2R$, z którego środka jest usunięty sześcian o boku 100 razy krótszym. Zarówno wspomniane obroty, jak i przejście do układu kostki standardowej są wspólne dla wszystkich źródeł światła wewnątrz sceny (rys. 26.6).

Wywołanie procedury `UpdateShadowMatrix` z listingu 22.8, obliczającej i umieszczającej w pamięci GPU iloczyn BP_lV_l, który jest macierzą przejścia od układu świata do układu tekstury cienia, jest dla kostkowych tekstur cienia zbędne, bo przekształcenia współrzędnych punktu w celu zbadania, czy jest on w cieniu, są w tym przypadku realizowane w inny sposób. Warto zmodyfikować tę procedurę, aby spowodować natychmiastowy powrót, jeśli odpowiedni bit w polu `cshmask` jest niezerowy.

Listing 26.18 przedstawia procedury rysowania obrazu sceny z cieniami[23]. Wyświetlenia wszystkich obiektów dokonuje procedura `DrawMyScene` przy użyciu programów szaderów wybranych na podstawie drugiego parametru, stałej ukrytej za znaczącą nazwą makrodefinicji zapisanej w kodzie aplikacji. Stała `ROPT_FINAL` wybiera programy obliczające kolory pikseli końcowego obrazu (i wykonuje wszelkie inne niezbędne czynności, np. przywiązuje

[23] Zadaniem dla Czytelnika jest dostosowanie ich do własnego projektu.

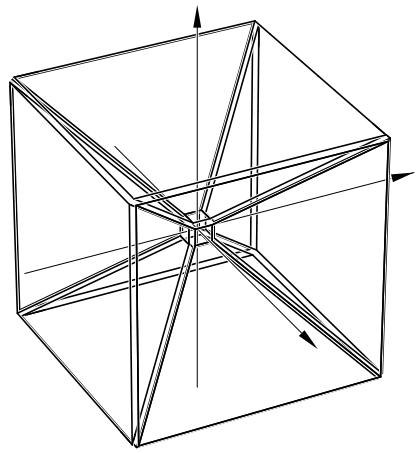
Listing 26.18. Procedury rysowania sceny

C

```
1: void DrawSceneToShadows ( AppData *ad )
2: {
3:   int    l;
4:   GLuint mask;
5:
6:   glViewport ( 0, 0, SHADOW_MAP_SIZE, SHADOW_MAP_SIZE );
7:   glEnable ( GL_POLYGON_OFFSET_FILL );
8:   glPolygonOffset ( 2.0f, 4.0f );
9:   for ( l = 0, mask = 0x01;  l < ad->light.nls;  l++, mask <<= 1 )
10:    if ( (ad->light.mask & mask) && ad->light.ls[l].shadow_fbo ) {
11:      BindShadowTxtFBO ( l );
12:      glClear ( GL_DEPTH_BUFFER_BIT );
13:      if ( ad->light.shmask & mask )
14:        DrawMyScene ( ad, ROPT_SHADOWS );
15:      else if ( light.cshmask & mask )
16:        DrawMyScene ( ad, ROPT_CUBESHADOWS );
17:    }
18:  glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
19:  glDisable ( GL_POLYGON_OFFSET_FILL );
20:  for ( l = 0, mask = 0x01;  l < ad->light.nls;  l++, mask <<= 1 ) {
21:    glActiveTexture ( GL_TEXTURE0+MAX_TEXTURES+l );
22:    if ( ad->light.shmask & mask )
23:      glBindTexture ( GL_TEXTURE_2D, ad->light.ls[l].shadow_txt );
24:    else if ( ad->light.cshmask & mask )
25:      glBindTexture ( GL_TEXTURE_CUBE_MAP, ad->light.ls[l].shadow_txt );
26:  }
27:  ExitIfGLError ( "DrawSceneToShadows" );
28: } /*DrawSceneToShadows*/
29:
30: void _RedrawMyWorld ( AppData *ad )
31: {
32:  glEnable ( GL_DEPTH_TEST );
33:  DrawSceneToShadows ( ad );
34:  glViewport ( 0, 0, ad->camera.win_width, ad->camera.win_height );
35:  glClear ( GL_DEPTH_BUFFER_BIT );
36:  LoadVPMatrix ( ad->trans.vm, ad->trans.pm, ad->trans.eyepos );
37:  DrawMyScene ( ad, ROPT_FINAL );
38: } /*_RedrawMyWorld*/
```

tekstury nakładane na obiekty, wybiera materiał), stała ROPT_SHADOWS wybiera programy odpowiednie do znajdowania obszarów cienia dla źródeł światła położonych na zewnątrz sceny, a programy używane po podaniu stałej ROPT_CUBESHADOWS znajdują obszar cienia otaczający źródło światła.

Obszary cienia dla kolejnych źródeł światła są znajdowane w pętli w liniach 9–17. Procedura BindShadowTxtFBO wybiera pozaekranowy bufor ramki, której załącznikiem jest teks-



Rysunek 26.6. Ostrosłupy widzenia wokół źródła światła (nieco rozsunięte)

tura cienia dla kolejnego źródła światła i wywołuje procedurę `LoadVPMatrix`, która przesyła odpowiednie macierze do bloku zmiennych jednolitych `TransBlock`. Do tej procedury (listing 22.8) trzeba jeszcze dodać instrukcje przypisujące polu `lightn` bloku `LSBlock` numer bieżącego źródła swiatła (czyli wartość zmiennej `l`). Po wykonaniu obliczeń dla wszystkich źródeł światła przywracany jest stan domyślny, w którym aktywnym buforem ramki jest bufor związany z oknem na ekranie, a potem w pętli tekstury cienia są przywiązywane do odpowiednich punktów dowiązania: do celu `GL_TEXTURE_2D` dla źródeł światła poza sceną i do celu `GL_TEXTURE_CUBE_MAP` dla źródeł wewnątrz sceny. Po znalezieniu tekstur cienia scena jest rysowana w oknie. Instrukcje poprzedzające wywołanie procedury `DrawMyScene` w linii 37 objaśnień nie wymagają.

Makrodefinicja `MAX_TEXTURES` określa liczbę punktów dowiązania zarezerwowanych dla tekstur nakładanych na obiekty (zobacz listing 19.12). Tekstury reprezentujące obszary cienia zajmują kolejne punkty dowiązania.

Zobaczmy teraz szadery. Listing 26.19 przedstawia szader wierzchołków programów znajdujących reprezentacje obszarów cienia. Oblicza on tylko położenie wierzchołka w układzie świata, wszelkie inne atrybuty wierzchołka są ignorowane, szkoda na nie czasu.

Listing 26.19. Szader wierzchołków programów znajdowania cieni

GLSL

```
1: #version 450 core
2:
3: layout(location=0) in vec4 in_Position;
4: uniform TransBlock { .... } trb;
5:
6: void main ( void )
7: {
8:   gl_Position = trb.mm * in_Position;
9: } /*main*/
```

Szader geometrii na listingu 26.20 jest częścią programu znajdowania obszaru cienia dla źródeł światła poza sceną. Przetwarza on wierzchołki jednego trójkąta, co polega na przekształceniu ich do układu kostki standardowej; w polu `vpm` bloku `TransBlock` jest iloczyn macierzy V przejścia od układu świata do układu obserwatora związanego ze źródłem światła i macierzy P, skonstruowanych w sposób opisany w podrozdziale 22.1. Szader fragmentów tego programu zawiera tylko pustą procedurę `main` — jedynym jej zadaniem jest być obecną, aby po powrocie z niej potok przetwarzania grafiki zapisał w teksturze cienia głębokość fragmentu.

Listing 26.20. Szader geometrii pierwszego programu znajdowania cieni

GLSL

```
1: #version 450 core
2:
3: layout(triangles) in;
4: layout(triangle_strip,max_vertices=3) out;
5:
6: uniform TransBlock { .... } trb;
7:
8: void main ( void )
9: {
10:   int i;
11:
12:   for ( i = 0; i < 3; i++ ) {
13:     gl_Position = trb.vpm * gl_in[i].gl_Position;
14:     EmitVertex ();
15:   }
16:   EndPrimitive ();
17: } /*main*/
```

Ciekawszą pracę ma do wykonania drugi program szaderów. Szader geometrii pokazany na listingu 26.21 ma 6 wywołań dla każdego trójkąta przekazanego przez potok przetwarzania grafiki. Wynik każdego z tych wywołań trafia na jedną z sześciu warstw tekstury cienia. Przekształcanie wierzchołków odbywa się w liniach 32–34; odejmowanie położenia źródła światła[24] i mnożenie przez jedną z macierzy z tablicy `cubtr` w linii 32 jest przejściem do układu obserwatora. Każda z tych macierzy reprezentuje obrót, w wyniku którego rzutnia (przednia ściana ostrosłupa widzenia) jest równoległa do odpowiedniej ściany kostki[25]. W linii 33 następuje obliczanie głębokości wierzchołka, która zostanie przekazana na wyjście szadera. W tym rozwiązaniu „nie działa" korekta głębokości, która ma na celu przeciwdziałanie zasłanianiu od światła punktów płaszczyzny przez nie same — dlatego szader sam dokonuje korekty, mnożąc głębokość (podzieloną przez R, czyli przeskalowaną do przedziału $[0,1]$)

[24] Współrzędna wagowa wektora `light.ls[lightn].position` jest równa 1.

[25] Mamy tu przykład zastosowania konstruktora macierzy w deklaracji zmiennej z nadaną wartością. W nawiasach są podane współczynniki w kolejnych kolumnach macierzy. Kwalifikator `const` zabrania zmieniania zawartości tablicy `cubtr`.

przez czynnik FCT, odrobinę większy od 1. W linii 34 następuje przejście od układu obserwatora do układu kostki standardowej.

Listing 26.21. Szader geometrii drugiego programu znajdowania cieni

GLSL

```
1: #version 450 core
2:
3: layout(invocations=6,triangles) in;
4: layout(triangle_strip,max_vertices=3) out;
5:
6: out float Depth;
7:
8: uniform TransBlock { .... } trb;
9:
10: struct LSPar { .... };
11: uniform LSBlock { .... } light;
12:
13: const mat4 cubtr[6] =
14:   {mat4( 0, 0,-1, 0,  0,-1, 0, 0, -1, 0, 0, 0,  0, 0, 0, 1),
15:    mat4( 0, 0, 1, 0,  0,-1, 0, 0,  1, 0, 0, 0,  0, 0, 0, 1),
16:    mat4( 1, 0, 0, 0,  0, 0,-1, 0,  0, 1, 0, 0,  0, 0, 0, 1),
17:    mat4( 1, 0, 0, 0,  0, 0, 1, 0,  0,-1, 0, 0,  0, 0, 0, 1),
18:    mat4( 1, 0, 0, 0,  0,-1, 0, 0,  0, 0,-1, 0,  0, 0, 0, 1),
19:    mat4(-1, 0, 0, 0,  0,-1, 0, 0,  0, 0, 1, 0,  0, 0, 0, 1)};
20:
21: #define FCT 1.00001
22: void main ( void )
23: {
24:   int  i, k;
25:   vec3 lpos;
26:   vec4 p;
27:
28:   gl_Layer = k = gl_InvocationID;
29:   lpos = light.ls[light.lightn].position.xyz;
30:   for ( i = 0; i < 3; i++ ) {
31:     p = gl_in[i].gl_Position;
32:     p = cubtr[k] * vec4 ( p.xyz/p.w-lpos, 1.0 );
33:     Depth = -(p.z / light.max_depth) * FCT;
34:     gl_Position = trb.pm * p;
35:     EmitVertex ();
36:   }
37:   EndPrimitive ();
38: } /*main*/
```

Zmienna wejściowa Depth szadera fragmentów na listingu 26.22 ma wartość otrzymaną przez interpolację głębokości wierzchołków trójkąta wyprowadzonych przez szader geo-

metrii. Szader przypisuje tę wartość zmiennej `gl_FragDepth`, skąd trafi ona do bufora głębokości, tj. do odpowiedniej ściany tekstury będącej załącznikiem bufora ramki.

Listing 26.22. Szader fragmentów drugiego programu znajdowania cieni

GLSL

```
1: #version 450 core
2:
3: in float Depth;
4:
5: void main ( void )
6: {
7:   gl_FragDepth = Depth;
8: } /*main*/
```

Uwaga: Potok przetwarzania grafiki oblicza głębokość fragmentu na podstawie współrzędnej z w układzie kostki standardowej; leży ona w przedziale $[-1,1]$ i jeśli jest stosowane rzutowanie perspektywiczne, to przekształcenie współrzędnej z między układami obserwatora a kostki standardowej jest funkcją homograficzną, a zatem nieliniową (zobacz podrozdz. 6.2). Szader fragmentów otrzymuje w zmiennej wejściowej `gl_FragCoord.z` liczbę $\zeta = (z+1)/2$. Domyślnie liczba ta jest przekazywana na wyjście. Szader fragmentów może przypisać wbudowanej zmiennej wyjściowej `gl_FragDepth` inną wartość, ale jeśli szader zawiera taką instrukcję przypisania i nie zostanie ona wykonana (bo np. jest w instrukcji warunkowej), to wyjściowa głębokość jest nieokreślona. Dlatego szader fragmentów, który zamierza modyfikować głębokość fragmentu tylko gdy jest spełniony pewien warunek, w razie jego niespełnienia musi wykonać instrukcję przypisania `gl_FragDepth = gl_FragCoord.z;`.

Teraz zobaczmy sposób użycia tekstur cienia podczas wykonywania końcowego obrazu. Program szaderów (dla sceny, w której nie ma płatów) składa się z szaderów wierzchołków, geometrii i fragmentów. Szader wierzchołków jest taki sam jak na listingu 10.2. Zadaniem szadera geometrii jest tylko wyprowadzenie wierzchołków trójkąta z atrybutami współrzędnych położenia w układzie kostki standardowej, położenia w układzie świata, koloru i wektora normalnego. W przedstawianej tu implementacji szader nie oblicza współrzędnych cienia dla wierzchołka (tych, które w aplikacji 2G były przekazywane w tablicy `ShadowPos`, zobacz listing 22.3), bo to obliczenie wykonuje szader fragmentów, pokazany na listingu 26.23. Dlatego jest tu zastosowany szader geometrii pokazany na listingu 10.3.

Szader może obliczać oświetlenie według modelu Lamberta, Blinna-Phonga lub innego; pętla w liniach 60–70 przebiega po źródłach światła. Warunek sprawdzany w linii 63 jest spełniony, jeśli została utworzona tekstura cienia dla źródła światła umieszczonego poza sceną. Sprawdzenia, czy fragment jest w cieniu, dokonuje wtedy funkcja `IsEnlighted`. Jeśli warunek w linii 65 jest spełniony, to źródło światła jest wewnątrz sceny i jego tekstura cienia jest kostkowa; wtedy przystępuje do pracy funkcja `IsEnlightedCube`. Jeśli źródło światła nie ma tekstury cienia, to zmienna `s` otrzymuje wartość 1, co oznacza, że fragment jest oświetlony. Jeśli `s == 0`, to dalsze instrukcje powinny zostać pominięte.

Listing 26.23. Szader fragmentów do wykonania końcowego obrazu

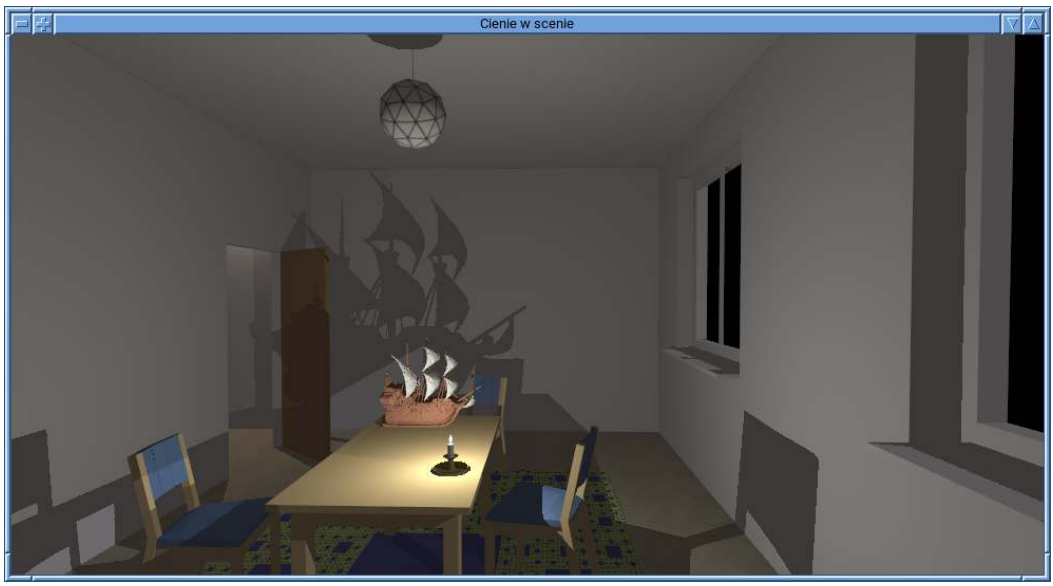
GLSL

```
#version 450

#define MAX_TEXTURES 4  /* musi być tak samo jak w kodzie w C */

in FVertex { .... } In; /* listing 10.4 */
out vec4 out_Colour;

uniform TransBlock { .... } trb;  /* listing 26.14 */
struct LSPar { .... };
uniform LSBlock { .... } light;

layout(binding=MAX_TEXTURES) uniform sampler2D   shtex[MAX_NLIGHTS];
layout(binding=MAX_TEXTURES) uniform samplerCube cshtex[MAX_NLIGHTS];

const mat4 cubtr[6] = { .... };   /* listing 26.21 */

float IsEnlighted ( uint l )
{
  vec4 pos;

  pos = light.ls[l].shadow_vpm * vec4(In.Position,1.0);
  pos.xyz /= pos.w;
  return float ( pos.z <= texture ( shtex[l], pos.xy ).r );
} /*IsEnlighted*/

float CubeDepth ( vec3 v )
{
  v = abs ( v );
  if ( v.x >= v.y ) {
    if ( v.x >= v.z )    return v.x;
    else                 return v.z;
  }
  else if ( v.y >= v.z ) return v.y;
  else                   return v.z;
} /*CubeDepth*/

float IsEnlightedCube ( uint l )
{
  vec3  v;
  float depth;

  depth = CubeDepth ( v = In.Position-light.ls[l].position.xyz ) /
          light.max_depth;
  return float ( depth <= texture ( cshtex[l], v ).r );
} /*IsEnlightedCube*/
```

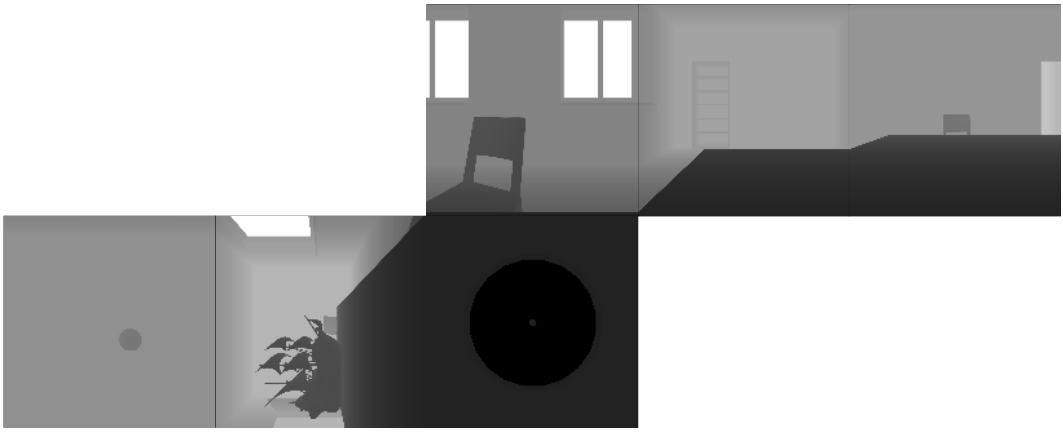
```
46:
47: vec3 posDifference ( vec4 p, vec3 pos, out float dist ) { .... }
48: float attFactor ( vec3 att, float dist ) { .... } /* listing 10.4 */
49:
50: vec3 LambertLighting ( void )
51: {
52:   vec3  normal, lv, vv, Colour;
53:   float s, d, dist;
54:   uint  l, mask;
55:
56:   normal = normalize ( In.Normal );
57:   vv = posDifference ( trb.eyepos, In.Position, dist );
58:   e = dot ( vv, normal );
59:   Colour = MatEmission ( dot ( normal, vv ) );
60:   for ( l = 0, mask = 0x01;  l < light.nls;  l++, mask <<= 1 )
61:     if ( (light.mask & mask) != 0 ) {
62:       Colour += light.ls[l].ambient * In.Colour;
63:       if ( (light.shmask & mask) != 0 )
64:         s = IsEnlighted ( l );
65:       else if ( (light.cshmask & mask) != 0 )
66:         s = IsEnlightedCube ( l );
67:       else
68:         s = 1.0;
69:       .... /* dalej instrukcje jak w liniach 32-50 na listingu 22.5 */
70:     }
71:   return = clamp ( Colour, 0.0, 1.0 );
72: } /*LambertLighting*/
```

Brak podanych na wejściu współrzędnych fragmentu w układzie kostki jednostkowej dla źródeł światła (jakie były używane w aplikacji 2G) powoduje konieczność obliczenia ich przez szader. W liniach 21–22 funkcja `IsEnlighted` oblicza te współrzędne na podstawie położenia fragmentu w układzie świata. Zwróćmy uwagę, że wykonywane tu przekształcenie jest określone tak samo jak to, któremu szader na listingu 22.3 (w linii 49) poddaje wierzchołki trójkątów. Funkcja `texture`, której pierwszy parametr jest typu `sampler2D`, wymaga podania drugiego parametru typu `vec2`; w linii 23 składa się on ze współrzędnych x, y położenia fragmentu w kostce jednostkowej. Wartość tekstury (mająca określoną tylko współrzędną `r`), która jest głębokością punktu na brzegu obszaru cienia, jest porównywana z głębokością (współrzędną ζ) fragmentu. Funkcja `float` zamienia wynik porównania `false` albo `true` na liczbę zmiennopozycyjną 0 albo 1.

Funkcja `IsEnlightedCube`, badająca, czy fragment jest w cieniu dla źródła światła wewnątrz sceny, w linii 42 oblicza wektor $\boldsymbol{v}$ — różnicę położeń fragmentu i źródła światła w układzie świata i wywołuje funkcję `CubeDepth`, podając jej ten wektor jako parametr. Wartością tej funkcji jest największa spośród wartości bezwzględnych współrzędnych x, y i z wektora $\boldsymbol{v}$; po podzieleniu przez R (w liniach 42–43) daje to głębokość punktu $\zeta \in [0,1]$, która jest porównywana z wartością tekstury głębokości w linii 44.



Rysunek 26.7. Obraz sceny z cieniami od świecy



Rysunek 26.8. Tekstura obszaru cienia od świecy

Rysunek 26.7 przedstawia obraz sceny oświetlonej przez trzy źródła światła, z których jedno (Księżyc) jest za oknami pokoju, a pozostałe dwa (płomień świecy i lampa w przedpokoju) są umieszczone wewnątrz sceny. Na rysunku 26.8 jest pokazana tekstura reprezentująca obszar cienia dla świecy. Na zakończenie rozdziału proponuję kilka tematów do zastanowienia; mają one formę mniej lub bardziej zaawansowanych ćwiczeń.

Ćwiczenia

1. Napisz i uruchom aplikację wykonującą obrazy z cieniami od źródeł światła wewnątrz sceny, korzystającą z opisanych wyżej szaderów.
2. Rozszerz szadery, implementując teksturowanie, rysowanie płatów itp.
3. Wywołana w linii 15 na listingu 26.17 procedura `M4x4Frustumf` konstruuje macierz

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{101}{99} & -\frac{2}{99}R \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

   Zbadaj, czy na podstawie tego wzoru da się *zauważalnie* przyspieszyć przejście do układu kostki standardowej podczas znajdowania obszaru cienia.
4. Przedyskutuj wady i zalety (wpływ na koszt obliczeń) dwóch rozwiązań: przekazywania z części przedniej do tylnej potoku przetwarzania grafiki tablicy wektorów współrzędnych cienia (które będą interpolowane przez etap rasteryzacji) i przekazywania tylko położenia wierzchołka w układzie świata (aby na podstawie współrzędnych położenia fragmentu szader fragmentów obliczał współrzędne cienia). Wnioski z tej dyskusji skonfrontuj z eksperymentem.
5. *Zaimplementuj metodę PCF dla źródeł światła wewnątrz sceny.
6. *Napisz odpowiednie szadery i aplikację, która wszystkie reprezentacje obszarów cienia od kilku źródeł światła wewnątrz sceny znajduje *w jednym etapie* rysowania. Do tego celu należy użyć **wielowarstwowej tekstury kostkowej**; tworzy się ją i używa za pośrednictwem celu `GL_TEXTURE_CUBE_MAP_ARRAY`, a szadery mają do niej dostęp za pomocą zmiennych typu `samplerCubeArray`. Potrzebne informacje wyszukaj w dokumentacji OpenGL-a i GLSL-a.

27

*Opóźnione cieniowanie

W opisanych dotąd aplikacjach kolory pikseli są obliczane przez szadery fragmentów na końcu potoku przetwarzania grafiki. Przedstawiona w tym rozdziale technika zwana **opóźnionym cieniowaniem** (*deferred shading*) polega na zapisaniu przez szader fragmentów informacji, na podstawie których kolory pikseli są obliczane później. Podejście to daje wiele korzyści (właściwie to dwie, ale jakie!). Pierwszą z nich jest znaczna oszczędność czasu, jeśli obliczanie koloru (np. na podstawie któregoś z modeli oświetlenia przedstawionych w następnym rozdziale lub jeszcze bardziej wyrafinowanego) długo trwa, a potem kolor ten nie został przypisany pikselowi, bo fragment nie przeszedł testu widoczności, albo został przypisany, ale później piksel otrzymał kolor innego fragmentu. W opóźnionym cieniowaniu kolor piksela jest na podstawie zapamiętanych informacji obliczany tylko raz i jest to kolor obiektu widocznego w tym pikselu. Druga korzyść to możliwość obliczania koloru piksela na podstawie informacji zapamiętanych dla tego piksela i pikseli sąsiednich; można wtedy użyć dowolnych metod przetwarzania obrazu, aby osiągnąć różne efekty specjalne.

Opóźnione cieniowanie oczywiście ma też wady. Proces rysowania jest bardziej skomplikowany, a informacje, które trzeba zapamiętać, zajmują kilkanaście razy więcej miejsca niż tablica z kolorami pikseli, czyli końcowy obraz. Ponadto nie jest wykonalny antyaliasing przez wielokrotne próbkowanie (*multisampling*). Aby otrzymać obrazy antyaliasowane, trzeba wykonać nadpróbkowanie (*supersampling*), tworząc w pierwszym etapie obraz w większej rozdzielczości, co zajmuje jeszcze więcej miejsca i jednak zabiera więcej czasu[1].

Zestaw tablic, w których mają być przechowywane informacje potrzebne do wykonania końcowego obrazu, jest nazywany **G-buforem**; litera G wzięła się stąd, że znaczna część tych informacji dotyczy geometrii rysowanej sceny. W opisanej tu implementacji G-bufor składa się z tekstur dołączonych do pozaekranowego bufora ramki. Drugi etap rysowania wykona szader obliczeniowy, czytający potrzebne dane z tych tekstur i zapisujący wynik obliczeń

[1]Tzw. **bufor wielopróbkowy** (*multisample buffer*, zobacz [1]) nie jest dostępny dla aplikacji ani szaderów. Zauważmy też, że w wielokrotnym próbkowaniu punkty wyznaczane w etapie rasteryzacji odcinków i trójkątów są dla poprawienia końcowego efektu rozmieszczone w obszarze piksela nieregularnie (podobnie do przykładów na rys. 25.5). Zwykłe przyjęcie dwu-, trzy- lub czterokrotnie większej rozdzielczości obrazu skutkuje regularnym rozmieszczeniem punktów w pikselu, co daje gorsze wyniki.

w obrazie końcowym. Ale aplikacje napisane zgodnie ze specyfikacją OpenGL-a, w której szadery obliczeniowe są niedostępne (wcześniejszą niż 4.3) muszą realizować drugi etap obliczeń przy użyciu programu z potokiem przetwarzania grafiki (zawierającym szadery wierzchołków i fragmentów), co zostawiam Czytelnikom jako temat do własnych przemyśleń.

27.1. Implementacja G-bufora

Do zrealizowania opisanych dalej przykładów opóźnionego cieniowania przyda się G-bufor, w którym dla każdego piksela będą przechowywane: wektor współrzędnych położenia fragmentu w układzie świata $\boldsymbol{p}$, wektor normalny powierzchni $\boldsymbol{n}$, wektor normalny trójkąta przybliżającego powierzchnię $\boldsymbol{m}$, wektor współrzędnych tekstury $\boldsymbol{u}$ i indeks do tablicy materiałów. Ten ostatni jest niewielką liczbą całkowitą, więc do jego przechowywania wystarczy jeden bajt[2]. Pozostałe informacje muszą być przechowywane w postaci zmiennopozycyjnej (w pojedynczej precyzji). Punkt $\boldsymbol{p}$ i wektory $\boldsymbol{n}$ i $\boldsymbol{m}$ mają trzy współrzędne, ale tekstury użyte jako załączniki pozaekranowego bufora ramki muszą mieć 1, 2 albo 4 składowe. Dlatego na informacje zapamiętane dla każdego piksela w G-buforze potrzebne będzie 57 bajtów, zamiast czterech bajtów potrzebnych dla składowych r, g, b, a piksela w końcowym obrazie.

Na listingu 27.1 jest pokazany szader fragmentów, który zamiast obliczać kolor na podstawie modelu oświetlenia, otrzymane na wejściu dane (wytworzone w etapie rasteryzacji przez interpolację atrybutów wierzchołków) przekazuje na wyjście, skąd trafią do G-bufora. Zmienne wyjściowe szadera są zadeklarowane w liniach 10–14. Kwalifikator `layout` każdej z nich zawiera numer położenia zmiennej; jest to numer załącznika koloru w pozaekranowym buforze ramki, w którym wykonamy pierwszy etap rysowania.

Blok zmiennych jednolitych `MatBlock` w treści tego szadera został skrócony (linia 16, porównaj z listingiem 18.1, linie 18–21 i 27.4, linia 22); jest w nim tylko pierwsze pole, przechowujące numer materiału, z którego jest wykonany rysowany w danej chwili obiekt. Oczywiście, aby pozostałe programy szaderów w aplikacji działały, przesunięcia pól w tym bloku względem jego początku muszą być odczytane z programu zawierającego pełny opis tego bloku, ale pierwsze pole w bloku o opisie skróconym ma takie samo przesunięcie[3].

Razem z wektorami normalnymi jest wyprowadzana (w czwartej składowej) głębokość fragmentu. Dzięki niej możemy odtworzyć wektor $\boldsymbol{p}$ bez zapamiętywania go, a zatem można nie wyprowadzać go w zmiennej `pos` i zmniejszyć ilość miejsca zajmowanego przez G-bufor o 16 bajtów na piksel. Jest to opisane na końcu tego podrozdziału.

Listing 27.2 przedstawia procedury, których zadaniem jest utworzenie i obsługa G-bufora. Podobnie jak w implementacji bufora akumulacji w rozdziale 25 użyjemy dwóch pozaekranowych buforów ramki; załącznikami pierwszego z nich są tekstury, z których składa się G-bufor, a drugi służy do wykonania końcowego obrazu[4]. Procedura `SetupDFBO` tworzy

[2] Ale gdyby materiałów do opisu obiektów w scenie było więcej niż 128, to już będą potrzebne dwa bajty.

[3] W pracy nad dużym projektem takie uproszczenia mogą powodować kłopoty, bo trudniej jest dopilnować spójności ewentualnych zmian we wszystkich szaderach. Ale tu chciałem skrócić listing.

[4] To mógłby być bufor ramki związany z oknem, ale otrzymany obraz możemy poddać dodatkowym przekształceniom. Do tego może być potrzebny obraz o szerokim zakresie dynamicznym, niedostępny na ekranie.

Listing 27.1. Szader fragmentów zapisujący informacje w G-buforze

GLSL

```
1: #version 450 core
2:
3: in FVertex {
4:     vec4      Position;
5:     vec2      TxtCoord;
6:     vec3      Normal;
7:     flat vec3 TNormal;
8:   } In;
9:
10: layout(location=0) out vec4 pos;
11: layout(location=1) out vec4 normal;
12: layout(location=2) out vec4 tnormal;
13: layout(location=3) out vec2 txtcoord;
14: layout(location=4) out int  matnum;
15:
16: uniform MatBlock { int mtn; } mat;
17:
18: void main ( void )
19: {
20:   pos = In.Position;
21:   normal = vec4 ( In.Normal, gl_FragCoord.z );
22:   tnormal = vec4 ( In.TNormal, gl_FragCoord.z );
23:   txtcoord = In.TxtCoord;
24:   matnum = mat.mtn;
25: } /*main*/
```

bufory ramki i ich załączniki, procedura `ResizeDFBO` zmienia wymiary załączników (trzeba ją wywołać po zmianie wymiarów okna), a procedura `DeleteDFBO` sprząta.

Struktura typu `DeferredFB` ma pola `width` i `height`, przechowujące wymiary obrazu w pikselach, oraz tablice `dfbo` i `dtxt` do przechowywania identyfikatorów buforów ramki i tekstur będących ich załącznikami. W tablicy `internf` są zapisane wewnętrzne formaty tych tekstur. Pierwsze pięć z nich wchodzi w skład G-bufora. Szósta tekstura jest buforem głębokości dla pierwszego etapu rysowania. W siódmej teksturze ma powstać obraz końcowy; tekstura ta jest jedynym załącznikiem drugiego bufora ramki. Ostatnie dwie tekstury będą użyte do obrazowania poświaty i modyfikowania oświetlenia rozproszonego.

Procedura `AllocDFBOTextures` w linii 19 rezerwuje identyfikatory tekstur, po czym w pętli dla każdej z nich rezerwuje pamięć. Pętla w liniach 26–27 przywiązuje załączniki do pierwszego bufora ramki. Po sprawdzeniu, że bufor ramki nadaje się do pracy, w linii 31 jest wywołana procedura `glDrawBuffers`, która określa numery załączników koloru, do których program z szaderem fragmentów z listingu 27.1 ma wyprowadzać dane[5].

Tekstura, w której ma powstać końcowy obraz, jest dołączana do drugiego bufora ramki w liniach 33–34. Wszystkie załączniki obu buforów ramki muszą być teksturami (nie bufo-

[5] Bez tego wywołania G-bufor nie zadziała.

Listing 27.2. Procedury obsługi G-bufora

C

```
#define DFB_TEXTURES 9

typedef struct {
    int    width, height;
    GLuint dfbo[2], dtxt[DFB_TEXTURES];
  } DeferredFB;

static const GLenum internf[DFB_TEXTURES] =
  { GL_RGBA32F, GL_RGBA32F, GL_RGBA32F, GL_RG32F, GL_R8I,
    GL_DEPTH_COMPONENT32F, GL_RGBA32F, GL_RGBA32F, GL_RGBA32F };

static void AllocDFBOTextures ( DeferredFB *dfb, int w, int h )
{
  static const GLenum buffers[6] = { GL_COLOR_ATTACHMENT0,
    GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT2, GL_COLOR_ATTACHMENT3,
    GL_COLOR_ATTACHMENT4, GL_DEPTH_ATTACHMENT };
  int i;

  glGenTextures ( DFB_TEXTURES, dfb->dtxt );
  for ( i = 0; i < DFB_TEXTURES; i++ ) {
    glBindTexture ( GL_TEXTURE_2D, dfb->dtxt[i] );
    glTexStorage2D ( GL_TEXTURE_2D, 1, internf[i], w, h );
  }
  glBindTexture ( GL_TEXTURE_2D, 0 );
  glBindFramebuffer ( GL_FRAMEBUFFER, dfb->dfbo[0] );
  for ( i = 0; i < 6; i++ )
    glFramebufferTexture ( GL_FRAMEBUFFER, buffers[i], dfb->dtxt[i], 0 );
  if ( glCheckFramebufferStatus ( GL_FRAMEBUFFER ) !=
       GL_FRAMEBUFFER_COMPLETE )
    ExitOnError ( "AllocDFBOTextures 0" );
  glDrawBuffers ( 5, buffers );
  glBindFramebuffer ( GL_FRAMEBUFFER, dfb->dfbo[1] );
  glFramebufferTexture ( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                         dfb->dtxt[6], 0 );
  if ( glCheckFramebufferStatus ( GL_FRAMEBUFFER ) !=
       GL_FRAMEBUFFER_COMPLETE )
    ExitOnError ( "AllocDFBOTextures 1" );
  dfb->width = w;  dfb->height = h;
  glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
} /*AllocDFBOTextures*/

void SetupDFBO ( DeferredFB *dfb, int w, int h )
{
  glGenFramebuffers ( 2, dfb->dfbo );
  AllocDFBOTextures ( dfb, w, h );
```

```
46:   ExitIfGLError ( "SetupDFBO" );
47: } /*SetupDFBO*/
48:
49: void ResizeDFBO ( DeferredFB *dfb, int w, int h )
50: {
51:   glDeleteTextures ( DFB_TEXTURES, dfb->dtxt );
52:   AllocDFBOTextures ( dfb, w, h );
53:   ExitIfGLError ( "ResizeDFBO" );
54: } /*ResizeDFBO*/
55:
56: void DeleteDFBO ( DeferredFB *dfb )
57: {
58:   glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
59:   glDeleteFramebuffers ( 2, dfb->dfbo );
60:   glDeleteTextures ( DFB_TEXTURES, dfb->dtxt );
61:   ExitIfGLError ( "DeleteDFBO" );
62: } /*DeleteDFBO*/
```

rami roboczymi), bo drugi etap rysowania będzie realizowany przez szader obliczeniowy, który będzie je widział jako obrazy. Drugi bufor ramki będzie potrzebny, aby gotowy obraz przesłać na ekran.

Listing 27.3 przedstawia przykładową procedurę rysowania przy użyciu G-bufora. Struktura typu `AppData` (z danymi aplikacji) zawiera pole `dfb`, które jest opisaną wyżej strukturą reprezentującą G-bufor. Po przygotowaniach w liniach 28–30 następuje wywołanie procedury `DrawSceneToShadows` (przedstawionej na listingu 26.18), której zadaniem jest znalezienie obszarów cienia dla włączonych w danej chwili źródeł światła. W liniach 32–34 następuje ustawienie wymiarów klatki i przesłanie do bloku `TransBl` macierzy określających rzutowanie dla końcowego obrazu. Pomocnicza procedura `GBufferBegin` wykonuje pierwszy etap rysowania. W linii 6 zostaje uaktywniony pozaekranowy bufor ramki, którego załączniki są G-buforem. Kasowanie bufora głębokości i tła jest wykonywane w liniach 7 i 8. Wszystkim tekselom załącznika numer 4, czyli tekstury `matnum`, zostaje przypisana wartość `-1`; jeśli dany teksel nie zmieni wartości, to piksel końcowego obrazu otrzyma kolor tła. W linii 9 jest wywoływana procedura, która kolejno rysuje wszystkie obiekty sceny. Jej drugi parametr (ukryta za znaczącą nazwą stała określona w aplikacji) powoduje używanie do tego programów zawierających szader fragmentów z listingu 27.1. Potem w liniach 10–14 elementy G-bufora są udostępniane jako obrazy szaderom, które mają wytworzyć końcowy obraz.

Drugi etap jest realizowany przez uaktywniany w linii 36 program z szaderem obliczeniowym. Każdy jego wątek oblicza kolor jednego piksela końcowego obrazu. Po uruchomieniu obliczeń i ich dokończeniu (tj. po powrocie z procedury wywołanej w linii 38) następuje wywołanie procedury `GBufferEnd`, która przesyła obraz na ekran za pomocą procedury `glBlitFramebuffer`.

Listing 27.4 przedstawia najważniejsze części szadera realizującego drugi etap opóźnionego cieniowania. Jego lokalne grupy robocze mają wymiary $1\times1\times1$; zadaniem wątku szadera jest obliczenie koloru jednego piksela.

Listing 27.3. Procedury rysowania w G-buforze

C

```
static void GBufferBegin ( AppData *ad )
{
  int i;
  const GLint mo[4] = {-1,-1,-1,-1};

  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, ad->dfb.dfbo[0] );
  glClear ( GL_DEPTH_BUFFER_BIT );
  glClearBufferiv ( GL_COLOR, 4, mo );
  DrawMyScene ( ad, ROPT_DEFERRED );
  for ( i = 0; i < 5; i++ )
    glBindImageTexture ( i, ad->dfb.dtxt[i], 0, GL_FALSE, 0,
                         GL_READ_ONLY, internf[i] );
  glBindImageTexture ( 5, ad->dfb.dtxt[6], 0, GL_FALSE, 0,
                       GL_WRITE_ONLY, internf[6] );
} /*GBufferBegin*/

static void GBufferEnd ( AppData *ad )
{
  glBindFramebuffer ( GL_READ_FRAMEBUFFER, ad->dfb.dfbo[1] );
  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
  glBlitFramebuffer ( 0, 0, ad->dfb.width, ad->dfb.height,
                      0, 0, ad->dfb.width, ad->dfb.height,
                      GL_COLOR_BUFFER_BIT, GL_NEAREST );
} /*GBufferEnd*/

void RedrawMyWorld1 ( AppData *ad )
{
  glEnable ( GL_DEPTH_CLAMP );
  glDepthFunc ( GL_LEQUAL );
  glEnable ( GL_DEPTH_TEST );
  DrawSceneToShadows ( ad );
  LoadViewport ( &ad->trans,
                 0, 0, ad->camera.win_width, ad->camera.win_height );
  LoadVPMatrix ( &ad->trans );
  GBufferBegin ( ad );
  glUseProgram ( ad->program_id[6] );  /* listing 27.4 */
  glDispatchCompute ( ad->dfb.width, ad->dfb.height, 1 );
  glMemoryBarrier ( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
  GBufferEnd ( ad );
  glUseProgram ( 0 );
  glFlush ();
  ExitIfGLError ( "RedrawMyWorld1" );
} /*RedrawMyWorld1*/
```

Listing 27.4. Prosty szader opóźnionego cieniowania

GLSL

```
#version 450 core

#define MAX_TEXTURES   4
#define MAX_MATERIALS 20
#define MAX_NLIGHTS    8

layout(local_size_x=1) in;

layout(rgba32f,binding=0) uniform image2D  pos;
layout(rgba32f,binding=1) uniform image2D  normal;
layout(rgba32f,binding=2) uniform image2D  tnormal;
layout(rg32f,binding=3)   uniform image2D  txtcoord;
layout(r8i,binding=4)     uniform iimage2D matnum;
layout(rgba32f,binding=5) uniform image2D  Out;

struct { vec3 Position, Normal, TNormal;  vec2 TxtCoord; } In;

layout(binding=0) uniform sampler2D tex[MAX_TEXTURES];

uniform TransBlock { .... } trb;
uniform LSBlock { .... } light;
uniform MatBlock { .... } mat;

Material mm;

void main ( void )
{
  ivec2 xy;
  vec3  Colour;
  int   mtn;

  xy = ivec2 ( gl_GlobalInvocationID.xy );
  if ( (mtn = imageLoad ( matnum, xy ).x) >= 0 ) {
    In.Position = imageLoad ( pos, xy );
    In.Normal = normalize ( imageLoad ( normal, xy ).xyz );
    In.TNormal = imageLoad ( tnormal, xy ).xyz;
    mm = mat.mat[mtn];
    if ( mm.txtnum >= 0 )
      mm.diffref = texture ( tex[mm.txtnum], imageLoad (txtcoord, xy).xy );
    Colour = LambertLighting ();
    imageStore ( Out, xy, vec4 (AGamma ( Colour ), 1.0) );
  }
  else
    imageStore ( Out, xy, vec4(0.0) );
} /*main*/
```

W liniach 9–13 są zadeklarowane zmienne dające dostęp do G-bufora. Zwróćmy uwagę na kwalifikatory dających ten dostęp zmiennych, deklarujące oprócz numeru punktu dowiązania każdego obrazu jego wewnętrzny format — ma on być zgodny z formatem podanym podczas tworzenia tekstury (listing 27.2, linie 9–10, 20–23 i 26–27).

Reprezentacja materiału (struktura `Material`) zawiera numer tekstury, zgodnie z opisem w p. 19.8.4 (listing 19.12). Zauważmy, że w opóźnionym cieniowaniu szader obliczający kolory pikseli musi mieć dostęp *jednocześnie do wszystkich* materiałów i tekstur.

Pierwsza instrukcja procedury `main` na podstawie numeru wątku w (dwuwymiarowej) globalnej grupie roboczej ustala współrzędne piksela, którego kolor szader ma obliczyć. W linii 33 z G-bufora jest odczytywany numer materiału obiektu widocznego w tym pikselu. Jeśli numer ten jest ujemny, to piksel ma mieć kolor tła (w tym przypadku czarny, przypisywany w linii 44). Jeśli numer jest nieujemny, to piksel należy do obrazu jakiegoś obiektu i wtedy w liniach 34–36 z G-bufora są odczytywane wektory $\boldsymbol{p}$, $\boldsymbol{n}$ i $\boldsymbol{m}$. Opis materiału do obliczeń oświetlenia jest przypisywany zmiennej `mm` w linii 37, przy czym, jeśli kolor materiału jest określony przez teksturę, to z G-bufora jest odczytywany wektor współrzędnych tekstury, po czym wartość odpowiedniej tekstury jest przypisywana polu `diffref` opisu materiału, reprezentującemu zdolność materiału do odbijania światła w sposób rozproszony. Ten szader oblicza kolor za pomocą pominiętej na listingu procedury `LambertLighting`, realizującej model lambertowski, który można oczywiście zastąpić dowolnym innym modelem. Wynik obliczeń jest zapisywany w obrazie wynikowym za pomocą procedury `imageStore`.

Zobaczmy teraz, jak można (i czy warto) zmniejszyć ilość miejsca zajmowanego przez G-bufor. Zadanie polega na odtworzeniu współrzędnych w układzie świata punktu $\boldsymbol{p}$ odpowiadającego fragmentowi na podstawie innych dostępnych danych. Razem z wektorem normalnym, w niezajętej przezeń czwartej składowej piksela w obrazie `normal`, podana jest głębokość fragmentu[6]. Jest to liczba $\zeta \in [0,1]$, na podstawie której możemy obliczyć współrzędną z w układzie kostki standardowej: $z = 2\zeta - 1$. Po znalezieniu pozostałych współrzędnych punktu w tym układzie możemy dokonać przejścia do układu świata.

Współrzędne x i y punktu w układzie kostki standardowej możemy obliczyć na podstawie wzorów (6.1), znając współrzędne ξ, η fragmentu w układzie okna. Na pozór są one numerami `x` i `y` wątku szadera w grupie roboczej. Ale w opisie układów współrzędnych w podrozdziale 6.1 jest pewna nieścisłość. Położenie i wymiary klatki w oknie określa się za pomocą liczb całkowitych — numeru kolumny i wiersza piksela w dolnym lewym narożniku klatki oraz jej szerokości i wysokości w pikselach. Stąd początki układów współrzędnych zilustrowanych na rysunku 6.1 znajdują się w środkach narożnych pikseli okna, a więc są przesunięte o połowę szerokości i wysokości piksela względem narożników okna. Aby otrzymać potrzebne tu współrzędne fragmentu w układzie, którego początek jest dolnym lewym narożnikiem *okna*, trzeba przyjąć $\xi = \mathrm{x} + 1/2$, $\eta = \mathrm{y} + 1/2$ i teraz już możemy obliczyć

$$x = 2(\xi - \xi_l)/w - 1, \quad y = 2(\eta - \eta_b)/h - 1.$$

[6]Głębokość jest też zapisana w buforze głębokości, do którego dostęp szaderowi opóźnionego cieniowania możemy łatwo zapewnić. Ale obrazy, w których są zapisane wektory normalne, muszą mieć piksele o czterech składowych, więc tu miejsca nie zaoszczędzimy.

Mając wektor $\boldsymbol{Q} = (x, y, z, 1)$ współrzędnych jednorodnych fragmentu w układzie kostki standardowej, pozostaje obliczyć wektor $\boldsymbol{P} = (PV)^{-1}\boldsymbol{Q}$ i podzielić jego pierwsze trzy współrzędne przez czwartą.

Listing 27.5 przedstawia procedurę obliczającą współrzędne punktu w układzie świata na podstawie współrzędnych (ξ, η) piksela (otrzymanych z numeru wątku) i głębokości ζ. Wartością parametru `q` jest wektor (ξ, η, ζ). Aby szader mógł wykonać opisane wyżej obliczenie, do bloku zmiennych jednolitych `TransBlock` zostały dodane dwa pola. Wartością pola `viewport` (nadawaną przez procedurę `LoadViewport`) jest wektor (ξ_l, η_b, w, h) opisujący klatkę. Pole `vpmi` zawiera odwrotność macierzy PV, przechowywanej w polu `vpm`.

Listing 27.5. Procedura odtwarzania współrzędnych punktu w układzie świata

GLSL

```
uniform TransBlock {
    mat4 mm, mmti, vm, pm, vpm, vpmi;
    vec4 eyepos;
    vec4 viewport;
  } trb;

vec3 CalcPosition ( vec3 q )
{
  vec4 p;

  p = trb.vpmi *
        vec4 ( 2.0*(q.x-trb.viewport.x)/trb.viewport.z-1.0,
               2.0*(q.y-trb.viewport.y)/trb.viewport.w-1.0,
               2.0*q.z-1.0, 1.0 );
  return p.xyz/p.w;
} /*CalcPosition*/
```

Ceną za uzyskaną oszczędność pamięci jest niedokładność odtworzenia punktu $\boldsymbol{p}$ spowodowana przez błędy zaokrągleń. O ile to na ogół nie ma wielkiego znaczenia w obliczeniach wektorów $\boldsymbol{v}$ (do obserwatora) i $\boldsymbol{l}_i$ (do źródeł światła), potrzebnych w modelu oświetlenia, o tyle błędy te mogą *bardzo* zaburzyć wyniki testowania, czy dany punkt jest w obszarze cienia. Przypomnijmy, że aby powierzchnia nie zasłaniała od światła własnych punktów, należy podczas znajdowania reprezentacji obszaru cienia wprowadzić odpowiednie poprawki głębokości (za pomocą procedury `glPolygonOffset`, zobacz s. 568). Jeśli współrzędne punktów w układzie świata są obliczane zgodnie z podanym tu opisem, to potrzebne poprawki są znacznie większe.

27.2. Obrazowanie poświaty

Wokół silnych źródeł światła w ciemnym otoczeniu, na przykład wokół płomienia świecy w oświetlonym przez nią pomieszczeniu lub wokół latarni albo reflektorów samochodu w nocnym krajobrazie, możemy zaobserwować poświatę będącą skutkiem rozpraszania

światła przez atmosferę. Dodanie takiego efektu do obrazu może istotnie podnieść jego atrakcyjność. Zobaczmy, jak to osiągnąć za pomocą G-bufora.

Załącznik `matnum` G-bufora zawiera, dla każdego piksela, numer materiału przedmiotu widocznego w tym pikselu. Źródło światła jest „zrobione" z materiału, który ma niezerową emisję, podaną w polach `emission0` i `emission1` struktury `Material` (listing 18.1). Możemy utworzyć dodatkowy obraz, którego piksele otrzymają wartość emisji materiału, a następnie „rozmyć" go i dodać do końcowego obrazu. Rozmycie zrealizujemy, filtrując obraz emisji za pomocą funkcji, której najważniejsze własności wypada teraz przypomnieć.

Funkcja rozkładu normalnego Gaussa o odchyleniu standardowym σ jest dana wzorem

$$\mathcal{N}_\sigma(x) \stackrel{\text{def}}{=} \frac{1}{\sigma\sqrt{2\pi}} \mathrm{e}^{-x^2/(2\sigma^2)}.$$

Funkcja ta jest parzysta, tj. $\mathcal{N}_\sigma(-x) = \mathcal{N}_\sigma(x)$ dla każdego x, przyjmuje maksymalną wartość dla $x = 0$ i ze wzrostem $|x|$ maleje do zera, przy czym w wielu zastosowaniach praktycznych (także w tu opisanym) dla $|x| > 3\sigma$ jej wartości są zaniedbywalnie małe. Całka z funkcji $\mathcal{N}_\sigma$ po całym zbiorze liczb rzeczywistych jest równa 1.

Filtr potrzebny do przetwarzania obrazów jest funkcją dwóch zmiennych; otrzymamy go ze wzoru

$$F_\sigma(x, y) = \mathcal{N}_\sigma(x)\mathcal{N}_\sigma(y) = \frac{1}{\sigma\sqrt{2\pi}} \mathcal{N}_\sigma(r), \text{ gdzie } r = \sqrt{x^2 + y^2}.$$

Mając obraz oryginalny p, chcemy otrzymać obraz przefiltrowany q określony wzorem

$$q(\xi, \eta) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} p(\xi - x, \eta - y) F_\sigma(x, y) \,\mathrm{d}x \,\mathrm{d}y.$$

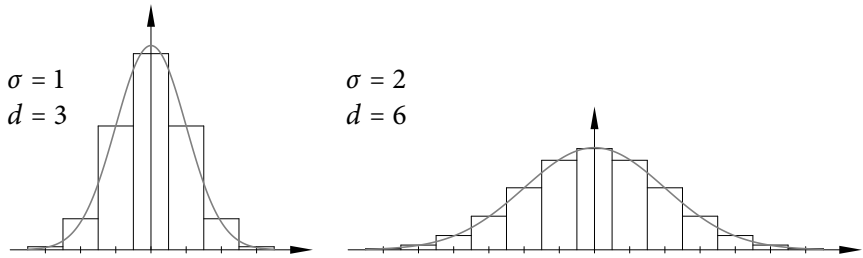
Formalne określenie tego przekształcenia, zwanego **splotem** funkcji p z F, wymaga rozszerzenia obrazów (funkcji określonych w prostokątnej klatce) na całą płaszczyznę; można uznać, że wartość funkcji p poza klatką jest zerem. Przetwarzając piksele, zastąpimy całkę **kwadraturą**, czyli kombinacją liniową wartości funkcji podcałkowej w skończenie wielu punktach, ale ją też trzeba obliczać umiejętnie.

Aby obliczyć kolor piksela (ξ, η) obrazu przefiltrowanego, trzeba zsumować wartości funkcji p w pikselach należących do dostatecznie dużego kwadratu, którego środkiem jest punkt (ξ, η), pomnożone przez wartości funkcji F_σ. Długość $2d + 1$ boku tego kwadratu odpowiada wielkości rozmycia pojedynczego piksela; może to być kilkadziesiąt lub ponad sto pikseli, co oznacza kilkaset do kilkunastu tysięcy składników do zsumowania. Na szczęście definicja funkcji F_σ jako iloczynu tensorowego umożliwia wykonanie obliczenia w dwóch etapach; najpierw obliczymy

$$\tilde{q}(\xi, \eta) = \sum_{i=-d}^{d} p(\xi - i, \eta) N_i \approx \int_{-\infty}^{\infty} p(\xi - x, \eta) \mathcal{N}_\sigma(x) \,\mathrm{d}x,$$

a potem

$$\hat{q}(\xi, \eta) = \sum_{j=-d}^{d} \tilde{q}(\xi, \eta - j) N_j \approx q(\xi, \eta),$$



Rysunek 27.1. Funkcje rozkładu normalnego i ich przybliżenia kawałkami stałe

przy użyciu współczynników N_i będących wartościami średnimi funkcji $\mathcal{N}_\sigma$ w przedziałach o długości 1:

$$N_i = \int_{i-1/2}^{i+1/2} \mathcal{N}_\sigma(x)\,\mathrm{d}x.$$

W każdym etapie wystarczy obliczyć dla każdego piksela sumę tylko $2d+1$ składników. Dla ustalonego σ można przyjąć liczbę $d = \lceil 3\sigma \rceil$, aby $2d+1$ przedziałów jednostkowych (odpowiadających pikselom) pokrywało przedział $[-3\sigma, +3\sigma]$. Rysunek 27.1 przedstawia wykresy funkcji $\mathcal{N}_1$ i $\mathcal{N}_2$ oraz ich przybliżenia, które w przedziałach jednostkowych mają stałe wartości N_i.[7]

Filtr oparty na funkcji Gaussa stwarza dodatkową możliwość: przefiltrowanie obrazu kolejno za pomocą funkcji F_{σ_1} i F_{σ_2} daje taki sam wynik[8] jak użycie filtru z odchyleniem standardowym $\sqrt{\sigma_1^2 + \sigma_2^2}$. W celu uzyskania odpowiednio dużego rozmycia można więc obraz przefiltrować kilkakrotnie, co jednak jest bardziej kosztowne niż jednorazowe użycie filtru z dużym odchyleniem standardowym.

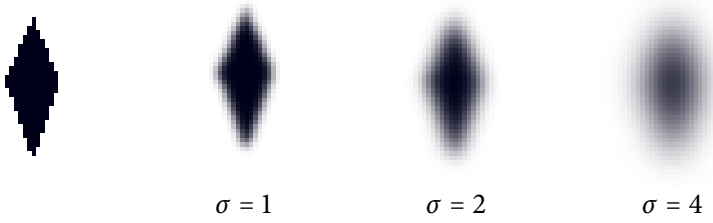
Wielkość obszaru poświaty powinna zależeć od wielkości źródła światła na obrazie, określonej przez jego położenie względem obserwatora oraz wielkość i kształt ostrosłupa widzenia i wielkość całego obrazu. Szader dokonujący rozmycia powinien dawać możliwość wybierania odchylenia standardowego, aby dobierać je do wielkości obrazu, ale można też manipulować liczbą powtórzeń filtrowania. Rysunek 27.2 przedstawia negatywy[9] obrazu płomienia świecy, którego model jest sztywną bryłą obrotową, i obrazów otrzymanych z niego po zastosowaniu filtrów z różnymi odchyleniami standardowymi.

Jest jeszcze jeden problem, który można rozwiązać za pomocą G-bufora. Dodanie do obrazu końcowego poświaty, czyli przefiltrowanego obrazu świecących obiektów sprawia, że poświata zawsze „wychodzi na pierwszy plan". Jeśli w obszarze poświaty znajdzie się obraz obiektu położonego bliżej niż źródło światła, to powstaje niepoprawny efekt, w którym

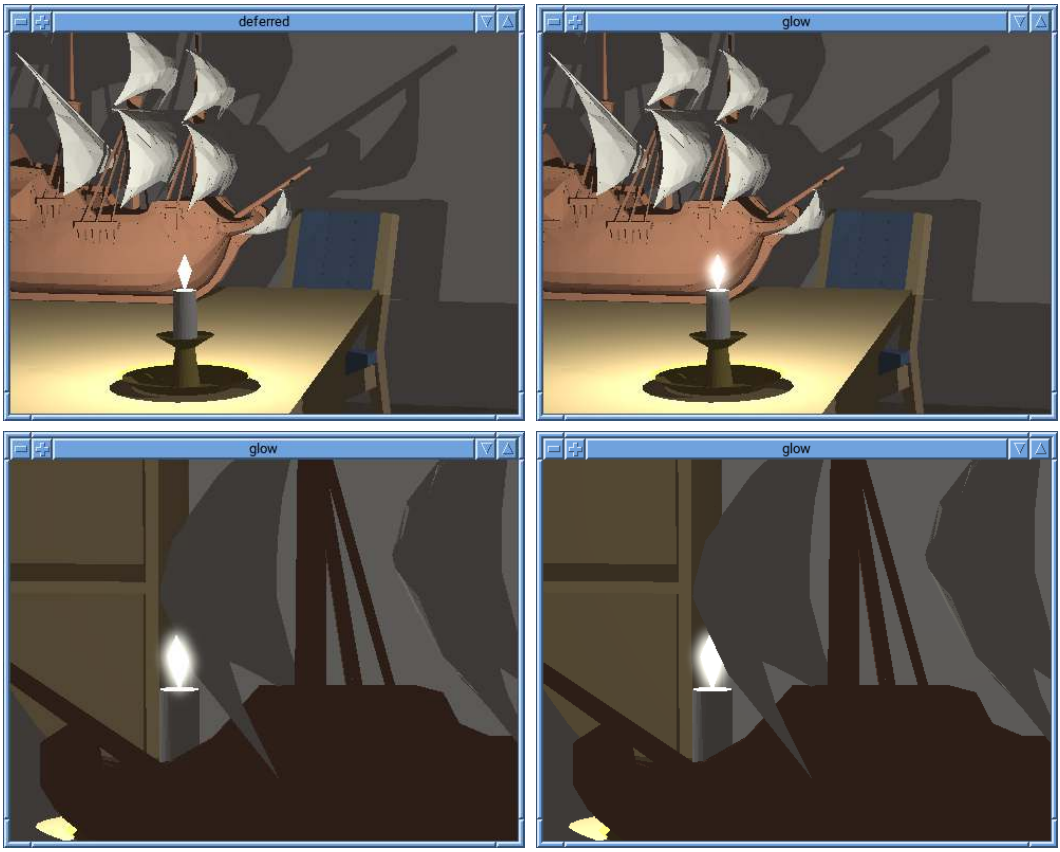
[7] Wartości średnie funkcji $\mathcal{N}_\sigma$ w wybranych $2d+1$ przedziałach obliczyłem numerycznie (za pomocą odpowiedniej kwadratury) i podzieliłem jeszcze przez ich sumę, aby otrzymać liczby, których suma jest równa 1.

[8] Mowa tu o filtrowaniu za pomocą całek. Przybliżające je kwadratury dają trochę inne wyniki, ale ich różnice są niezauważalne.

[9] Powód zamieszczenia negatywów jest podany na s. 155.



Rysunek 27.2. Obrazy płomienia świecy przed i po przefiltrowaniu



Rysunek 27.3. Obrazy z poświatą wokół płomienia, $\sigma = 8$

poświata występuje na tle tego obiektu (zobacz obrazki na dole rys. 27.3). Aby móc temu przeciwdziałać, dla każdego piksela zajętego przez obraz poświaty zapamiętamy minimalną głębokość fragmentu emitującego światło, którego obraz trafił do tego piksela. Głębokość tę będziemy przechowywać w składowej alfa pikseli.

Listing 27.6 przedstawia procedury rysowania sceny z poświatą wokół źródeł światła przy użyciu G-bufora i szaderów opisanych dalej. Procedura `RedrawMyWorld2` powstała przez zmianę nazwy i zastąpienie instrukcji w linii 36 na listingu 27.3 instrukcjami w liniach 27–30.

Listing 27.6. Procedury obrazowania poświaty

```C
void BlurLightEmission ( AppData *ad )
{
  glBindImageTexture ( 6, ad->dfb.dtxt[7], 0, GL_FALSE, 0,
                       GL_READ_WRITE, GL_RGBA32F );
  glBindImageTexture ( 7, ad->dfb.dtxt[8], 0, GL_FALSE, 0,
                       GL_READ_WRITE, GL_RGBA32F );
  glUseProgram ( ad->program_id[8] );  /* listing 27.7 */
  glUniform1i ( ad->prog8ngfloc, ad->ngf );
  glUniform1i ( ad->prog8passloc, 0 );
  glDispatchCompute ( ad->dfb.width, ad->dfb.height, 1 );
  glMemoryBarrier ( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
  glUniform1i ( ad->prog8passloc, 1 );
  glDispatchCompute ( ad->dfb.width, ad->dfb.height, 1 );
  glMemoryBarrier ( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
} /*BlurLightEmission*/

void RedrawMyWorld2 ( AppData *ad )
{
  glEnable ( GL_DEPTH_CLAMP );
  glDepthFunc ( GL_LEQUAL );
  glEnable ( GL_DEPTH_TEST );
  DrawSceneToShadows ( ad );
  LoadViewport ( &ad->trans,
                 0, 0, ad->camera.win_width, ad->camera.win_height );
  LoadVPMatrix ( &ad->trans );
  GBufferBegin ( ad );
  BlurLightEmission ( ad );
  glUseProgram ( ad->program_id[7] );  /* listing 27.8 */
  glUniform1f ( ad->prog7glowfctloc, ad->glowfct );
  glUniform1i ( ad->prog7obscglloc, ad->obscgl );
  glDispatchCompute ( ad->dfb.width, ad->dfb.height, 1 );
  glMemoryBarrier ( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
  GBufferEnd ( ad );
  glUseProgram ( 0 );
  glFlush ();
  ExitIfGLError ( "RedrawMyWorld2" );
} /*RedrawMyWorld2*/
```

Wywołana w linii 27 procedura w liniach 3–6 udostępnia szaderom obrazy w utworzonych razem z G-buforem teksturach dodatkowych, których wymiary w pikselach odpowiadają wielkości okna. Program uaktywniony w linii 7 składa się z szadera obliczeniowego przedstawionego na listingu 27.7. Wartość nadawana w linii 8 zmiennej jednolitej `ngf` wybiera współczynniki N_i odpowiadające odchyleniu standardowemu, które ma być przyjęte w obliczeniach. Z kolei w liniach 9 i 12 zmienna jednolita `pass` otrzymuje wartości wybierające etapy filtrowania; w razie potrzeby instrukcje w liniach 9–14 można umieścić w pętli, aby

powtórzyć filtrowanie kilka razy; wtedy zmiennej `pass` należy przypisywać kolejno wartości 0, 1, 2, 3, ...

Wykonywanie końcowego obrazu przez program szaderów uaktywniony w linii 28 i uruchomiony w linii 31 jest poprzedzone nadaniem wartości jego dwóm zmiennym jednolitym. Zmienna `glowfct` jest czynnikiem, przez który obraz poświaty zostaje pomnożony, a zmienna boolowska `obscgl` włącza lub wyłącza[10] zasłanianie poświaty przez obiekty położone bliżej obserwatora.

Na listingu 27.7 jest przedstawiony szader filtrujący. Obrazy, w których odbywa się filtrowanie, są dostępne za pośrednictwem zmiennych `glow0` i `glow1`. Procedura `GetInputColour` odczytuje z obrazu wejściowego piksel, którego współrzędne (utworzone z indeksu wątku szadera) są podane jako parametr. Jeśli współrzędne „wychodzą poza obraz", to w linii 59 następuje powrót.

Jeśli zmienna jednolita `pass` ma wartość 0, to z obrazu `matnum` jest odczytywany numer materiału obiektu widocznego w danym pikselu i z opisu materiału o tym numerze jest brana wartość pola `emission0`. Jeśli choć jedna ze składowych *r*, *g*, *b* emisji jest dodatnia, to jako czwarta współrzędna wektora przekazywanego jako wynik jest dołączana głębokość fragmentu odczytana z G-bufora. W przeciwnym razie przekazywana jest głębokość 1, odpowiadająca punktom tylnej ściany ostrosłupa widzenia.

Listing 27.7. Szader filtrujący poświatę

GLSL

```
1: #version 450 core
2:
3: #define MAX_MATERIALS 20
4:
5: layout(local_size_x=1) in;
6:
7: layout(rgba32f,binding=1) uniform image2D  normal;
8: layout(r8i,binding=4)     uniform iimage2D matnum;
9: layout(rgba32f,binding=6) uniform image2D  glow0;
10: layout(rgba32f,binding=7) uniform image2D  glow1;
11:
12: struct Material { vec4 emission0, emission1;  .... };
13: uniform MatBlock { int mtn;  Material mat[MAX_MATERIALS]; } mat;
14:
15: uniform int pass;
16: uniform int ngf;
17:
18: const float mgf[663] =
19:   { /* sigma = 1 */
20:     0.3831031644, 0.2418428568, 0.0606257426, 0.0059798184,
21:     /* sigma = sqrt(2) */
22:     0.2763541975, 0.2174365006, 0.1058829014, 0.0318889802, 0.0059334031,
```

[10] w celach badawczych

```
23:     0.0006811159,
24:     /* sigma = 2 */
25:     0.1976407387, .... /*   5 pominiętych liczb */, 0.0024055143,
26:     /* sigma = 2*sqrt(2) */
27:     0.1404261498, .... /*   8 pominiętych liczb */, 0.0009362786,
28:     /* sigma = 4 */
29:     0.0996536389, .... /*  11 pominiętych liczb */, 0.0011331269,
30:     /* sigma = 4*sqrt(2) */
31:     0.0705715323, .... /*  16 pominiętych liczb */, 0.0007809232,
32:     /* sigma = 8 */
33:     0.0499449655, .... /*  23 pominięte liczby  */, 0.0005580935,
34:     /* sigma = 8*sqrt(2) */
35:     0.0353313889, .... /*  33 pominięte liczby  */, 0.0003875479,
36:     /* sigma = 16 */
37:     0.0249906981, .... /*  47 pominiętych liczb */, 0.0002780284,
38:     /* sigma = 16*sqrt(2) */
39:     0.0176730978, .... /*  67 pominiętych liczb */, 0.0001934286,
40:     /* sigma = 32 */
41:     0.0124984914, .... /*  95 pominiętych liczb */, 0.0001388965,
42:     /* sigma = 32*sqrt(2) */
43:     0.0088379009, .... /* 135 pominiętych liczb */, 0.0000966758,
44:     /* sigma = 64 */
45:     0.0062498553, .... /* 191 pominiętych liczb */, 0.0000694360};
46:
47: const int mgfi[14] =
48:     { 0, 4, 10, 17, 27, 40, 58, 83, 118, 167, 236, 333, 470, 663 };
49:
50: ivec2 wh;
51:
52: vec4 GetInputColour ( ivec2 xy )
53: {
54:   int   mtn;
55:   float depth;
56:   vec3  em;
57:
58:   if ( xy.x < 0 || xy.x >= wh.x || xy.y < 0 || xy.y >= wh.y )
59:     return vec4 ( 0.0, 0.0, 0.0, 1.0 );
60:   if ( pass == 0 ) {
61:     depth = imageLoad ( normal, xy ).w;
62:     if ( (mtn = imageLoad ( matnum, xy ).x) < 0 )
63:       return vec4 ( 0.0, 0.0, 0.0, 1.0 );
64:     else {
65:       em = mat.mat[mtn].emission0.rgb;
66:       if ( em.r > 0.0 || em.g > 0.0 || em.b > 0.0 )
67:         return vec4 ( mat.mat[mtn].emission0.rgb, depth );
68:       else
69:         return vec4 ( 0.0, 0.0, 0.0, 1.0 );
```

```
70:     }
71:   }
72:   else if ( (pass & 0x01) == 0 )
73:     return imageLoad ( glow0, xy );
74:   else
75:     return imageLoad ( glow1, xy );
76: } /*GetInputColour*/
77:
78: void main ( void )
79: {
80:   ivec2 xy;
81:   vec4  Colour, c1, c2;
82:   int   i, j;
83:   float depth;
84:
85:   wh = imageSize ( matnum );
86:   xy = ivec2 ( gl_GlobalInvocationID.xy );
87:   Colour = GetInputColour ( xy );
88:   depth = Colour.w;
89:   Colour.rgb *= mgf[mgfi[ngf]];
90:   if ( (pass & 0x01) == 0 ) {
91:     for ( i = 1, j = mgfi[ngf]+1;  j < mgfi[ngf+1];  i++, j++ ) {
92:       c1 = GetInputColour ( xy + ivec2(i,0) );
93:       if ( c1.w < depth ) depth = c1.w;
94:       c2 = GetInputColour ( xy - ivec2(i,0) );
95:       if ( c2.w < depth ) depth = c2.w;
96:       Colour.rgb += mgf[j] * ( c1.rgb + c2.rgb );
97:     }
98:     Colour.a = depth;
99:     imageStore ( glow1, xy, Colour );
100:  }
101:  else {
102:    for ( i = 1, j = mgfi[ngf]+1;  j < mgfi[ngf+1];  i++, j++ ) {
103:      c1 = GetInputColour ( xy + ivec2(0,i) );
104:      if ( c1.w < depth ) depth = c1.w;
105:      c2 = GetInputColour ( xy - ivec2(0,i) );
106:      if ( c2.w < depth ) depth = c2.w;
107:      Colour.rgb += mgf[j] * ( c1.rgb + c2.rgb );
108:    }
109:    Colour.a = depth;
110:    imageStore ( glow0, xy, Colour );
111:  }
112: } /*main*/
```

Procedura `main` w liniach 85 i 86 ustala rozdzielczość przetwarzanego obrazu i współrzędne piksela w tym obrazie, a potem w linii 87 odczytuje piksel (kolor z głębokością) z obrazu wejściowego. Współczynniki $N_0, \ldots, N_d$ filtru odpowiadającego odchyleniu standar-

dowemu σ wybranemu przy użyciu zmiennej ngf są podane w tablicy mgf na miejscach mgfi[ngf], ..., mgfi[ngf+1]-1[11].

W linii 89 kolor odczytanego piksela jest mnożony przez współczynnik N_0, a potem, w zależności od tego, czy zmienna pass ma wartość parzystą, czy nie, są wykonywane instrukcje w liniach 91–99 albo 102–110. Z obrazu wejściowego są odczytywane piksele na prawo i na lewo albo z góry i z dołu piksela, którgo kolor należy obliczyć. W liniach 96 i 107 następuje obliczanie i sumowanie składników kwadratury. Dla wszystkich odczytanych pikseli znajdowana jest najmniejsza głębokość, która zostaje przypisana ostatniej składowej zmiennej Colour, po czym wynik obliczeń zostaje zapisany w obrazie glow1 albo glow0.

Listing 27.8. Procedura main szadera wykonującego obrazy z poświatą

GLSL

```
1: layout(rgba32f,binding=6) uniform image2D  glow0;
2:
3: uniform float glowfct;
4: uniform bool  obscgl;
5:
6: void main ( void )
7: {
8:   ivec2 xy;
9:   vec3  Colour;
10:  vec4  aux0, aux1;
11:  int   mtn;
12:
13:  xy = ivec2 ( gl_GlobalInvocationID.xy );
14:  aux0 = imageLoad ( glow0, xy );
15:  Colour = glowfct * aux0.rgb;
16:  if ( (mtn = imageLoad ( matnum, xy ).x) >= 0 ) {
17:    aux1 = imageLoad ( normal, xy );
18:    if ( obscgl && aux1.w < aux0.w )
19:      Colour = vec3(0.0);
20:    In.Normal = normalize ( aux1.xyz );
21:    In.Position = imageLoad ( pos, xy ).xyz;
22:    In.TNormal = imageLoad ( tnormal, xy ).xyz;
23:    m = mat.mat[mtn];
24:    if ( m.txtnum >= 0 )
25:      m.diffref = texture ( tex[m.txtnum], imageLoad ( txtcoord, xy ).xy );
26:    Colour += LambertLighting ();
27:  }
28:  imageStore ( Out, xy, vec4 (AGamma ( Colour ), 1.0) );
29: } /*main*/
```

Listing 27.8 przedstawia procedurę main szadera obliczeniowego wykonującego końcowy obraz z poświatą. W linii 15 zmienna Colour otrzymuje wartość poświaty; kolor oświetlonej

[11] Zmienna ngf musi mieć wartość od 0 do 12, chyba że tablice mgf, i mgfi zostaną wydłużone, aby zwiększyć wybór dostępnych odchyleń standardowych. Wynikający z parzystości funkcji $\mathcal{N}_\sigma$ fakt, że $N_i = N_{-i}$ dla każdego i, umożliwia skrócenie tablicy mgf i drobne przyspieszenie obliczeń.

powierzchni obliczony przez procedurę realizującą model oświetlenia zostanie dodany do poświaty, ale jeśli zmienna jednolita `obscgl` ma wartość `true` i głębokość fragmentu jest mniejsza niż głębokość obrazu poświaty (to jest badane w linii 18), to poświata jest zasłaniana, tj. zmiennej `Colour` jest przypisywany wektor zerowy. Jeśli numer materiału jest liczbą ujemną, to do końcowego obrazu (w linii 28) trafi sama poświata, zastępując doskonale czarny kolor tła.

Aby można było udoskonalić opisane wyżej rozwiązanie, trzeba się przyjrzeć jego wadom. Po pierwsze, jeśli źródło światła jest całkowicie zasłonięte przez pewien obiekt albo znajduje się tuż poza brzegiem obrazu, to poświata zniknie, choć jej część *powinna* być widoczna. Po drugie, poświaty wszystkich źródeł światła są mnożone przez ten sam czynnik (podany w zmiennej `glowfct`), a indywidualne dobieranie czynników (gdy jest więcej niż jedno źródło światła) okazuje się bardzo pożądane. Przy tym jeśli obiekty emitujące światło zajmują niewielki obszar na obrazie, szader filtrujący niepotrzebnie „miele" wiele zerowych pikseli.

Jasność poświaty otrzymanej przy ustalonym odchyleniu standardowym i wartości zmiennej `glowfct` bardzo zmienia się wraz ze zmianami wielkości obrazu źródła światła spowodowanymi zmianą położenia obserwatora, długości ogniskowej „obiektywu kamery" albo wymiarów okna. Jeśli jest więcej niż jedno źródło światła, to dobranie parametrów tak, aby otrzymać dobry obraz poświaty wokół każdego z nich, może być niewykonalne.

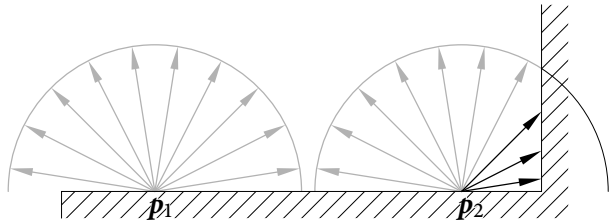
Można indywidualnie dobierać wartość emisji każdego źródła światła przed pierwszym etapem rysowania, ale lepszym rozwiązaniem jest osobne przetwarzanie obrazów źródeł światła z ich poświatami. Dla każdgo świecącego obiektu trzeba utworzyć teksturę, tzw. *billboard*, na której zmieści się obraz tego obiektu wraz z poświatą, oraz wykonać i przefiltrować te obrazy i pomnożyć je przez indywidualnie dobrane współczynniki. Realizację tego pomysłu pozostawiam Czytelnikom jako dość zaawansowane ćwiczenie.

27.3. Modyfikowanie oświetlenia światłem rozproszonym

Jeden ze składników modeli oświetlenia (np. Lamberta, Phonga i Blinna-Phonga) opisuje światło dochodzące z otoczenia. Założenie, że jego intensywność nie zależy od miejsca ani kierunku, prowadzi do otrzymania obrazów, na których części obiektów oświetlone tylko tym światłem wydają się płaskie. Pewną poprawę można uzyskać za pomocą opisanego w p. 18.4.3 oświetlenia hemisferycznego albo opisanych w następnym rozdziale metod oświetlenia przez obraz otoczenia (zobacz p. 28.3.2 i 28.4.3). Jeszcze inne efekty daje opisany niżej sposób modyfikowania przy użyciu G-bufora światła dochodzącego z otoczenia uwzględniający obecność innych powierzchni w pobliżu oświetlonego punktu. Technika ta jest nazywana **zasłanianiem otoczenia w przestrzeni obrazu** (*screen space ambient occlusion*, w skrócie *SSAO*).

Punkt nieprzezroczystej powierzchni odbija światło dochodzące z wszystkich kierunków strony, po której znajduje się obserwator. Jednostkowe wektory tych kierunków tworzą półsferę. Po ustaleniu największej odległości obiektów, których wpływ na oświetlenie danego punktu z otoczenia zamierzamy uwzględnić, możemy obliczyć (w przybliżeniu), jaka część półsfery kierunków jest zajęta przez te obiekty. Dzięki temu możemy zmodyfikować oświetlenie danego punktu przez otoczenie, odpowiednio osłabiając światło dochodzące z większej

odległości i uwzględniając światło odbite przez obiekty znajdujące się bliżej. Otrzymany efekt będzie najbardziej widoczny w kątach pomieszczeń i w różnych zakamarkach.



Rysunek 27.4. Zasłanianie otoczenia

Rysunek 27.4 przedstawia przekrój przez mające wspólną krawędź ściany nieprzezroczystego obiektu i przez półsfery o promieniu R wyznaczające kierunki, z których dochodzi światło do dwóch punktów na jednej z tych ścian. Przyjmiemy, że jeśli w odległości R lub mniejszej od danego punktu nie ma żadnych innych obiektów, to do tego punktu dochodzi światło, którego *całkowita* intensywność we wzorach (10.1), (18.1) i (18.2) jest sumą składników I_i^{amb} pochodzących od poszczególnych punktowych źródeł światła.

Takim punktem jest punkt $\boldsymbol{p}_1$ na rysunku, dlatego w końcowym obliczeniu jego oświetlenia **czynnik osłabienia** (*occlusion factor*), przez który należy pomnożyć całkowitą intensywność światła rozproszonego w otoczeniu (sumę składników I_i^{amb}), jest równy 1. Czynnik osłabienia dla punktu $\boldsymbol{p}_2$ powinien być mniejszy, bo w mniejszej niż R odległości od niego leży część drugiej ściany. Przyjmiemy za to, że z tej części do punktu $\boldsymbol{p}_2$ dochodzi światło odbite od niej, przy czym uwzględnimy tylko światło, które przed odbiciem dotarło bezpośrednio od punktowych źródeł światła. Jeśli ta druga ściana ma inny kolor niż pierwsza i jest silnie oświetlona światłem padającym na nią prawie prostopadle, to położone w jej pobliżu punkty pierwszej ściany (takie jak $\boldsymbol{p}_2$) są zauważalnie oświetlone rozproszonym światłem o kolorze drugiej ściany. Jest to zjawisko **przeciekania koloru**, mające w języku angielskim nieco makabryczną nazwę *color bleeding*.

Obliczenie czynnika osłabienia i oświetlenia światłem odbitym przez powierzchnie w pobliżu danego punktu opiera się na pojęciach radiometrycznych przedstawionych w następnym rozdziale i na założeniu, że światło jest odbijane od powierzchni w sposób lambertowski. Czytelników chcących zrozumieć podstawy opisanego niżej algorytmu zachęcam do przeczytania podrozdziałów 28.1–28.3, a tu podaję tylko niezbędne wiadomości w skrócie.

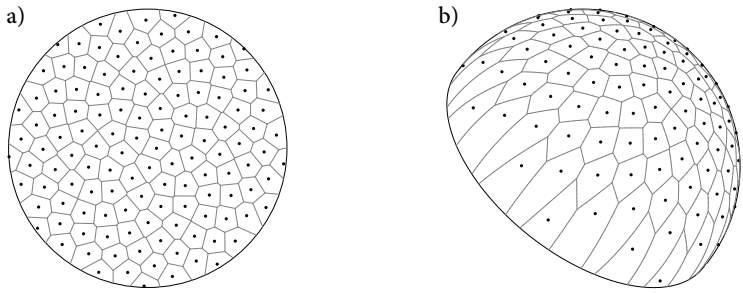
Fizyczna wielkość, która przekłada się na jasność obserwowanego punktu $\boldsymbol{p}$ powierzchni jest nazywana radiancją i dla powierzchni lambertowskiej nie zależy od kierunku ani odległości do obserwatora. Radiancja światła odbitego od powierzchni lambertowskiej jest iloczynem tzw. dwukierunkowej funkcji odbicia światła (oznaczanej skrótem BRDF) i całkowitej irradiancji światła padającego na powierzchnię w punkcie $\boldsymbol{p}$. Irradiancja jest to (mierzona w watach na metr kwadratowy) gęstość mocy. Jest ona całką po półsferze $S_{\boldsymbol{n}+}$ składającej się z wszystkich wektorów jednostkowych $\boldsymbol{l}$, które tworzą z wektorem normal-

nym powierzchni $\boldsymbol{n}$ kąt ostry[12]; funkcja podcałkowa opisuje radiancję światła dochodzącego do punktu $\boldsymbol{p}$ z kierunku $\boldsymbol{l}$ pomnożoną przez kosinus kąta padania światła (tj. kąta między wektorami $\boldsymbol{l}$ a $\boldsymbol{n}$).

Dwukierunkowa funkcja odbicia zależy od długości fali świetlnej i we wzorze (10.1) jest określona przez wektor $\boldsymbol{a}$; jest on wartością pola `diffref` w opisie materiału. Jego współrzędne r, g, b, należące do przedziału $[0,1)$, opisują ułamki mocy światła czerwonego, zielonego i niebieskiego, które jest odbijane. Wartości dwukierunkowej funkcji odbicia dla tych trzech kolorów otrzymamy, dzieląc wektor $\boldsymbol{a}$ przez liczbę π.

Półsferę $S_{\boldsymbol{n}+}$ o środku w punkcie $\boldsymbol{p}$ podzielimy na sektory. Uznamy, że wektory jednostkowe $\boldsymbol{l}$, w których kierunku w odległości od $\boldsymbol{p}$ mniejszej lub równej R nie ma innych obiektów, należą do sektora, z którego dochodzi światło rozproszone w scenie, a jeśli punkt $\boldsymbol{p}+r\boldsymbol{l}$, gdzie $0 < r \leqslant R$, jest punktem pewnej powierzchni sceny, to z kierunku $\boldsymbol{l}$ do punktu $\boldsymbol{p}$ zamiast światła rozproszonego dochodzi światło odbite od tej powierzchni.

Całki po sektorach przybliżymy kwadraturami. W tym celu podzielimy półsferę $S_{\boldsymbol{n}+}$ na N kawałków, w każdym z nich wybierzemy jeden punkt (wektor $\boldsymbol{l}_i$), wartość funkcji podcałkowej w tym punkcie pomnożymy przez pole kawałka i zsumujemy iloczyny. W ten sposób wszystkie punkty kawałka zawierającego punkt $\boldsymbol{l}_i$ zaliczymy do tego samego sektora.



Rysunek 27.5. Sposób konstruowania kwadratury w kole

Z różnych możliwych sposobów podziału półsfery wypróbowałem i polecam taki, w którym w kole mającym z nią wspólny brzeg zostały rozmieszczone punkty $\boldsymbol{p}_i$, a następnie wybrane takie wektory $\boldsymbol{l}_i$, aby punkt $\boldsymbol{p}_i$ był rzutem prostopadłym punktu $\boldsymbol{p}+\boldsymbol{l}_i$ na płaszczyznę koła. Do tego służy układ współrzędnych o początku w środku koła (tj. w punkcie $\boldsymbol{p}$), taki że koło leży w płaszczyźnie xy.[13] W tym układzie punkt $\boldsymbol{p}_i$ ma współrzędne

$$x_i = r_i \cos\varphi_i, \quad y_i = r_i \sin\varphi_i, \quad i = 0, \ldots, N-1, \tag{27.1}$$

gdzie $r_i = \sqrt{(2i+1)/(2N)}$, $\varphi_i = (i+\frac{1}{2})2\pi\tau^2$, $\tau = (\sqrt{5}-1)/2$ (porównaj z (25.3)). Wzór ten zapewnia rozmieszczenie punktów w kole jednostkowym z równomierną gęstością. Możemy podzielić półsferę na takie kawałki (rys. 27.5b), których rzuty prostopadłe na płaszczyznę koła

[12] Tu jest przyjęte założenie, że kąt między wektorem $\boldsymbol{n}$ a wektorem $\boldsymbol{v}$ mającym kierunek do obserwatora jest ostry, a więc zwrot wektora $\boldsymbol{n}$ wskazuje widoczną stronę powierzchni.

[13] Zauważmy, że układ współrzędnych użyty w konstrukcji punktów $\boldsymbol{p}_i$ nie jest określony jednoznacznie — parę jego osi x i y możemy dowolnie obracać wokół osi z, której wersorem jest wektor $\boldsymbol{n}$. To nie szkodzi.

są **regionami Woronoja**, tj. obszarami składającymi się z punktów koła położonych najbliżej poszczególnych punktów $\boldsymbol{p}_i$ (rys. 27.5a). Opisany sposób rozmieszczania punktów $\boldsymbol{p}_i$ ma tę zaletę, że ich regiony Woronoja mają małe średnice i *w przybliżeniu* to samo pole. Przyjmiemy hipotezę, że *istnieje* podział koła na kawałki o małych średnicach i *jednakowym* polu, z których każdy zawiera jeden punkt $\boldsymbol{p}_i$. Podział sfery, na którym opiera się kwadratura, odpowiada temu hipotetycznemu podziałowi koła[14].

Uzasadnieniem takiego podziału półsfery jest fakt, że funkcja podcałkowa jest iloczynem radiancji i kosinusa kąta między wektorami $\boldsymbol{l}$ a $\boldsymbol{n}$. W dobrym przybliżeniu (tym lepszym, im drobniejszy jest podział) iloczyn pola i-tego kawałka półsfery i kosinusa kąta między wektorami $\boldsymbol{l}_i$ a $\boldsymbol{n}$ jest polem i-tego kawałka koła. Jeśli wszystkie kawałki koła mają to samo pole, to gdy k jest liczbą wektorów $\boldsymbol{l}_i$ należących do sektora, z którego do punktu $\boldsymbol{p}$ dochodzi światło rozproszone w otoczeniu (tj. takich, że w kierunku $\boldsymbol{l}_i$ do odległości R od punktu $\boldsymbol{p}$ nie ma powierzchni zasłaniających światło z otoczenia), możemy przyjąć czynnik osłabienia (światła rozproszonego) równy k/N.

Podobnie, całkując irradiancję światła dochodzącego do punktu $\boldsymbol{p}$ z powierzchni znajdujących się w mniejszej niż R odległości od niego, zamiast mnożyć radiancję z kierunku $\boldsymbol{l}_i$ przez kosinus kąta między wektorami $\boldsymbol{l}_i$ a $\boldsymbol{n}$ i pole kawałka półsfery, możemy od razu pomnożyć radiancję przez pole odpowiedniego kawałka koła — a ponieważ te kawałki mają pola jednakowe, możemy zsumować radiancję z wszystkich kierunków $\boldsymbol{l}_i$ i przez liczbę $1/N$ pomnożyć sumę.

Aby otrzymać wektor $\boldsymbol{l}_i$ w układzie współrzędnych świata, w którym prowadzimy obliczenia oświetlenia, mając współrzędne punktu $\boldsymbol{p}_i$ w opisanym wyżej układzie, należy wektor $\tilde{\boldsymbol{l}}_i = (x_i, y_i, z_i)$ ze współrzędną $z_i = \sqrt{1 - x_i^2 - y_i^2}$ przekształcić do układu świata. Użyte przekształcenie musi być izometrią, na przykład odbiciem symetrycznym danym wzorem

$$\boldsymbol{l} = H\tilde{\boldsymbol{l}} = \pm(\tilde{\boldsymbol{l}} - \boldsymbol{w}\gamma\langle \boldsymbol{w}, \tilde{\boldsymbol{l}}\rangle), \tag{27.2}$$

w którym $\boldsymbol{w} = \boldsymbol{n} \pm (0, 0, 1)$, $\gamma = 2/\langle \boldsymbol{w}, \boldsymbol{w}\rangle$, przy czym znaki dobieramy tak, aby otrzymać *dłuższy* wektor $\boldsymbol{w}$ i aby iloczyn skalarny wektora $\boldsymbol{l}$ z wektorem $\boldsymbol{n}$ był dodatni.

Listing 27.9 przedstawia procedurę, która wykonuje obraz z oświetleniem obliczonym metodą SSAO. Początkowe instrukcje, identyczne jak na listingach 27.3 i 27.6, mają za zadanie zapisać w G-buforze potrzebne informacje. W linii 13 jest wywoływana opisana w poprzednim podrozdziale procedura wytwarzająca obraz poświaty. Przygotowanie obliczeń następuje w liniach 14–21. Opisany dalej szader obliczeniowy potrzebuje dostępu do jeszcze dwóch tekstur. Pierwsza z nich jest miejscem powstawania końcowego obrazu, a druga jest pomocniczą teksturą wykorzystywaną przez szader tworzący obraz poświaty, dla której teraz znajdziemy dodatkowe zastosowanie. Szader będzie zapisywał wyniki obliczeń w tekselach tych tekstur (za pośrednictwem zmiennych typu `image2D`) i czytał wyniki interpolacji tekseli (do których dostęp umożliwiają zmienne typu `sampler2D`).

[14] Pola regionów sąsiadujących z brzegiem koła mają większe odchyłki od średniego pola, ale jest ich niewiele. Podział koła na obszary o małej średnicy i identycznych polach π/N możemy otrzymać, modyfikując diagram Woronoja, przy czym diagram wystarczy utworzyć i zmodyfikować tylko w wyobraźni.

Listing 27.9. Procedura wykonywania obrazu ze zmodyfikowanym oświetleniem z otoczenia

C

```
1: void RedrawMyWorld3 ( AppData *ad )
2: {
3:   int i;
4:
5:   glEnable ( GL_DEPTH_CLAMP );
6:   glDepthFunc ( GL_LEQUAL );
7:   glEnable ( GL_DEPTH_TEST );
8:   DrawSceneToShadows ( ad );
9:   LoadViewport ( &ad->trans,
10:                 0, 0, ad->camera.win_width, ad->camera.win_height );
11:  LoadVPMatrix ( &ad->trans );
12:  GBufferBegin ( ad );
13:  BlurLightEmission ( ad );
14:  glActiveTexture ( GL_TEXTURE0+MAX_TEXTURES+MAX_NLIGHTS );
15:  glBindTexture ( GL_TEXTURE_2D, ad->dfb.dtxt[1] );
16:  glActiveTexture ( GL_TEXTURE0+MAX_TEXTURES+MAX_NLIGHTS+1 );
17:  glBindTexture ( GL_TEXTURE_2D, ad->dfb.dtxt[6] );
18:  glUseProgram ( ad->program_id[9] );  /* listingi 27.10, 27.11 i 27.12 */
19:  glUniform1f ( ad->prog9glowfctloc, ad->glowfct );
20:  glUniform1i ( ad->prog9obscglloc, ad->obscgl );
21:  glUniform1f ( ad->prog9ssao_radiusloc, ad->ssao_radius );
22:  for ( i = 0; i < 3; i++ ) {
23:    glUniform1i ( ad->prog9stageloc, i );
24:    glDispatchCompute ( ad->dfb.width, ad->dfb.height, 1 );
25:    glMemoryBarrier ( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
26:  }
27:  GBufferEnd ( ad );
28:  glUseProgram ( 0 );
29:  glFlush ();
30:  ExitIfGLError ( "RedrawMyWorld3" );
31: } /*RedrawMyWorld3*/
```

Instrukcje w liniach 19 i 20 nadają wartości zmiennym jednolitym używanym w obliczeniach poświaty, a w linii 21 jest wprowadzany promień R. Szader obliczeniowy zbudowany z programu uaktywnionego w linii 18 wykonuje obliczenie w trzech etapach, następujących w kolejnych przebiegach pętli w liniach 22–26. W linii 23 numer kolejnego etapu jest przypisywany zmiennej jednolitej `stage`, a potem wywoływany jest program szaderów, w liczbie wątków równej liczbie pikseli obrazu. Końcowe instrukcje są takie same jak w procedurach opisanych wcześniej.

Listingi 27.10–27.12 przedstawiają części szadera obliczeniowego wykonującego końcowy obraz; nie ma na nich procedur obliczających bezpośrednie oświetlenie przez punktowe źródła światła z uwzględnieniem cieni, opisanych w poprzednich rozdziałach. Z niepokazanej procedury `LambertLighting` została usunięta instrukcja dodająca wkład danego źródła w światło rozproszone w otoczeniu (listing 26.23, linia 62).

Listing 27.10. Deklaracje zmiennych szadera wykonującego końcowy obraz

GLSL

```
1: #version 450 core
2:
3: #define MAX_TEXTURES   4
4: #define MAX_MATERIALS 20
5: #define MAX_NLIGHTS    8
6:
7: #define NSSAO        128
8: #define PI   3.141592653
9: #define DPHI 2.399963229
10:
11: layout(local_size_x=1) in;
12:
13: layout(rgba32f,binding=0) uniform image2D  pos;
14: layout(rgba32f,binding=1) uniform image2D  normal;
15: layout(rgba32f,binding=2) uniform image2D  tnormal;
16: layout(rg32f,binding=3)   uniform image2D  txtcoord;
17: layout(r8i,binding=4)     uniform iimage2D matnum;
18: layout(rgba32f,binding=6) uniform image2D  glow;
19: layout(rgba32f,binding=7) uniform image2D  ambient;
20: layout(rgba32f,binding=5) uniform image2D  Out;
21: layout(binding=MAX_TEXTURES+MAX_NLIGHTS) uniform sampler2D normaltxt;
22: layout(binding=MAX_TEXTURES+MAX_NLIGHTS+1) uniform sampler2D outtxt;
23:
24: layout(binding=0) uniform sampler2D tex[MAX_TEXTURES];
25:
26: struct { vec3 Position, Normal, TNormal;  vec2 TxtCoord; } In;
27:
28: uniform TransBlock { .... } trb;
29: struct LSPar { .... };
30: uniform LSBlock { .... } light;
31: struct Material { .... };
32: uniform MatBlock { .... } mat;
33:
34: uniform float glowfct = 2.0;
35: uniform float emfct = 0.1;
36: uniform bool  obscgl = true;
37: uniform float ssao_radius = 0.15;
38: uniform int   stage;
39:
40: Material mm;
```

W liniach 7–9 na listingu 27.10 są nowe makrodefinicje; NSSAO określa liczbę N wektorów $\boldsymbol{l}_i$, liczby π przedstawiać nie trzeba, a nazwą DPHI jest oznaczona liczba $2\pi\tau^2$.

Elementy G-bufora są udostępnione przez zmienne zadeklarowane w liniach 13–17. W linii 18 jest zapewniony dostęp do obrazu poświaty, a deklaracja w linii 19 daje dostęp do ob-

razu pomocniczego, który przydał się do filtrowania obrazu poświaty, a teraz będzie użyty do przechowania obliczonej w drugim etapie irradiancji. Deklaracje w liniach 21 i 22 dają dostęp do tekstur, z których pierwsza jest elementem G-bufora przechowującym wektor normalny i głębokość, a druga jest obrazem końcowym. Obrazy tych tekstur są jednocześnie dostępne za pomocą zmiennych `normal` i `Out`; powód wprowadzenia tego podwójnego dostępu jest opisany dalej.

Wartością zmiennej `ssao_radius` (linia 37) jest promień R półkuli, w której poszukiwane są powierzchnie zasłaniające otoczenie oświetlanego punktu. Zmienna jednolita `stage` (linia 38) wybiera jeden z trzech etapów obliczenia.

Listing 27.11 przedstawia procedurę `main` szadera; jej zadaniem jest obliczenie koloru jednego piksela, o współrzędnych ustalonych przez instrukcję w linii 8. W linii 9 z G-bufora jest odczytywany numer materiału, a potem wykonywane są instrukcje realizujące bieżący etap. Jeśli zmienna `stage` ma wartość 0, to obliczenie jest wykonywane podobnie jak przez procedurę z listingu 27.8, z pominięciem instrukcji dodających poświatę i z pominięciem światła rozproszonego. Wynik trafia do miejsca, w którym powstanie (w ostatnim etapie) końcowy obraz do przesłania do okna. W tym etapie piksele są zapisywane *bez* korekcji gamma.

Listing 27.11. Procedura `main` szadera wykonującego końcowy obraz

GLSL

```
1: void main ( void )
2: {
3:   ivec2 xy;
4:   vec3  Colour, diffref;
5:   vec4  g;
6:   int   mtn;
7:
8:   xy = ivec2 ( gl_GlobalInvocationID.xy );
9:   mtn = imageLoad ( matnum, xy ).x;
10:  switch ( stage ) {
11: case 0:
12:    if ( mtn >= 0 ) {
13:      In.Position = imageLoad ( pos, xy ).xyz;
14:      In.Normal = normalize ( imageLoad ( normal, xy ).xyz );
15:      In.TNormal = imageLoad ( tnormal, xy ).xyz;
16:      mm = mat.mat[mtn];
17:      if ( mm.txtnum >= 0 )
18:        mm.diffref = texture ( tex[m.txtnum], imageLoad (txtcoord, xy).xy );
19:      Colour = LambertLighting ();
20:    }
21:    else
22:      Colour = vec3(0.0);
23:    imageStore ( Out, xy, vec4(Colour, 1.0) );
24:    return;
25: case 1:
26:    if ( mtn >= 0 ) {
```

```
27:       In.Normal = normalize ( imageLoad ( normal, xy ).xyz );
28:       In.Position = imageLoad ( pos, xy ).xyz;
29:       Colour = AmbientLight ( xy );
30:     }
31:     else
32:       Colour = vec3(0.0);
33:     imageStore ( ambient, xy, vec4(Colour, 1.0) );
34:     return;
35: case 2:
36:     g = imageLoad ( glow, xy );
37:     if ( obscgl && imageLoad ( normal, xy ).w < g.w )
38:       Colour = vec3(0.0);
39:     else
40:       Colour = glowfct * g.rgb;
41:     Colour += imageLoad ( Out, xy ).rgb;
42:     if ( mtn >= 0 ) {
43:       if ( mat.mat[mtn].txtnum >= 0 )
44:         diffref = texture ( tex[mat.mat[mtn].txtnum],
45:                             imageLoad ( txtcoord, xy ).xy ).rgb;
46:       else
47:         diffref = mat.mat[mtn].diffref.rgb;
48:       Colour += diffref * imageLoad ( ambient, xy ).rgb;
49:     }
50:     imageStore ( Out, xy, vec4(AGamma ( Colour ), 1.0) );
51:     return;
52:   }
53: } /*main*/
```

Drugi etap realizują instrukcje w liniach 26–33. Jeśli piksel należy do obrazu jakiegoś obiektu, to z G-bufora są odczytywane wektory współrzędnych położenia $\boldsymbol{p}$ i wektora normalnego $\boldsymbol{n}$ w układzie świata. Procedura `AmbientLight`, przedstawiona na listingu 27.12, oblicza irradiancję światła rozproszonego oświetlającego dany punkt.

W trzecim etapie (linie 36–50) szader oblicza końcowy kolor piksela. W linii 36 jest odczytywana poświata, która następnie jest poddawana testowi głębokości, podobnie jak w linii 18 na listingu 27.8. W liniach 43–47 jest ustalany kolor materiału, tj. wektor $\boldsymbol{a}$ opisujący jego zdolność odbijania światła. W linii 48 współrzędne tego wektora są mnożone przez przeskalowaną o czynnik $1/\pi$ irradiancję światła rozproszonego, odczytaną z obrazu `ambient`, w którym została zapisana w poprzednim etapie. Obliczony kolor piksela jest zapamiętywany w obrazie `Out`, tym razem po dokonaniu korekcji gamma.

Irradiancja światła rozproszonego jest obliczana przez procedury na listingu 27.12. Obliczenia geometryczne są wykonywane w dwóch układach współrzędnych: świata i kostki jednostkowej, która powstaje przez zmniejszenie o połowę i przesunięcie kostki standardowej (zobacz s. 553). Przejścia między tymi układami realizują funkcje `Project` i `Unproject` (linie 1–7 i 9–15). Argumentem pierwszej z nich jest wektor współrzędnych punktu w układzie świata, a wartością wektor współrzędnych tego punktu w układzie kostki jednostkowej, druga funkcja dokonuje przejścia w przeciwną stronę.

Listing 27.12. Procedury realizujące zasłanianie otoczenia

GLSL

```
vec3 Project ( vec3 p )
{
  vec4 q;

  q = trb.vpm * vec4 ( p, 1.0 );
  return 0.5*(q.xyz/q.w + vec3(1.0,1.0,1.0));
} /*Project*/

vec3 Unproject ( vec3 q )
{
  vec4 p;

  p = trb.vpmi * vec4 ( q - vec3(0.5,0.5,0.5), 0.5 );
  return p.xyz/p.w;
} /*Unproject*/

vec3 SSAOColour ( ivec2 xy, vec3 amb )
{
  vec3  l, w, p, q, bleeding;
  int   i, k;
  float phi, r, gamma, z;

  bleeding = vec3(0.0);
  w = In.Normal;
  w.z += In.Normal.z > 0.0 ? 1.0 : -1.0;
  gamma = 2.0 / dot ( w, w );
  for ( i = 1, k = 0, phi = 0.5*DPHI;  i < 2*NSSAO;  i += 2, phi += DPHI ) {
    r = sqrt ( float(i)/float(2*NSSAO) );
    l = vec3 ( r*cos ( phi ), r*sin ( phi ),
               sqrt ( float(2*NSSAO-i)/float(2*NSSAO) ) );
    l -= w*(gamma*dot ( w, l ));
    if ( In.Normal.z > 0.0 ) l = -l;
    p = In.Position + ssao_radius*l;
    q = Project ( p );
    if ( q.x < 0.0 || q.x > 1.0 || q.y < 0.0 || q.y > 1.0 )
      k ++;
    else {
      z = texture ( normaltxt, q.xy ).w;
      if ( q.z < z )
        k ++;
      else {
        l = Unproject ( vec3 ( q.x, q.y, z ) ) - In.Position;
        if ( length ( l ) > 1.5*ssao_radius )
          k ++;
        else
```

```
46:           bleeding += texture ( outtxt, xy ).rgb;
47:       }
48:     }
49:   }
50:   return (float(k)*amb + bleeding/PI)/float(NSSAO);
51: } /*SSAOColour*/
52:
53: vec3 AmbientLight ( ivec2 xy )
54: {
55:   vec3 amb;
56:   uint i, mask;
57:
58:   amb = vec3(0.0);
59:   for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
60:     if ( (light.mask & mask) != 0 )
61:       amb += light.ls[i].ambient;
62:   return SSAOColour ( xy, amb );
63: } /*AmbientLight*/
```

Procedura `AmbientLight` w pętli oblicza sumę intensywności I_i^{amb} światła rozproszonego pochodzącego od włączonych źródeł światła i przekazuje ją procedurze `SSAOColour`, której zadaniem jest zmodyfikowanie tej sumy po obliczeniu czynnika osłabienia i światła dochodzącego od pobliskich powierzchni.

W liniach 24–26 następuje obliczenie wektora $\boldsymbol{w}$ i liczby γ, za pomocą których będzie przy użyciu wzoru (27.2) realizowane obliczanie wektorów $\boldsymbol{l}_i$. W pętli, dla $i = 0, \ldots, N-1$, w liniach 28–30 jest obliczany wektor $\tilde{\boldsymbol{l}}_i$, a na jego podstawie w liniach 31–32 wektor $\boldsymbol{l}_i$. W liniach 33 i 34 jest obliczany (w układzie świata) punkt $\boldsymbol{p} + R\boldsymbol{l}_i$ i jego współrzędne w układzie kostki jednostkowej.

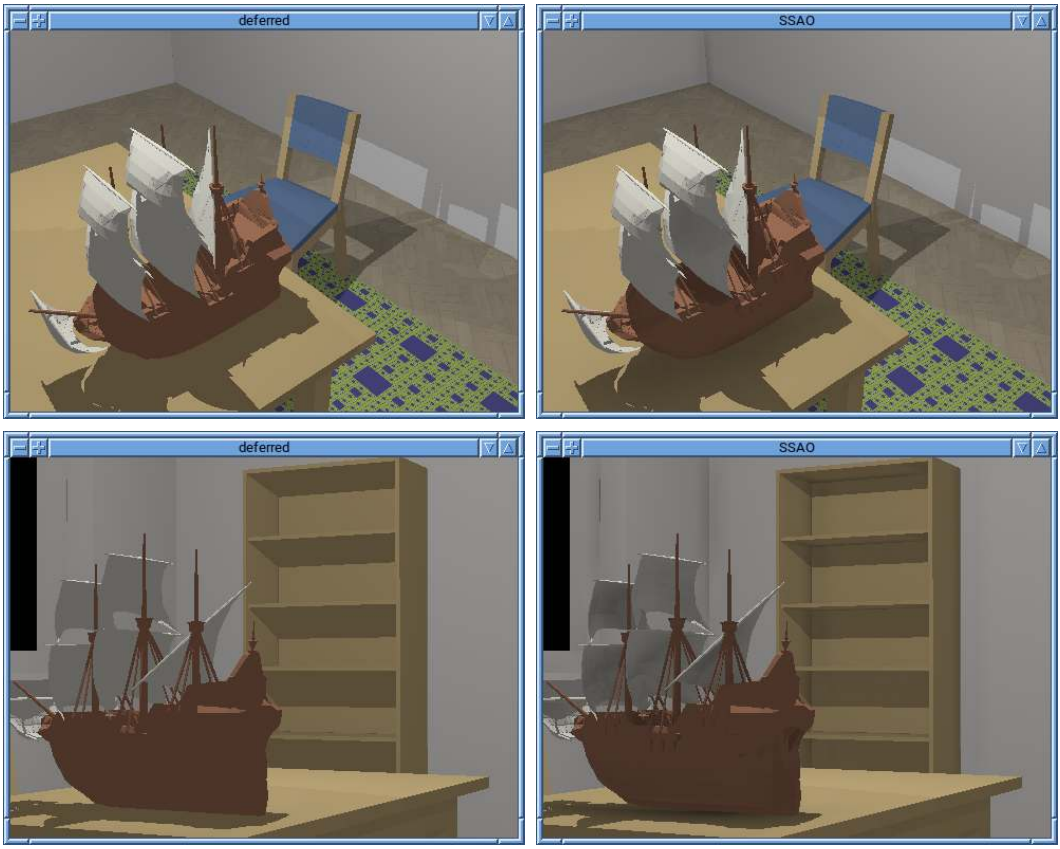
Zmienna k jest licznikiem wektorów $\boldsymbol{l}_i$ należących do sektora, z którego dochodzi światło rozproszone. Jeśli rzut badanego punktu $\boldsymbol{p} + R\boldsymbol{l}_i$ wystaje poza brzeg obrazu, to z braku informacji, że tam jest jakaś powierzchnia zasłaniająca otoczenie, w linii 36 licznik jest zwiększany. Jeśli ten punkt mieści się w obrazie, to w linii 38 odczytywana jest głębokość punktu $\hat{\boldsymbol{p}}_i$ widocznego w miejscu obrazu punktu $\boldsymbol{p} + R\boldsymbol{l}_i$; posługując się teksturą, unikamy obliczania i zaokrąglania współrzędnych (ξ, η) tego punktu w obrazie, a ewaluator tekstury dokonuje interpolacji głębokości zapamiętanej w tekselach, co tu jest pożądane.

W linii 39 badamy, czy głębokość punktu $\boldsymbol{p} + R\boldsymbol{l}_i$ jest mniejsza niż głębokość punktu $\hat{\boldsymbol{p}}_i$ i jeśli tak, to zwiększamy licznik k. W przeciwnym razie w linii 42 obliczane są współrzędne (w układzie świata) wektora $\hat{\boldsymbol{p}}_i - \boldsymbol{p}$. Licznik jest zwiększany, jeśli długość tego wektora jest większa niż $\frac{3}{2}R$,[15] a jeśli nie jest, to obliczamy światło dochodzące do punktu $\boldsymbol{p}$ od powierzchni w pobliżu, tj. od punktu $\hat{\boldsymbol{p}}_i$. Jest tu jednak pewne uproszczenie, które może zafałszować otrzymany obraz. Rzecz w tym, że punkty $\boldsymbol{p}$ i $\hat{\boldsymbol{p}}_i$ mogą „nie widzieć się" nawzajem — jest tak wtedy, gdy iloczyn skalarny wektora $\boldsymbol{p} - \hat{\boldsymbol{p}}_i$ i wektora normalnego $\hat{\boldsymbol{n}}_i$ powierzchni, na

[15] To odstępstwo od algorytmu zapowiedzianego wcześniej uznałem za potrzebne po przeprowadzeniu eksperymentów.

której leży punkt $\hat{\boldsymbol{p}}_i$, nie jest dodatni. Wtedy do punktu $\boldsymbol{p}$ z kierunku $\hat{\boldsymbol{p}}_i - \boldsymbol{p}$ (**uwaga:** *nie* $\boldsymbol{l}_i$) dochodzi światło odbite od pewnego punktu położonego bliżej niż $\hat{\boldsymbol{p}}_i$. Gdyby ten punkt był widoczny dla obserwatora, to można byłoby znaleźć go po zmniejszeniu parametru R, ale jeśli nie jest, to nie znajdziemy w G-buforze informacji, których tam nie ma. W takim przypadku przyjęcie światła dochodzącego z $\hat{\boldsymbol{p}}_i$ działa dobrze pod warunkiem, że ten nieznany punkt jest oświetlony podobnie jak $\hat{\boldsymbol{p}}_i$ i odbija światło podobnie jak on.

Irradiancja światła dochodzącego z punktów w pobliżu $\boldsymbol{p}$ jest sumowana w zmiennej `bleeding`. W linii 50 następuje obliczenie zmodyfikowanej iradiancji światła z otoczenia, która zostanie zapamiętana do następnego etapu w obrazie `ambient`.



Rysunek 27.6. Obrazy bez modyfikacji i ze zmodyfikowanym oświetleniem rozproszonym

Po prawej stronie rysunku 27.6 są pokazane obrazy wykonane przy użyciu opisanego tu algorytmu; promień $R = 0.15$ został dobrany do wymiarów pomieszczenia, $6 \times 4 \times 3$, przy czym jednostce układu świata odpowiada 1 m. Można je porównać z umieszczonymi po lewej stronie obrazami bez modyfikacji oświetlenia. Bez niej w szczególności obszary cienia pod oświetlonymi z góry półkami są jednobarwnymi plamami, a dzięki niej krawędzie styku półek ze ścianami regału stały się widoczne.

28

*Radiometria w służbie grafiki

Przepustkę do umiejętności tworzenia doskonalszych obrazów daje teoria, której elementy są przedstawione w tym rozdziale.

Światło jest promieniowaniem elektromagnetycznym o długości fal w przedziale od 380 nm do 780 nm; dla fal o takiej długości atmosfera ziemska jest przezroczysta i na takie fale reagują receptory w ludzkim oku[1]. Są dwa podejścia do ilościowego opisywania światła. Pierwsze, zwane **radiometrią**, skupia się na **strumieniu energetycznym**, czyli mocy światła wyrażanej w **watach** (W), i wielkościach pochodnych, takich jak gęstość mocy światła padającego na powierzchnię. Drugie podejście, czyli **fotometria**, skupia się na subiektywnym postrzeganiu jasności światła przez ludzi, których oczy różnie reagują na światło o różnych długościach fal. Odpowiednikiem strumienia energetycznego jest w fotometrii **strumień świetlny** mierzony w **lumenach** (lm). Piksele na ekranie komputera emitują światło o określonej mocy, a za zamianę mocy światła na wrażenia „jasno/ciemno" odpowiada zmysł wzroku obserwatora. Dlatego modele oświetlenia potrzebne do *wykonywania* obrazów opierają się na pojęciach radiometrycznych. Przyjrzyjmy im się pokrótce.

28.1. Wielkości radio- i fotometryczne

Rozważmy punkt w przestrzeni emitujący światło we wszystkich kierunkach i załóżmy, że ośrodek (np. powietrze) jest całkowicie przezroczysty. Rozważmy sfery, których wspólnym środkiem jest źródło światła, i wycinek jednej z tych sfer oraz otrzymane z niego za pomocą jednokładności wycinki pozostałych sfer. Na mocy zasady zachowania energii strumień energetyczny światła przechodzącego przez wszystkie te wycinki jest identyczny i proporcjonalny do miary kąta bryłowego zajmowanego przez każdy wycinek na jego sferze — jest ona równa polu wycinka podzielonemu przez kwadrat promienia sfery[2]. Moc światła

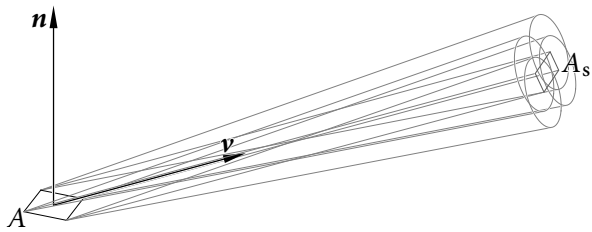
[1] Podany zakres długości fal świetlnych jest umowny; podawane w różnych publikacjach granice między światłem widzialnym a nadfioletem i podczerwienią są w przedziałach $[360\,\text{nm}, 430\,\text{nm}]$ i $[680\,\text{nm}, 830\,\text{nm}]$.

[2] Jednostka określonej w ten sposób miary jest nazywana **steradianem** (sr); bryłowy kąt pełny (całej sfery) ma miarę 4π sr.

wpadającego do oka lub padającego na urządzenie pomiarowe, którego element światłoczuły ma ustalone wymiary, jest zatem odwrotnie proporcjonalna do kwadratu odległości oka lub urządzenia od świecącego punktu. Oczywiście, moc światła wysyłanego w różnych kierunkach może być różna. Iloraz mocy światła rozchodzącego się w wąskim stożku (w zbliżonych kierunkach) i miary kąta bryłowego tego stożka nazywa się **intensywnością kątową.**

Punktowe źródła światła są wygodnymi pojęciami teoretycznymi, ale w skali makroskopowej nie istnieją; zawsze światło jest odbijane lub emitowane przez pewną powierzchnię (lub obszar trójwymiarowy zajmowany np. przez galaretę, mgłę, dym, płomienie lub zjonizowany neon). Rozważmy **element powierzchni** o małej średnicy. Można przyjąć, że powierzchnia ta jest gładka[3], a zatem ma w danym punkcie określony wektor normalny $\boldsymbol{n}$. Niech światło opuszczające rozpatrywany punkt w kierunku pewnego wektora $\boldsymbol{v}$ dociera do obserwatora. Jeśli oba wektory mają ten sam kierunek, to element jest zorientowany „na wprost" do obserwatora i jeśli jego miarę oznaczymy symbolem A, a jego odległość od obserwatora symbolem r, to (przy założeniu, że $A \ll r^2$) element ten zajmuje w polu widzenia obserwatora kąt bryłowy A/r^2 sr. W ogólnym przypadku element zajmuje kąt bryłowy $|\cos \sphericalangle(\boldsymbol{n}, \boldsymbol{v})|A/r^2$ sr i na przykład, gdy wektory $\boldsymbol{n}$ i $\boldsymbol{v}$ są prostopadłe, element w polu widzenia obserwatora zajmuje kąt bryłowy 0 sr. Iloczyn $A_s = |\cos \sphericalangle(\boldsymbol{n}, \boldsymbol{v})|A$ jest **miarą skróconego elementu powierzchni**.

Irradiancja jest to (wyrażana w watach na metr kwadratowy) gęstość strumienia (tj. mocy) światła padającego na element powierzchni (lub na siatkówkę oka), przy czym strumień jest dzielony przez miarę (pole) elementu *nieskróconego*.



Rysunek 28.1. Ilustracja określenia radiancji

Radiancja w kierunku wektora $\boldsymbol{v}$ jest to iloraz strumienia energetycznego światła opuszczającego element powierzchni w kierunkach wektorów należących do wąskiego stożka zawierającego wektor $\boldsymbol{v}$ i miary skróconego elementu oraz miary kąta bryłowego tego stożka (zobacz rys. 28.1), a dokładniej jest to wartość graniczna tego ilorazu, gdy średnica elementu powierzchni dąży do zera i kąt bryłowy stożka dąży do zera, a kierunki elementów tego stożka dążą do kierunku wektora $\boldsymbol{v}$. Wyrażamy ją w watach na metr kwadratowy na steradian.

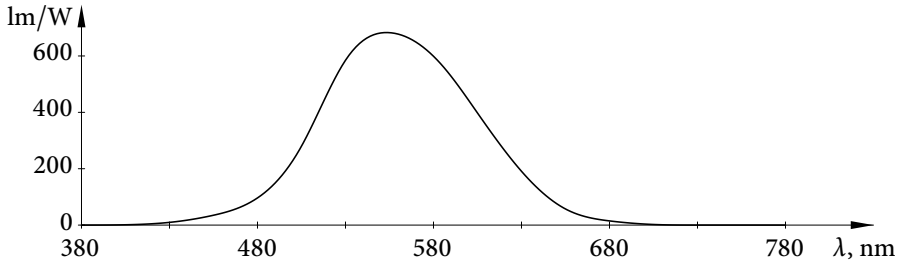
Gęstość mocy światła emitowanego lub odbitego przez powierzchnię (mierzona w watach na metr kwadratowy pola elementu nieskróconego) jest nazywana **emitancją** (*radiosity*). Pojęcie to ma największe znaczenie w obliczeniach globalnego oświetlenia sceny składającej się z powierzchni matowych. Jeśli wypromieniowane lub odbite światło rozchodzi się po jednej

[3] Nie zwracamy tu uwagi na nierówności (chropowatości) powierzchni widoczne tylko przez mikroskop.

stronie (nieprzezroczystej) powierzchni z taką samą radiancją we wszystkich kierunkach, to emitancja jest iloczynem radiancji i liczby π, o czym przekonamy się w podrozdziale 28.3.

Zajmijmy się teraz fotometrią; nie ma ona zastosowania w wykonywaniu obrazów, ale ma znaczenie w projektowaniu wystroju i oświetlenia pomieszczeń — także tych, które chcemy rysować. Subiektywne wrażenie jasności elementu powierzchni zależy od irradiancji światła padającego na siatkówkę oka obserwatora. Okazuje się, że jest ona proporcjonalna do radiancji światła opuszczającego element: obszar na siatkówce, na którym powstaje jego obraz, ma pole proporcjonalne do kąta bryłowego zajmowanego w polu widzenia przez element (a więc jest proporcjonalna do miary skróconego elementu i do odwrotności kwadratu jego odległości od obserwatora). Z kolei moc światła wpadającego przez źrenicę jest odwrotnie proporcjonalna do kwadratu odległości, a zatem podczas dzielenia mocy przez pole obrazu na siatkówce kwadrat odległości ulega skróceniu. Stąd obserwowana jasność dowolnego punktu powierzchni nie zależy od jego odległości od obserwatora[4].

Odpowiednikiem intensywności kątowej w fotometrii jest **światłość**, której jednostka ma nazwę **kandela** (cd = lm/sr). Irradiancji odpowiada **iluminancja**; jej jednostką jest **luks** (lx = $\mathrm{lm/m^2}$). Wreszcie, wielkość opisująca subiektywną jasność powierzchni to **luminancja**. Radiancję mierzy się w $\mathrm{W/(m^2 sr)}$, a jednostką luminancji jest $\mathrm{lm/(m^2 sr)} = \mathrm{cd/m^2}$.



Rysunek 28.2. Zależność między strumieniem energetycznym a strumieniem świetlnym

Dla ustalonej długości fali świetlnej odpowiadające sobie wielkości radiometryczne i fotometryczne, tzn. strumień energetyczny i strumień świetlny, intensywność kątowa i światłość, irradiancja i iluminancja oraz radiancja i luminancja pozostają w tej samej proporcji. Współczynnik proporcjonalności określa czułość receptorów w ludzkim oku na światło o danej długości fali, przy czym jest to łączna czułość receptorów wszystkich rodzajów (zobacz dodatek C). Dla światła o długości fali 555 nm (odpowiadającej maksymalnej czułości receptorów) strumieniowi energetycznemu o mocy jednego wata odpowiada strumień świetlny 683 lumenów. Rysunek 28.2 przedstawia wykres funkcji zwanej **skutecznością świetlną** (*luminous efficacy*), opisującej przelicznik watów na lumeny dla fal o różnych długościach[5].

[4] Innymi słowy, dla dowolnej półprostej, której początkiem jest punkt $\boldsymbol{p}$ świecącej powierzchni, radiancja światła dochodzącego z punktu $\boldsymbol{p}$ do każdego punktu tej półprostej, z którego punkt $\boldsymbol{p}$ jest widoczny przez doskonale przezroczysty ośrodek, jest taka sama. Dlaczego więc inne gwiazdy nie są jasne jak Słońce? Bo gdy ostry obraz obiektu na siatkówce oka jest mniejszy niż pojedynczy receptor światła, wygląda to inaczej.

[5] Ten wykres i wszystkie wykresy w dodatku C trzeba traktować orientacyjnie. Odtworzyłem je (najlepiej jak umiałem) na podstawie różnych źródeł, głównie Wikipedii. Niech mi to będzie wybaczone.

28.2. Dwukierunkowa funkcja odbicia i załamania światła

Rozchodzenie się światła w przestrzeni wypełnionej rozmaitymi ośrodkami optycznymi (w tym oświetlanymi przedmiotami) jest zjawiskiem tak skomplikowanym, że *wszystkie* stosowane w grafice komputerowej modele oświetlenia opisują je w uproszczeniu[6]. W **optyce geometrycznej** zakłada się, że kwanty światła — fotony — w jednorodnych ośrodkach przezroczystych poruszają się po liniach prostych, zmieniając kierunek wskutek odbicia światła od powierzchni lub załamania na granicy przezroczystych ośrodków. Foton może też zmienić kierunek ruchu lub zostać pochłonięty w wyniku oddziaływania z cząstką dymu lub mgły, ale nie może zmienić swojej energii. W tym ujęciu nie mieści się opis (wszechobecnych, choć rzadko dostrzeganych) zjawisk takich jak dyfrakcja, interferencja i luminescencja.

Foton może odbić się od powierzchni lub jego tor może ulec załamaniu na granicy przezroczystych ośrodków, może też oddać swoją energię ośrodkowi (np. w postaci ciepła) i zniknąć. Gdy energia każdego fotonu po odbiciu lub załamaniu pozostaje niezmieniona, mamy do czynienia z tzw. **optyką liniową**, w której obowiązuje **zasada Helmholtza**. Zgodnie z nią, jeśli foton może przebyć pewną drogę, to inny foton może przebyć tę samą drogę w przeciwną stronę[7]. Choć nie każdy używany w grafice komputerowej model oświetlenia powierzchni spełnia zasadę Helmholtza i zasadę zachowania energii, ich spełnienie ma wpływ na jakość obrazów i jest konieczne do poprawnego działania *niektórych* metod obliczania globalnego oświetlenia.

Własności optyczne powierzchni są opisane przez **dwukierunkową funkcję odbicia i załamania światła** $\rho(\boldsymbol{p}; \boldsymbol{l}, \boldsymbol{v})$ (*bidirectional scattering distribution function, BSDF*). Oprócz punktu $\boldsymbol{p}$ powierzchni ma ona dwa argumenty — wektory jednostkowe $\boldsymbol{l}$ i $\boldsymbol{v}$. Jej wartość jest ilorazem radiancji światła odbitego przez powierzchnię lub załamanego na granicy ośrodków w kierunku $\boldsymbol{v}$ i irradiancji światła padającego z kierunku $\boldsymbol{l}$. Funkcja ta zależy od parametrów takich jak wektor normalny $\boldsymbol{n}$ oraz opisu chropowatości powierzchni i własności materiału (w tym współczynników pochłaniania i załamania światła, które zależą od długości fali świetlnej). Bardziej zaawansowane modele oświetlenia uwzględniają też polaryzację światła, ale dokładne ich opisy wykraczają poza ramy tej książki.

W roku 1986 James T. Kajiya [47] sformułował równanie, które nazwał *the rendering equation*; jest ono podstawą algorytmów wykonywania obrazów wysokiej jakości, nawet fotorealistycznych. Zgodnie z tym równaniem radiancja punktu $\boldsymbol{p}$ powierzchni, obserwowanego z dowolnego kierunku $\boldsymbol{v}$, jest sumą radiancji światła wypromieniowanego (jeśli obiekt świeci) i światła odbitego i załamanego przez powierzchnię w tym kierunku:

$$L(\boldsymbol{p}, \boldsymbol{v}) = L_{\mathrm{e}}(\boldsymbol{p}, \boldsymbol{v}) + L_{\mathrm{r}}(\boldsymbol{p}, \boldsymbol{v}). \tag{28.1}$$

[6] Pełna teoria fizyczna, czyli **elektrodynamika kwantowa**, jest zbyt skomplikowana, aby można było ją stosować w grafice, a prostsze modele dają wystarczająco dokładny opis praktycznie wszystkich zjawisk optycznych spotykanych „na co dzień".

[7] Zasada Helmholtza ma też sformułowanie dla światła spolaryzowanego, nie obowiązuje natomiast w silnych polach magnetycznych, w pewnych (rzadko spotykanych) kryształach i wtedy, gdy przedmioty poruszają się z wielkimi prędkościami. Zatem „na co dzień" mamy do czynienia z optyką liniową.

Równanie wyraża zatem bilans energetyczny w danym punkcie (lub raczej w jego małym otoczeniu). Radiancja światła odbitego i załamanego w kierunku wektora $\boldsymbol{v}$ jest całką:

$$L_{\mathrm{r}}(\boldsymbol{p},\boldsymbol{v}) = \int_{\boldsymbol{l}\in S} \rho(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v}) I(\boldsymbol{p},\boldsymbol{l})\,\mathrm{d}S. \tag{28.2}$$

Całkowanie odbywa się po sferze jednostkowej S, tj. po wszystkich, reprezentowanych przez jednostkowe wektory $\boldsymbol{l}$ kierunkach, z których dochodzi światło. Symbol $I(\boldsymbol{p},\boldsymbol{l})$ oznacza irradiancję powierzchni w punkcie $\boldsymbol{p}$ światłem dochodzącym z kierunku $\boldsymbol{l}$. Obraz oświetlonych przedmiotów przedstawia zatem mniej lub bardziej dokładnie obliczone wartości tej całki dla punktów widocznych na tym obrazie. Jeśli uwzględnione jest tylko światło dochodzące bezpośrednio od źródeł punktowych, to całkowanie sprowadza się do obliczenia skończonej sumy. Z kolei w p. 18.4.3 obliczyliśmy całkę opisującą *radiancję* światła odbitego od powierzchni lambertowskiej, zakładając, że światło dochodzi z nieskończenie wielu (tj. wszystkich) kierunków, a jego radiancja ma stałe wartości w połowach sfery S.

Aby znaleźć warunek zapewniający spełnienie zasady zachowania energii, rozważmy płaski element powierzchni o małej średnicy; jego pole, oznaczone literą A, też jest małe. Niech $\boldsymbol{p}$ oznacza dowolny punkt tego elementu i niech $\boldsymbol{n}$ będzie jego jednostkowym wektorem normalnym. Strumień energetyczny światła padającego z kierunku wektora $\boldsymbol{l}$ na ten element jest równy $E = AI(\boldsymbol{p},\boldsymbol{l})$.

Dla dowolnego wektora $\boldsymbol{v}$, w którego kierunku światło zostało odbite lub załamane, rozważmy niewielki wycinek sfery jednostkowej S zawierający ten wektor. Pole, tj. miarę kąta bryłowego tego wycinka oznaczmy symbolem $\mathrm{d}S$. Wtedy strumień energii światła z tego elementu w obrębie tego kąta bryłowego jest iloczynem radiancji $L(\boldsymbol{p},\boldsymbol{v})$, miary skróconego elementu $A|\cos\sphericalangle(\boldsymbol{n},\boldsymbol{v})|$ i miary kąta bryłowego $\mathrm{d}S$ (rys. 28.1). Całkowita moc światła odbitego i załamanego przez element jest całką po wszystkich kierunkach, w których światło się rozchodzi, tj. po sferze jednostkowej S:

$$\int_{\boldsymbol{v}\in S} L(\boldsymbol{p},\boldsymbol{v}) A|\cos\sphericalangle(\boldsymbol{n},\boldsymbol{v})|\,\mathrm{d}S = AI(\boldsymbol{p},\boldsymbol{l}) \int_{\boldsymbol{v}\in S} \rho(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v})|\cos\sphericalangle(\boldsymbol{n},\boldsymbol{v})|\,\mathrm{d}S.$$

Ponieważ nie wszystkie fotony padające na element opuszczają go, z zasady zachowania energii wynika, że powyższa całka musi być mniejsza niż E, a zatem dla każdego wektora $\boldsymbol{l}$ musi być

$$\int_{\boldsymbol{v}\in S} \rho(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v})|\cos\sphericalangle(\boldsymbol{n},\boldsymbol{v})|\,\mathrm{d}S < 1.$$

Uwaga: Opisane w podrozdziale 18.1 modele oświetlenia Phonga i Blinna-Phonga nie spełniają zasady Helmholtza. Spełnienie zasady zachowania energii przez te modele zależy od doboru występujących w nich parametrów. Sprawdzenie, przez obliczenie powyższej całki (dla *wszystkich* wektorów jednostkowych $\boldsymbol{l}$), czy dane parametry zapewniają zachowanie energii, jest dosyć trudne, ale dla modelu Lamberta jest to znacznie łatwiejsze, o czym przekonamy się już w następnym podrozdziale.

Dwukierunkowa funkcja odbicia i załamania światła jest zwykle przedstawiana jako suma dwóch funkcji oznaczanych skrótami BRDF i BTDF, z których pierwsza, ρ_r, charakteryzuje odbijanie (*reflectance*) światła, a druga, ρ_t, opisuje przechodzenie (*transmission*) światła przez powierzchnię rozgraniczającą przezroczyste ośrodki; jeśli oświetlany obiekt jest nieprzezroczysty, to ta druga funkcja ma wartość 0 dla każdej pary wektorów $(\boldsymbol{l}, \boldsymbol{v})$. Z zasady Helmholtza wynika, że dwukierunkowa funkcja odbicia (BRDF) jest symetryczna: w każdym punkcie $\boldsymbol{p}$ dla każdej pary wektorów $(\boldsymbol{l}, \boldsymbol{v})$ jest spełniona równość $\rho_r(\boldsymbol{p}; \boldsymbol{l}, \boldsymbol{v}) = \rho_r(\boldsymbol{p}; \boldsymbol{v}, \boldsymbol{l})$. Natomiast funkcja przechodzenia (BTDF) jest iloczynem funkcji symetrycznej i czynnika $(\eta_2/\eta_1)^2$ określonego przez współczynniki załamania światła ośrodków, w których foton porusza się po przejściu i przed przejściem przez rozgraniczającą je powierzchnię[8].

28.3. Model oświetlenia Lamberta

Zauważmy, że gdy światło dochodzi do powierzchni z określonego kierunku (wektora $\boldsymbol{l}$), irradiancja jest proporcjonalna do kosinusa kąta między wektorami $\boldsymbol{n}$ a $\boldsymbol{l}$. Dlatego w lambertowskim modelu oświetlenia, który wprowadziliśmy w podrozdziale 10.2, dwukierunkowa funkcja odbicia i załamania światła przyjmuje tylko dwie wartości: stałą dodatnią[9], jeśli obserwator widzi oświetloną stronę powierzchni, co ma miejsce gdy iloczyny skalarne $\langle \boldsymbol{l}, \boldsymbol{n} \rangle$ i $\langle \boldsymbol{v}, \boldsymbol{n} \rangle$ mają te same znaki, i 0, jeśli źródło światła i obserwator znajdują się po jej przeciwnych stronach. Niezależność wspomnianej stałej od wektora $\boldsymbol{v}$ oznacza brak odblasków na powierzchni, która w tym modelu jest nieprzezroczysta, idealnie gładka i idealnie matowa. Jest oczywiste, że model Lamberta spełnia zasadę Helmholtza[10].

28.3.1. Zasada zachowania energii w modelu Lamberta

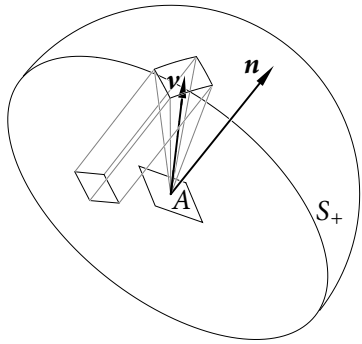
Rozważana w poprzednim podrozdziale całka jest w modelu Lamberta całką po półsferze S_+ składającej się z takich wektorów $\boldsymbol{v}$, dla których wspomniane iloczyny skalarne mają ten sam znak[11]. Obliczenie całki po powierzchni polega na podzieleniu jej na małe fragmenty, pomnożeniu miary (pola) każdego takiego fragmentu przez wartość funkcji podcałkowej w dowolnym punkcie tego fragmentu, obliczeniu sumy iloczynów i przejściu do granicy, gdy średnice fragmentów dążą do zera, a ich liczba do nieskończoności. Iloczyn pola wycinka (tj. fragmentu) półsfery i funkcji podcałkowej — kosinusa kąta między wektorami $\boldsymbol{n}$ a $\boldsymbol{v}$ — jest

[8] Obecność tego czynnika wynika z zasady zachowania energii; po uzasadnienie odsyłam do książki [35]. Z zasady Helmholtza wynika równość zapisana „symetrycznym" wzorem $\eta_i^2 \rho_t(\boldsymbol{p}; \boldsymbol{l}, \boldsymbol{v}) = \eta_2^2 \rho_t(\boldsymbol{p}; \boldsymbol{v}, \boldsymbol{l})$.

[9] Stała ta jest w istocie funkcją długości fali świetlnej λ. Zwykle w obliczeniach funkcję tę reprezentujemy za pomocą wektora o współrzędnych r, g, b (czyli za pomocą trzech stałych skalarnych), odpowiadających uśrednionym wartościom tej funkcji w zakresach długości fal światła czerwonego, zielonego i niebieskiego.

[10] Ale zrealizowany w aplikacji 1D sposób obliczania oświetlenia powierzchni zakrzywionej przybliżanej przez płaskie trójkąty (zobacz s. 286–287) narusza tę zasadę, bo zamiana wektorów $\boldsymbol{l}$ i $\boldsymbol{v}$ *może* zmienić wynik badania, czy obserwator widzi oświetloną stronę powierzchni. Takie subtelności mają jednak znaczenie tylko w najbardziej zaawansowanych metodach obliczania globalnego oświetlenia.

[11] Symbol S_+ oznacza zatem jedną z półsfer, $S_{\boldsymbol{n}+}$ albo $S_{\boldsymbol{n}-}$, składających się odpowiednio z takich wektorów jednostkowych $\boldsymbol{l}$, że $\langle \boldsymbol{l}, \boldsymbol{n} \rangle > 0$ i $\langle \boldsymbol{l}, \boldsymbol{n} \rangle < 0$.



Rysunek 28.3. Całkowanie mocy światła odbitego w modelu Lamberta

w przybliżeniu (tym lepszym, im mniejsza jest średnica wycinka) polem rzutu prostopadłego tego wycinka na płaszczyznę elementu A, co ilustruje rysunek 28.3 (zobacz też rys. 27.5). Rzuty wszystkich wycinków (niezależnie od sposobu podziału półsfery) wypełniają koło o promieniu 1. Całka po półsferze S_+ z kosinusa kąta między wektorami $\boldsymbol{n}$ a $\boldsymbol{v}$ jest więc polem tego koła. Mamy zatem

$$\int_{\boldsymbol{v}\in S} \rho(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v})|\cos\measuredangle(\boldsymbol{n},\boldsymbol{v})|\,\mathrm{d}S = \rho(\boldsymbol{p})\int_{\boldsymbol{v}\in S_+} |\cos\measuredangle(\boldsymbol{n},\boldsymbol{v})|\,\mathrm{d}S = \rho(\boldsymbol{p})\pi.$$

Z otrzymanego wzoru i zasady zachowania energii wynika, że musi być $\rho(\boldsymbol{p}) < 1/\pi$. Iloczyn $c_\lambda = \rho(\boldsymbol{p})\pi$ określa, jaka część mocy światła o długości fali λ padającego na powierzchnię w małym otoczeniu punktu $\boldsymbol{p}$ opuszcza ją jako światło. Zastępując w powyższych rachunkach funkcję ρ przez radiancję L, która dla powierzchni lambertowskich jest stała w półsferze $S_{\boldsymbol{n}+}$, otrzymamy wspomniany wcześniej związek między radiancją a emitancją.

28.3.2. Oświetlenie przez obraz i jego przybliżenie wielomianowe

Obiekty rysowane przez aplikacje opisane we wcześniejszych rozdziałach są oświetlone przez źródła punktowe i przez rozproszone w otoczeniu światło, o którym założyliśmy, że jego radiancja nie zależy od kierunku, z którego światło to dochodzi; w rezultacie kształty części obiektów oświetlonych tylko przez to światło są na obrazach niewidoczne. Lepszy efekt daje opisane w p. 18.4.3 oświetlenie hemisferyczne. Jeszcze lepsze wyniki można uzyskać, uwzględniając oświetlenie, którego radiancja ma rozkład kierunkowy określony przez źródła światła i przedmioty otaczające rysowany obiekt. Dla powierzchni lambertowskich daje się to osiągnąć szczególnie tanio i to właśnie jest powodem, dla którego w szaderach realizujących nawet znacznie bardziej wyrafinowane modele oświetlenia często do opisu światła odbitego w sposób rozproszony przyjmuje się model Lamberta.

Składnik L_r opisujący światło odbite od punktu $\boldsymbol{p}$ powierzchni obliczymy na podstawie wzoru (28.2). Przyjmiemy, że $\langle\boldsymbol{v},\boldsymbol{n}\rangle \geqslant 0$ i obliczymy całkę po półsferze $S_{\boldsymbol{n}+}$ — zbiorze wektorów jednostkowych $\boldsymbol{l}$, dla których $\langle\boldsymbol{l},\boldsymbol{n}\rangle \geqslant 0$; w tym zbiorze funkcja ρ przyjmuje stałą wartość dodatnią, którą oznaczyliśmy $\rho(\boldsymbol{p})$. Rozważmy (dostatecznie mały) element A powierzchni

zawierający punkt $\boldsymbol{p}$ i wycinek dS półsfery $S_{\boldsymbol{n}+}$ zawierający wektor $\boldsymbol{l}$. Aby obliczyć[12] strumień energetyczny światła padającego na element A z kierunków należących do wycinka dS, należy pomnożyć radiancję $L(\boldsymbol{p}+\boldsymbol{l},-\boldsymbol{l})$ światła dochodzącego z kierunku $\boldsymbol{l}$,[13] miarę wycinka i miarę kąta bryłowego zajmowanego przez element A widziany z punktu $\boldsymbol{p}+\boldsymbol{l}$; ten ostatni czynnik jest równy $A\cos\measuredangle(\boldsymbol{l},\boldsymbol{n})$. Po podzieleniu przez A otrzymamy irradiancję powierzchni w punkcie $\boldsymbol{p}$. Stąd w modelu Lamberta radiancja światła odbitego od powierzchni oświetlonej ze *wszystkich* kierunków jest równa

$$L_{\mathrm{r}}(\boldsymbol{p}) = \rho(\boldsymbol{p}) \int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} L(\boldsymbol{p}+\boldsymbol{l},-\boldsymbol{l})\langle \boldsymbol{l},\boldsymbol{n}\rangle \,\mathrm{d}S = \rho(\boldsymbol{p}) I(\boldsymbol{p},\boldsymbol{n}). \tag{28.3}$$

Jeśli obiekt jest tak mały, że oświetlenie wszystkich jego punktów jest opisane przez tę samą funkcję L, to irradiancja opisana przez całkę w tym wzorze zależy *tylko* od wektora $\boldsymbol{n}$.

Najczęściej w praktyce są stosowane dwa sposoby reprezentowania funkcji I. Pierwszy sposób, czyli **oświetlenie przez obraz** (*image-based lighting*, *IBL*), polega na użyciu tekstury kostkowej. Dla każdego teksela trzeba wyznaczyć odpowiedni wektor $\boldsymbol{n}$, a następnie obliczyć i zapamiętać wartość funkcji $I(\boldsymbol{n})$. W tym zastosowaniu nie ma potrzeby używania tekstur o dużej rozdzielczości, bo zmienianie wektora $\boldsymbol{n}$ powoduje bardzo powolne zmiany wartości tej funkcji. W większości przypadków wystarczy rozdzielczość $6\times 64\times 64$ lub mniejsza.

Drugi sposób polega na zastąpieniu funkcji $I(\boldsymbol{n})$ przez przybliżający ją wielomian zmiennych x, y, z będących współrzędnymi wektora $\boldsymbol{n}$, co zajmuje znacznie mniej miejsca w pamięci niż tekstura. Ramamoorthi i Hanrahan [51], którzy wynaleźli ten sposób, stwierdzili, że użycie wielomianu drugiego stopnia zapewnia błąd na poziomie 1%, niedostrzegalny na obrazach[14]. Zaimplementujemy kolejno oba sposoby, po opanowaniu niezbędnych podstaw.

Rysunek 28.4a przedstawia teksturę kostkową (*skybox*) zestawioną z fotografii wykonanych w Beskidach pod szczytem Babiej Góry. Każdy teksel tekstury pokazanej na rysunku 28.4b przechowuje wartość $I(\boldsymbol{n})$ całki (28.3), obliczonej na półsferze $S_{\boldsymbol{n}+}$ wyznaczonej przez wektor $\boldsymbol{n}$ odpowiadający temu tekselowi, a dokładniej, przechowuje wektor (r,g,b), którego współrzędne są całkami ze składowych irradiancji światła kolorowego.

Rozważane tu całki trzeba obliczać numerycznie. Za pomocą opisanych dalej współrzędnych sferycznych można sprowadzić zadanie do całkowania po prostokącie. Inna możliwość, z której tu skorzystałem, polega na wybraniu węzłów kwadratury na półsferze przez rozmieszczenie w kole jednostkowym N punktów za pomocą wzoru (27.1), a następnie obliczeniu wektorów $\boldsymbol{l}_i$ na półsferze $S_{\boldsymbol{n}+}$ w sposób opisany w podrozdziale 27.3. Wektory $\boldsymbol{l}_i$ są węzłami kwadratury przybliżającej obliczane całki. Kwadratura ta jest pomnożoną przez $\frac{\pi}{N}$ sumą wartości radiancji w punktach $\boldsymbol{l}_i$.[15]

[12] w przybliżeniu — tym lepszym, im mniejsze są średnice elementu A i wycinka dS

[13] Fotony dochodzące z kierunku wektora $\boldsymbol{l}$ poruszają się w przeciwną stronę, dlatego drugim argumentem funkcji L jest wektor „$-\boldsymbol{l}$".

[14] Irradiancja oświetlenia hemisferycznego jest opisana przez wielomian stopnia 1.

[15] Liczba N węzłów kwadratury powinna być dobrana do kierunkowego rozkładu radiancji i dla skomplikowanych obrazów otoczenia może być potrzebne przyjęcie dużego N, choć dla obrazów na rysunku 28.4a w zupełności wystarczyło użycie kwadratury z 1500 węzłami. Podany w książce [25] szader, który całkuje irradiancję za pomocą współrzędnych sferycznych, oblicza (dla każdego teksela) wartości radiancji w ponad 15000 punktów — być może taka liczba punktów została przyjęta „na zapas".

Rysunek 28.4. Tekstura radiancji otoczenia (a) i otrzymana z niej tekstura irradiancji (b)

Z uwagi na ilość obliczeń warto do obliczenia całek zatrudnić GPU. Listingi 28.1–28.3 przedstawiają szadery, z których składa się program całkowania irradiancji wykonywany przez procedurę z listingu 28.4. Trzeba ją wywołać dla każdej tekstury kostkowej, którą chcemy przerobić na teksturę irradiancji. Szader wierzchołków nie robi nic, jego zadaniem jest być na miejscu; program jest uruchamiany poleceniem narysowania jednego punktu, który zostanie zamieniony na ściany sześcianu przez szader geometrii.

Listing 28.1. Szader wierzchołków programu całkowania irradiancji

GLSL

```
1: #version 450 core
2: void main ( void ) { }
```

Szader geometrii jest wywoływany w sześciu instancjach, z których każda wyprowadzi jedną, złożoną z dwóch trójkątów ścianę sześcianu. Każda z tych ścian zostanie narysowana na jednej ścianie tworzonej tekstury irradiancji. O tym, na której, decyduje przypisana (w linii 20) zmiennej `gl_Layer` wartość będąca numerem instancji; kolejność jest, choć nie musi być, zgodna z kolejnością ścian tekstury kostkowej, odpowiednio $x > 0$, $x < 0$, $y > 0$, $y < 0$, $z > 0$ i $z < 0$. Każda ściana tworzonej tekstury ma wypełnić klatkę (*viewport*), a zatem do etapu rasteryzacji szader przekazuje opisane w tablicy p wierzchołki kwadratu będącego przekrojem płaszczyzną $z = 0$ kostki standardowej[16].

[16] Podając wierzchołki od razu w układzie kostki standardowej, obywamy się *bez* rzutowania — nie musimy konstruować macierzy rzutowania, unikamy związanych z nim błędów zaokrągleń i oszczędzamy czas.

Listing 28.2. Szader geometrii programu całkowania irradiancji

GLSL

```
#version 450 core

layout(points,invocations=6) in;

layout(triangle_strip,max_vertices=4) out;

out vec3 Normal;

const vec4 p[4] = {{1,-1,0,1},{1,1,0,1},{-1,-1,0,1},{-1,1,0,1}};
const vec3 cv[8] = {vec3(-1.0,-1.0,-1.0),vec3(-1.0, 1.0,-1.0),
                    vec3(-1.0, 1.0, 1.0),vec3(-1.0,-1.0, 1.0),
                    vec3( 1.0,-1.0,-1.0),vec3( 1.0, 1.0,-1.0),
                    vec3( 1.0, 1.0, 1.0),vec3( 1.0,-1.0, 1.0)};
const int cvi[24] = {5,4,6,7, 2,3,1,0, 5,6,1,2, 7,4,3,0, 6,7,2,3, 1,0,5,4};

void main ( void )
{
  int i, k;

  gl_Layer = gl_InvocationID;
  for ( i = 0, k = 4*gl_InvocationID;  i < 4;  i++, k++ ) {
    Normal = cv[cvi[k]];
    gl_Position = p[i];
    EmitVertex ();
  }
  EndPrimitive ();
} /*main*/
```

Jednocześnie ze współrzędnymi jednorodnymi wierzchołka w układzie kostki standardowej szader wyprowadza w zmiennej `Normal` współrzędne kartezjańskie wierzchołka sześcianu $[-1,1]^3$ w układzie świata, wybierając te współrzędne z tablicy `cv` na podstawie numeru instancji. Czwórki liczb w tablicy `cvi` są numerami wierzchołków sześcianu dla poszczególnych ścian — i tu musi być[17] zapewniona zgodność orientacji każdej ściany sześcianu z orientacją odpowiedniej ściany tworzonej tekstury kostkowej (zobacz rys. 26.3).

Szader fragmentów (listing 28.3) w zmiennej `Normal` otrzymuje wynik interpolacji wierzchołków sześcianu, czyli współrzędne odpowiadającego fragmentowi punktu na ścianie tego sześcianu. Punkt ten określa wektor $\boldsymbol{n}$ i wyznaczoną przezeń półsferę $S_{\boldsymbol{n}^+}$, po której wątek szadera ma obliczyć całkę.

Konstrukcja odbicia symetrycznego H w liniach 19–21 i wektorów $\boldsymbol{l}_i$ w liniach 24–28 jest taka sama jak w procedurze `SSAOColour` na listingu 27.12. Radiancja światła dochodzącego z kierunku $\boldsymbol{l}_i$ jest wartością tekstury dostępnej poprzez zmienną `RadianceTxt`.

[17] i jest

Listing 28.3. Szader fragmentów programu całkowania irradiancji

GLSL

```
1: #version 450 core
2:
3: #define PI   3.141592653
4: #define DPHI 2.399963229
5:
6: in vec3 Normal;
7:
8: layout(location=0) out vec4 out_Colour;
9:
10: layout(binding=0) uniform samplerCube RadianceTxt;
11: uniform int N = 1500;
12:
13: void main ( void )
14: {
15:   int   i;
16:   float phi, r, gamma;
17:   vec3  l, w, irrad;
18:
19:   w = normalize ( Normal );
20:   w.z += Normal.z > 0.0 ? 1.0 : -1.0;
21:   gamma = 2.0 / dot ( w, w );
22:   irrad = vec3(0.0);
23:   for ( i = 1, phi = 0.5*DPHI;  i < 2*N;  i += 2, phi += DPHI ) {
24:     r = sqrt ( float(i)/float(2*N) );
25:     l = vec3 ( r*cos ( phi ), r*sin ( phi ),
26:                sqrt ( float(2*N-i)/float(2*N) ) );  /* sqrt ( 1 - r*r ) */
27:     l -= w*(gamma*dot ( w, l ));
28:     if ( Normal.z > 0.0 ) l = -l;
29:     irrad += texture ( RadianceTxt, l ).rgb;
30:   }
31:   out_Colour = vec4 ( irrad*(PI/float(N)), 1.0 );
32: } /*main*/
```

Parametrami procedury na listingu 28.4 są identyfikator programu zbudowanego z opisanych wyżej szaderów i identyfikator tekstury kostkowej opisującej radiancję otoczenia. W linii 7 jest wywołana procedura `CreateCubeTexture` z listingu 26.11, która tworzy teksturę o żądanej wielkości i rezerwuje miejsce w pamięci na jej teksele; procedura ta pozostawia teksturę przywiązaną do celu `GL_TEXTURE_CUBE_MAP`, zatem instrukcje w kolejnych liniach nadają wartości jej parametrom. W liniach 10–16 powstaje pozaekranowy bufor ramki, którego załącznikiem staje się ta tekstura. Przed rysowaniem nie ma potrzeby kasowania tła, bo wątki szadera fragmentów nadadzą wartości *wszystkim* tekselom. Tekstura przekazana jako parametr jest w linii 18 udostępniana szaderowi. Po wykonaniu obrazu niepotrzebny już bufor ramki jest likwidowany, a instrukcja w linii 28 przekazuje wynik — identyfikator gotowej tekstury irradiancji.

Listing 28.4. Procedura całkowania irradiancji

C

```
#define IBLTXTSIZE 64

GLuint ConstructIBLTexture ( GLuint program_id, GLuint radtx )
{
  GLuint fbo, irradtx;

  irradtx = CreateCubeTexture ( IBLTXTSIZE, GL_RGB32F, 0 );
  glTexParameteri ( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  glTexParameteri ( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
  glGenFramebuffers ( 1, &fbo );
  glBindFramebuffer ( GL_FRAMEBUFFER, fbo );
  glFramebufferTexture ( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, irradtx, 0 );
  if ( glCheckFramebufferStatus ( GL_FRAMEBUFFER ) !=
       GL_FRAMEBUFFER_COMPLETE )
    ExitOnError ( "ConstructIBLTexture" );
  glDrawBuffer ( GL_COLOR_ATTACHMENT0 );
  glViewport ( 0, 0, IBLTXTSIZE, IBLTXTSIZE );
  glBindTexture ( GL_TEXTURE_CUBE_MAP, radtx );
  glUseProgram ( program_id );
  glBindVertexArray ( empty_vao );
  glDrawArrays ( GL_POINTS, 0, 1 );
  glFlush ();
  glUseProgram ( 0 );
  glBindVertexArray ( 0 );
  glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
  glDeleteFramebuffers ( 1, &fbo );
  ExitIfGLError ( "ConstructIBLTexture" );
  return irradtx;
} /*ConstructIBLTexture*/
```

Utworzona przez procedurę `ConstructIBLTexture` tekstura musi mieć szeroki zakres dynamiczny (*high dynamic range*, zobacz podrozdz. 25.2), czyli przechowywać dane jako 32-bitowe liczby zmiennopozycyjne (co jest określone przez drugi parametr procedury `CreateCubeTexture`). Użycie (stałopozycyjnych) liczb ośmiobitowych nie wystarczy, bo wielkości radiometryczne mogą być reprezentowane przez liczby większe niż 1, a dokładność reprezentacji ośmiobitowej, odpowiedniej dla obrazów przeznaczonych do wyświetlania na ekranie, jest za mała dla danych, które będą przetwarzane w kolejnych obliczeniach.

Opisaną tu procedurę można i warto uzupełnić o nadawanie wartości zmiennej jednolitej `N` (listing 28.3, linia 11) — bez niego szader fragmentów użyje kwadratury o domyślnej liczbie węzłów. Ta modyfikacja umożliwiłaby dobieranie liczby węzłów do rozkładów kierunkowych radiancji reprezentowanych przez różne obrazy.

Użycie tekstury irradiancji do wykonania końcowego obrazu jest już bardzo łatwe; listing 28.5 przedstawia modyfikacje szadera z listingów 10.4 i 12.8, mające na celu zastąpienie oświetlenia bezkierunkowego oświetleniem przez obraz. Pole `ambient` opisu źródła świat-

Listing 28.5. Szader fragmentów realizujący lambertowskie oświetlenie przez obraz

GLSL

```
1: layout(binding=0) uniform samplerCube IrradianceTxt;
2:
3: vec3 LambertLighting ( void )
4: {
5:   vec3  normal, lv, vv, Colour;
6:   float d, dist;
7:   uint  i, mask;
8:
9:   normal = normalize ( In.Normal );
10:  vv = posDifference ( trb.eyepos, In.Position, dist );
11:  if ( dot ( vv, In.TNormal ) < 0.0 )
12:    normal = -normal;
13:  Colour = texture ( IrradianceTxt, normal ).rgb * mm.diffref;
14:  for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
15:    if ( (light.mask & mask) != 0 ) {
16:      lv = posDifference ( light.ls[i].position, In.Position, dist );
17:      d = dot ( lv, normal );
18:      ....  /* obliczanie i sumowanie oświetlenia przez źródła punktowe */
19:      ....  /* z uwzględnieniem cieni, zobacz listing 22.5 */
20:    }
21:  return clamp ( Colour, 0.0, 1.0 );
22: } /*LambertLighting*/
```

ła nie jest tu używane i odwołujące się do niego instrukcje zostały usunięte. Zamiast nich, irradiancja rozproszonego w otoczeniu światła oświetlającego fragment powierzchni o wektorze normalnym $\boldsymbol{n}$, wzięta z tekstury, jest w linii 13 mnożona przez wartość dwukierunkowej funkcji odbicia światła, podaną w polu `diffref` zmiennej `mm` zawierającej opis materiału.

Całkując irradiancję, można uwzględnić też bezpośrednie oświetlenie powierzchni przez źródła punktowe, na przykład Słońce, ale to by wykluczyło otrzymanie obrazu cieni rzucanych na powierzchnię rysowanego przedmiotu przez inne jego części lub inne przedmioty. Dlatego w pokazanym tu przykładzie takie oświetlenie jest realizowane przez pętlę w liniach 14–20. Zauważmy, że w instrukcji warunkowej w liniach 11–12 wektorom $\boldsymbol{n}$ i $\boldsymbol{m}$ nadajemy takie zwroty, aby iloczyn skalarny $\langle \boldsymbol{v}, \boldsymbol{m} \rangle$ był nieujemny[18]. Umożliwia to uproszczenie instrukcji pominiętej na listingu.

Aby znaleźć wielomian dobrze przybliżający funkcję $I(\boldsymbol{n})$, należy skorzystać z faktu, że funkcje określone na sferze (w szczególności funkcje L i I oraz wielomiany) są *wektorami*, czyli obiektami, które można dodawać i które można mnożyć przez liczby. W przestrzeni liniowej składającej się z takich wektorów określa się **iloczyn skalarny** wzorem

$$\langle f, g \rangle = \int_{\boldsymbol{w} \in S} f(\boldsymbol{w}) g(\boldsymbol{w}) \, \mathrm{d}S. \tag{28.4}$$

[18] Wektor $\boldsymbol{m}$ jest jednostkowym wektorem normalnym płaszczyzny trójkąta, zobacz podrozdział 12.1. Jego zwrot zmieniamy w wyobraźni, aby zachować warunek $\langle \boldsymbol{m}, \boldsymbol{n} \rangle > 0$.

Dwa wektory (dwie funkcje) są wzajemnie prostopadłe, gdy ich iloczyn skalarny jest zerem. **Normę**, czyli długość wektora f, definiujemy wzorem $\|f\| = \sqrt{\langle f, f\rangle}$. Niech V oznacza podprzestrzeń, której elementami są wszystkie wielomiany stopnia co najwyżej 2; jeden z nich, p, jest poszukiwanym przybliżeniem funkcji I. Różnica $I - p$ ma najmniejszą długość (i w tym sensie wielomian p jest najlepszym przybliżeniem funkcji I), gdy jest wektorem prostopadłym do wszystkich elementów przestrzeni V, co ma miejsce wtedy, gdy wektor p jest rzutem prostopadłym wektora I na tę przestrzeń[19].

Każdy wielomian stopnia co najwyżej 2 zmiennych x, y, z jest kombinacją liniową wielomianów $1, x, y, z, x^2, y^2, z^2, xy, yz, zx$, a więc zbiór wszystkich takich wielomianów jest przestrzenią dziesięciowymiarową. Zauważmy jednak, że $x^2 + y^2 + z^2 - 1 = 0$ w każdym punkcie (x, y, z) sfery S, zatem wielomiany wymienione wyżej po ograniczeniu dziedziny do tej sfery są liniowo zależne, wskutek czego współczynniki kombinacji liniowej, którą należy znaleźć (tj. wielomianu p), nie są jednoznacznie określone. Ten problem można rozwiązać, wybierając wielomiany spełniające **równanie Laplace'a**, czyli takie, których suma pochodnych cząstkowych drugiego rzędu ze względu na x, y, z jest równa 0. Rozwiązania równania Laplace'a są nazywane **funkcjami harmonicznymi**[20]. Można znaleźć zbiór dziewięciu takich liniowo niezależnych wielomianów stopnia co najwyżej 2, będący **bazą** przestrzeni V, zatem ma ona wymiar 9. W konstrukcji rzutu prostopadłego najwygodniej jest użyć **bazy ortogonalnej**, której każdy element jest wektorem prostopadłym do wszystkich pozostałych. Mając taką bazę złożoną z wielomianów $p_0, \ldots, p_8$, możemy rzut prostopadły funkcji I na przestrzeń V otrzymać za pomocą wzoru

$$p = \sum_{i=0}^{8} \frac{\langle p_i, I\rangle}{\|p_i\|^2} p_i. \tag{28.5}$$

Opis oświetlenia wykorzystywany przez szader może być tablicą liczb $a_i = \langle p_i, I\rangle / \|p_i\|^2$; aby go przygotować, należy wybrać odpowiednie wielomiany p_i i obliczyć te liczby[21].

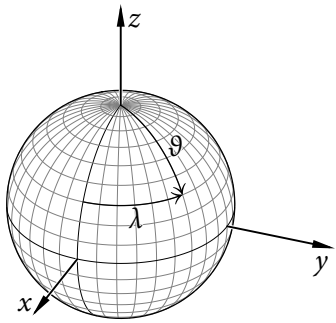
Do obliczenia potrzebnych dalej całek użyjemy **współrzędnych sferycznych** pokazanych na rysunku 28.5. Współrzędna λ punktu sfery S (wektora jednostkowego) jest długością geograficzną (wschodnią), a współrzędna ϑ jest **koszerokością**; jest to miara kąta między danym punktem a wersorem osi z — a więc ma wartość 0 na „biegunie północnym", $\frac{\pi}{2}$ na „równiku" oraz π na „biegunie południowym". Sfera ma parametryzację

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}(\lambda, \vartheta) = \begin{bmatrix} \cos\lambda \sin\vartheta \\ \sin\lambda \sin\vartheta \\ \cos\vartheta \end{bmatrix}, \quad \lambda \in [0, 2\pi),\ \vartheta \in [0, \pi].$$

[19] Na widok takiego zastosowania swojego twierdzenia Pitagoras zdziwiłby się i byłby bardzo zadowolony.

[20] Funkcje harmoniczne określone w przestrzeni $\mathbb{R}^3$ po ograniczeniu dziedziny do sfery jednostkowej są nazywane **harmonikami sferycznymi**. Ma to związek z wymyśloną przez starożytnych Greków harmonią sfer, choć nie bardzo wiem, jaki …

[21] Funkcja $I(\boldsymbol{n})$ jest wektorowa; każdą z jej składowych, r, g, b, przybliżymy osobnym wielomianem drugiego stopnia, zatem trzeba będzie znaleźć 27 liczb. Otrzymamy w ten sposób wielomianową funkcję wektorową opisującą irradiancję powierzchni oświetlonej światłem kolorowym.



Rysunek 28.5. Współrzędne sferyczne

Potrzebne będą całki z wielomianów $x^i y^j z^k$ dla $i + j + k \leqslant 4$. Miara dS zawierającego punkt (λ, ϑ) wycinka sfery jednostkowej odpowiadającego bliskim zera przyrostom dλ i dϑ współrzędnych sferycznych jest równa $\sin\vartheta\,\mathrm{d}\vartheta\,\mathrm{d}\lambda$. Stąd

$$\int_S x^i y^j z^k\,\mathrm{d}S = \int_{\lambda=0}^{2\pi}\int_{\vartheta=0}^{\pi}(\cos\lambda\sin\vartheta)^i(\sin\lambda\sin\vartheta)^j\cos^k\vartheta\sin\vartheta\,\mathrm{d}\vartheta\,\mathrm{d}\lambda = A_{ij}B_{k,i+j+1},$$

gdzie $\quad A_{ij} = \displaystyle\int_0^{2\pi}\cos^i\lambda\sin^j\lambda\,\mathrm{d}\lambda, \quad B_{kl} = \int_0^{\pi}\cos^k\vartheta\sin^l\vartheta\,\mathrm{d}\vartheta.$

Obliczanie całek A_{ij} i B_{kl} jest dosyć żmudnym ćwiczeniem z działań na funkcjach trygonometrycznych, którego wyniki są następujące: dla każdego i, j jest $A_{ij} = A_{ji}$. Jeśli i lub j jest nieparzyste, to $A_{ij} = 0$, a jeśli k jest nieparzyste, to $B_{kl} = 0$. Pozostałe całki wystarczające do obliczenia iloczynów skalarnych wielomianów stopnia co najwyżej 2 to

$$A_{00} = 2\pi, \quad A_{20} = A_{02} = \pi, \quad A_{40} = A_{04} = \frac{3}{4}\pi, \quad A_{22} = \frac{\pi}{4},$$
$$B_{01} = 2, \quad B_{03} = \frac{4}{3}, \quad B_{05} = \frac{16}{15}, \quad B_{21} = \frac{2}{3}, \quad B_{23} = \frac{4}{15}, \quad B_{41} = \frac{2}{5}.$$

Wielomiany

$$p_0 = 1, \quad p_1 = x, \quad p_2 = y, \quad p_3 = z, \quad p_4 = xy, \quad p_5 = yz, \quad p_6 = zx,$$
$$p_7 = x^2 - y^2, \quad p_8 = 2z^2 - x^2 - y^2$$

są funkcjami harmonicznymi; znając podane wyżej całki, można już łatwo sprawdzić, że baza składająca się z tych wielomianów jest ortogonalna[22], a kwadraty ich norm są równe

$$\|p_0\|^2 = 4\pi, \quad \|p_1\|^2 = \|p_2\|^2 = \|p_3\|^2 = \frac{4}{3}\pi, \quad \|p_4\|^2 = \|p_5\|^2 = \|p_6\|^2 = \frac{4}{15}\pi,$$
$$\|p_7\|^2 = \frac{16}{15}\pi, \quad \|p_8\|^2 = \frac{16}{5}\pi.$$

[22] Ramamoorthi i Hanrahan w [51] użyli **bazy ortonormalnej**, czyli bazy ortogonalnej, której elementy mają długość 1; na przykład zamiast funkcji stałej 1 (wielomianu p_0) jest w niej funkcja stała równa $1/\sqrt{4\pi}$. Brak normalizacji nie wpływa na wynik, tj. końcowe obrazy, a zmniejsza (odrobinę) koszt algorytmu.

Użycie funkcji wielomianowej drugiego stopnia zamiast tekstury do reprezentowania oświetlenia obiektów przez otoczenie daje sporą oszczędność pamięci, bo tablice tekseli muszą pomieścić $6 \cdot 64 \cdot 64 \cdot 3 = 73728$ liczb, podczas gdy współczynników wielomianów (dla trzech składowych koloru) jest w sumie 27. Do ich znalezienia zatrudnimy procedurę z listingu 28.6, wywołującą szader obliczeniowy z listingu 28.7. Jego zadaniem jest obliczenie tych współczynników za pomocą kwadratury przybliżającej całkę we wzorze (28.4), co jest realizowane w trzech etapach. W pierwszym etapie, dla wszystkich wielomianów p_i i składowych r, g, b irradiancji I, szader oblicza i zapisuje w pomocniczej tablicy składniki kwadratur, w drugim obliczane są sumy tych składników, a w trzecim szader dzieli te sumy przez kwadraty norm wielomianów p_i i zapisuje wynik (liczby $\langle p_i, I \rangle$) w docelowym miejscu.

Procedura `ComputeIrradiancePoly` w liniach 6–13 tworzy dwa bufory, z których pierwszy jest miejscem na końcowy wynik, a drugi będzie zawierał tablicę pomocniczą. W obliczeniu uwzględnimy wszystkie teksele tekstury irradiancji, a zatem tablica ta musi pomieścić $27 \cdot 6 \cdot 64 \cdot 64 = 663552$ liczby typu `float`, czyli zajmie dość dużo miejsca w pamięci GPU, ale jest potrzebna tylko „na chwilę", aby umożliwić obliczenia równoległe. Po ich zakończeniu można też zlikwidować teksturę irradiancji, aby zwolnić zajmowaną przez nią pamięć.

Aby trochę przyspieszyć obliczenia, szader uzyska dostęp do tekstury irradiancji za pośrednictwem dodatkowego ewaluatora. Listing 28.6 ilustruje sposób używania takich ewaluatorów, zastępujących ewaluatory „wbudowane" w teksturę. Ewaluator jest tworzony przez procedurę wywołaną w linii 14, a likwidowany w linii 32. Instrukcje w liniach 15 i 16 nadają parametrom ewaluatora takie wartości, aby wywoływana przez szader funkcja `texture` podawała wartości tekseli bez straty czasu na obliczenia interpolacji. Mając „gotowy" ewaluator, wybieramy (w linii 17) punkt dowiązania tekstury, przywiązujemy (w linii 18) teksturę do celu, a potem przywiązujemy ewaluator do tego samego punktu dowiązania, przy czym punkt dowiązania tekstury wybieramy, podając jako parametr procedury `glActiveTexture` jego nazwę symboliczną (np. `GL_TEXTURE0`), a procedura `glBindSampler` ma otrzymać numer tego punktu (np. `0`). Przed zlikwidowaniem ewaluatora trzeba go „odwiązać", co tu robi instrukcja w linii 31.

Instrukcja w linii 20 wybiera program zbudowany z szadera z listingu 28.7. Wyborem realizowanego przez szader etapu obliczeń steruje zmienna jednolita `n`, której położenie ma numer `0`. Jeśli zmienna ta ma wartość `0`, to szader wywołuje procedurę `Integrate`, która oblicza składniki kwadratur. W tym etapie globalna grupa robocza ma wymiary $64 \times 64 \times 6$; każdy wątek szadera ma za zadanie przetworzyć dane z jednego teksela.

Zbiór całkowania dla iloczynu skalarnego określonego wzorem (28.4), czyli sfera S, jest wpisany w kostkę standardową $[-1, 1]^3$, której ściany podzielimy na kwadraciki odpowiadające tekselom. Obraz takiego kwadracika w rzucie środkowym na sferę jest wycinkiem sfery, którego pole pomnożymy przez wartość funkcji podcałkowej w punkcie będącym rzutem środka kwadracika. Suma wszystkich tych iloczynów przybliża wartość całki.

Pozostawiam Czytelnikom wykazanie, że rzut środkowy na sferę S położonego na ścianie kostki standardowej kwadracika o polu A ma (z tym lepszym przybliżeniem, im mniejszy jest kwadracik) pole $A \cos^3 \alpha$, gdzie α oznacza kąt między wektorem normalnym $\boldsymbol{e}$ ściany kostki a wektorem $\boldsymbol{l}$ odpowiadającym środkowi kwadracika. I tak na przykład dla położonego

Listing 28.6. Procedura obliczania współczynników wielomianu opisującego irradiancję

```C
void ComputeIrradiancePoly ( GLuint prog_id, GLuint txt, GLuint *irrpoly )
{
  GLuint ibuf[2], samp;
  GLint  n;

  glGenBuffers ( 2, ibuf );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, ibuf[0] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER, 27*sizeof(GLfloat),
                 NULL, GL_DYNAMIC_DRAW );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, ibuf[1] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER,
                 6*IBLTXTSIZE*IBLTXTSIZE*27*sizeof(GLfloat), NULL,
                 GL_DYNAMIC_DRAW );
  glGenSamplers ( 1, &samp );
  glSamplerParameteri ( samp,  GL_TEXTURE_MAG_FILTER, GL_NEAREST );
  glSamplerParameteri ( samp,  GL_TEXTURE_MIN_FILTER, GL_NEAREST );
  glActiveTexture ( GL_TEXTURE0 );
  glBindTexture ( GL_TEXTURE_CUBE_MAP, txt );
  glBindSampler ( 0, samp );
  glUseProgram ( prog_id );
  glUniform1i ( 0, 0 );      /* uniform n = 0; */
  COMPUTE ( IBLTXTSIZE, IBLTXTSIZE, 6 )
  for ( n = 6*IBLTXTSIZE*IBLTXTSIZE; n > 1; n = (n+1)/2 ) {
    glUniform1i ( 0, n );    /* uniform n = n; */
    COMPUTE ( n/2, 27, 1 );
  }
  glUniform1i ( 0, 1 );      /* uniform n = 1; */
  COMPUTE ( 9, 3, 1 );
  glUseProgram ( 0 );
  glDeleteBuffers ( 1, &ibuf[1] );
  glBindSampler ( 0, 0 );
  glDeleteSamplers ( 1, &samp );
  glBindTexture ( GL_TEXTURE_CUBE_MAP, 0 );
  *irrpoly = ibuf[0];
  ExitIfGLError ( "ComputeIrradiancePoly" );
} /*ComputeIrradiancePoly*/
```

w płaszczyźnie $z = \pm 1$ kwadracika o środku $\boldsymbol{l} = (x, y, \pm 1)$ jest $\operatorname{tg}\alpha = \sqrt{x^2 + y^2}$, co umożliwia obliczenie $\cos\alpha = 1/\sqrt{x^2 + y^2 + 1}$. Błąd przybliżenia pól obrazów na sferze kwadracików o boku $\frac{1}{32}$ (odpowiadających tekselom tekstury kostkowej $6\times64\times64$) nie przekracza 0.006%, co w naszym zastosowaniu jest akceptowalne.

Na podstawie trójki liczb określających miejsce wątku w grupie roboczej, z których pierwsze dwie wyznaczają położenie kwadracika na ścianie kostki, a trzecia wybiera ścianę, szader w liniach 21 i 22 oblicza dwie z trzech współrzędnych środka kwadracika; trzecia współrzędna

jest równa 1 albo −1. Wartość przypisana zmiennej s w linii 23 jest odwrotnością kwadratu kosinusa α. W linii 24 jest obliczane wyrażenie $A\cos^3\alpha$.

W liniach 25–33 jest obliczany wektor jednostkowy $\boldsymbol{w} = \boldsymbol{l}/\|\boldsymbol{l}\|$ (czyli punkt sfery S) wycelowany w środek kwadracika. Wartość tekstury irradiancji dla tego wektora (czyli wartość teksela) jest w linii 34 mnożona przez $A\cos^3\alpha$. Trzeba ją jeszcze pomnożyć przez wartości wielomianów $p_0, \ldots, p_8$ i zapamiętać iloczyny w tablicy.

Z uwagi na to, że (także w układzie std430) następuje wyrównanie każdego elementu tablicy typu vec3 do wielkości zmiennej typu vec4, przez co te elementy są rozmieszczane z czterobajtowymi odstępami, a trzeba przechować ich wiele, zdecydowałem, że elementy tablic używanych przez szader będą typu float. Procedura StoreVec, której parametrami są indeks do tablicy i wektor, wpisuje do tablicy pomocniczej współrzędne tego wektora na trzech kolejnych miejscach. Instrukcje w liniach 36–44 mnożą wartość zmiennej ir przez wartości wielomianów $p_0, \ldots, p_8$ i zapamiętują obliczone wektory.

Listing 28.7. Szader obliczający współczynniki wielomianu przybliżającego irradiancję

GLSL

```
1: #version 450 core
2:
3: layout(local_size_x=1) in;
4:
5: layout(binding=0) uniform samplerCube IrradianceTxt;
6:
7: layout(binding=0,std430) buffer Irrad { float a[27]; } irrad;
8: layout(binding=1,std430) buffer Aux { float t[]; } aux;
9:
10: layout(location=0) uniform int n;
11:
12: void StoreVec ( uint ind, vec3 v )
13: { aux.t[ind] = v.r;  aux.t[ind+1] = v.g;  aux.t[ind+2] = v.b; }
14:
15: void Integrate ( uvec3 ti, uint size )
16: {
17:   float a, b, s;
18:   vec3  w, ir;
19:   uint  ind;
20:
21:   a = 2.0*((float(ti.x)+0.5)/float(size))-1.0;
22:   b = 2.0*((float(ti.y)+0.5)/float(size))-1.0;
23:   s = a*a + b*b + 1.0;
24:   s = 4.0/float(size*size) * inversesqrt ( s ) / s;
25:   switch ( ti.z ) {
26: case 0: w = vec3 (  1.0, a, b );  break;
27: case 1: w = vec3 ( -1.0, a, b );  break;
28: case 2: w = vec3 ( a,  1.0, b );  break;
29: case 3: w = vec3 ( a, -1.0, b );  break;
30: case 4: w = vec3 ( a, b,  1.0 );  break;
```

```
case 5: w = vec3 ( a, b, -1.0 );  break;
  }
  w = normalize ( w );
  ir = s*texture ( IrradianceTxt, w ).rgb;
  ind = 27*((ti.z*size + ti.y)*size + ti.x);
  StoreVec ( ind, ir );
  StoreVec ( ind+3, ir * w.x );
  StoreVec ( ind+6, ir * w.y );
  StoreVec ( ind+9, ir * w.z );
  StoreVec ( ind+12, ir * (w.x*w.y) );
  StoreVec ( ind+15, ir * (w.y*w.z) );
  StoreVec ( ind+18, ir * (w.z*w.x) );
  StoreVec ( ind+21, ir * ((w.x+w.y)*(w.x-w.y)) );
  StoreVec ( ind+24, ir * ((w.z+w.x)*(w.z-w.x) + (w.z+w.y)*(w.z-w.y)) );
} /*Integrate*/

void SumUp ( uint i, uint k )
{
  uint j;

  if ( (j = i+(n+1)/2) < n )
    aux.t[27*i+k] += aux.t[27*j+k];
} /*SumUp*/

void FinalComp ( uint i, uint k )
{
  const float ipnorm[9] =
    {0.079577472, 0.23873241, 0.23873241, 0.23873241,
     1.1936621, 1.1936621, 1.1936621, 0.29841552, 0.099471839};

  irrad.a[3*i+k] = ipnorm[i] * aux.t[3*i+k];
} /*FinalComp*/

void main ( void )
{
  switch ( n ) {
case 0:
    Integrate ( gl_GlobalInvocationID, gl_NumWorkGroups.x );
    break;
default:
    SumUp ( gl_GlobalInvocationID.x, gl_GlobalInvocationID.y );
    break;
case 1:
    FinalComp ( gl_GlobalInvocationID.x, gl_GlobalInvocationID.y );
    break;
  }
} /*main*/
```

Drugi etap obliczeń jest wykonywany za pomocą pierwszego algorytmu sumowania opisanego w podrozdziale G.1. Suma n składników jest obliczana w $\lceil \log_2 n \rceil$ krokach, przy czym w każdym kroku, dla danych m składników, „sumatory" (tj. wątki szadera) obliczają jednocześnie sumy $\lfloor m/2 \rfloor$ par składników, zmniejszając o tyle liczbę składników pozostających do zsumowania. Suma $6 \cdot 64 \cdot 64$ składników jest obliczana w 15 krokach.

Pierwszy wymiar grupy roboczej szadera w drugim etapie jest liczbą sumowanych par, a drugi wymiar jest stały, równy 27, bo tyle sum przybliżających całki trzeba obliczyć. W każdym kroku sumowania wartość zmiennej jednolitej `n` jest liczbą składników każdej z 27 sum, przy czym w pierwszym kroku jest to liczba tekseli, a w kolejnych krokach maleje. Sumy są obliczone (i zajmują początek tablicy pomocniczej), gdy zmienna `n` ma wartość 1.

Ostatni etap jest wykonywany przez procedurę `FinalComp`, którą szader wywołuje po nadaniu zmiennej jednolitej `n` wartości 1. Grupa robocza ma wtedy wymiary $9 \times 3 \times 1$. Pierwszy indeks wątku w grupie jest numerem obliczanego (wektorowego) współczynnika wielomianu w bazie $\{p_0, \ldots, p_8\}$, a drugi indeks jest numerem współrzędnej tego wektora. Liczby podane w tablicy `ipnorm` są odwrotnościami kwadratów norm elementów bazy (zobacz s. 753), na przykład `ipnorm[0]` to $\frac{1}{4\pi}$.

Listing 28.8 przedstawia podprogram, który w obliczeniu irradiancji powierzchni o wektorze normalnym $\boldsymbol{n}$ zastępuje funkcję `texture`, wywołaną w linii 13 na listingu 28.5. Należy pamiętać, aby numer punktu dowiązania bloku ze współczynnikami obliczonymi przez szader z listingu 28.7 był inny niż numery wybrane dla bloków z pozostałymi danymi, na przykład opisującymi płaty Béziera.

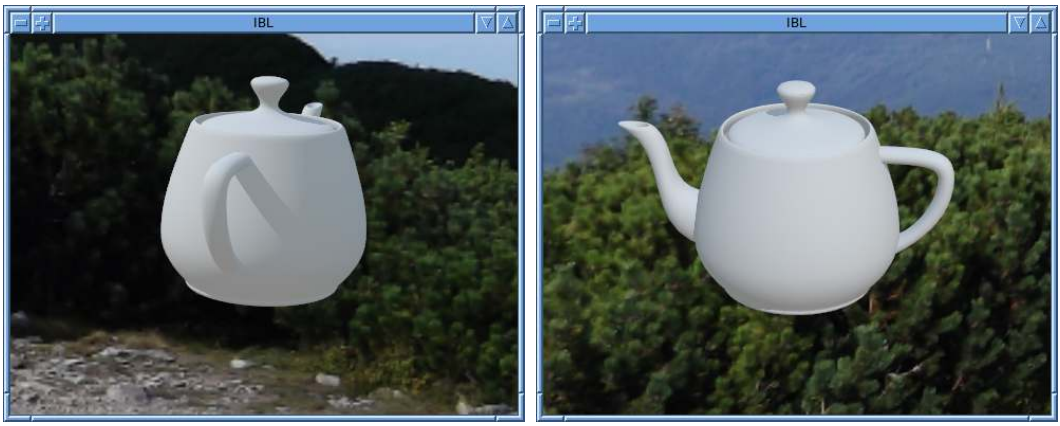
Listing 28.8. Procedura obliczania irradiancji reprezentowanej przez wielomian

GLSL

```
1: layout(binding=4,std430) buffer Irrad { float a[27]; } irrad;
2:
3: #define lpc(i) vec3 ( irrad.a[i], irrad.a[i+1], irrad.a[i+2] )
4:
5: vec3 LightPoly ( vec3 n )
6: {
7:   return lpc(0) + lpc(3)*n.x + lpc(6)*n.y + lpc(9)*n.z +
8:          lpc(12)*(n.x*n.y) + lpc(15)*(n.y*n.z) + lpc(18)*(n.z*n.x) +
9:          lpc(21)*((n.x+n.y)*(n.x-n.y)) +
10:         lpc(24)*((n.z+n.x)*(n.z-n.x)+(n.z+n.y)*(n.z-n.y));
11: } /*LightPoly*/
```

Rysunek 28.6 przedstawia wynik oświetlenia obiektu lambertowskiego przez światło dochodzące z otoczenia reprezentowanego przez tekstury z rysunku 28.4. W tym przykładzie między obrazami otrzymanymi za pomocą tekstury irradiancji a tymi wykonanymi przy użyciu wielomianowego przybliżenia irradiancji nie ma żadnej zauważalnej różnicy.

Oświetlenie rozproszone na przykład w pomieszczeniu z oknami istotnie zależy od miejsca, w którym znajduje się rysowany obiekt. Reprezentacja irradiancji za pomocą wielomianu zajmuje bardzo mało pamięci, co ułatwia realizację następującego pomysłu: aby uwzględnić podczas rysowania oświetlenie zależne od położenia, można wybrać pewne punkty w po-



Rysunek 28.6. Czajnik lambertowski oświetlony przez obraz i przez (punktowe) Słońce

mieszczeniu (np. rozmieszczone w wierzchołkach regularnej trójwymiarowej siatki), dla każdego z tych punktów znaleźć współczynniki odpowiedniego wielomianu i zapamiętać je w tablicy. Szader fragmentów na podstawie położenia $\boldsymbol{p}$ otrzymanego fragmentu w pomieszczeniu może znaleźć w tablicy otaczające punkty siatki, dla ustalonego wektora normalnego $\boldsymbol{n}$ obliczyć wartości wielomianów w tych punktach, a następnie dokonać interpolacji tych wartości w celu otrzymania irradiancji powierzchni w punkcie $\boldsymbol{p}$. Zrealizowanie tego pomysłu pozostawiam inwencji Czytelników.

28.4. Modele oświetlenia oparte na prawach fizyki

Pierwszym krokiem do otrzymania realistycznego obrazu dowolnego obiektu jest wybranie dwukierunkowej funkcji odbicia i załamania światła opisującej własności materiału i powierzchni tego obiektu. Rozważane w tym podrozdziale modele oświetlenia nieźle opisują obiekty o powierzchni chropowatej wykonane m.in. z ceramiki, metalu, szkła, niektórych minerałów i tworzyw sztucznych, ale mogą nie wystarczyć do zadowalającego opisu innych minerałów, materiałów sypkich (np. piasku), powierzchni pokrytych lakierem, tkanin, futra, drewna, liści i ludzkiej skóry[23].

28.4.1. Odbijanie światła przez mikrościanki

Opisane niżej modele oświetlenia są oparte na założeniu, że chropowata powierzchnia składa się z niewidocznych gołym okiem **mikrościanek**. Każda mikrościanka może zachowywać się jak powierzchnia lambertowska, od której światło odbija się z jednakową radiancją we

[23] Na wygląd oświetlonej skóry, a także powierzchni mleka, mlecznego szkła, miodu, porcelany, białego marmuru, stearyny i innych nie całkiem nieprzezroczystych substancji bardzo duży wpływ ma tzw. **dwukierunkowa funkcja podpowierzchniowego rozpraszania światła** (*bidirectional subsurface scattering distribution function*, *BSSDF*), której opisu nie zamieszczam w tej książce. Zachęcam Czytelników do samodzielnych poszukiwań.

wszystkich kierunkach, lub jak idealne lustro działające zgodnie z zasadami optyki geometrycznej: foton zostaje odbity w kierunku określonym przez wektor $-\boldsymbol{l}$, w którego kierunku poruszał się przed odbiciem, i przez wektor normalny $\boldsymbol{h}$ mikrościanki[24]. Światło na mikrościance może też ulec załamaniu zgodnie z opisem w podrozdziale A.1.

Przyjmuje się, że odbiciu rozproszonemu ulegają fotony, które przeniknęły przez powierzchnię, załamując się na mikrościance, po czym uległy nawet wielokrotnym odbiciom na przykład od cząstek pigmentu znajdujących się pod tą powierzchnią i ponownie przeniknęły przez powierzchnię (w tej samej lub innej mikrościance), ulatując w przypadkowych kierunkach. Z tego powodu pigment ma zasadniczy wpływ na kolor światła odbitego w sposób rozproszony i nie ma wpływu na kolor odblasków światła na powierzchni.

Dwukierunkowa funkcja odbicia światła (BRDF) w rozważanych tu modelach oświetlenia ma postać

$$\rho_\mathrm{r} = k_\mathrm{d}\rho_\mathrm{d} + k_\mathrm{s}\rho_\mathrm{s}. \tag{28.6}$$

Pierwszy składnik odpowiada za światło odbite w sposób rozproszony (*diffuse reflection*), a drugi powoduje powstanie odblasków (*specular reflection*); nieujemne stałe k_d i k_s o sumie równej 1 określają proporcję świateł odbitych tymi dwoma sposobami.

Na początek rozważmy odblaski. Przyjmuje się, że są one skutkiem odbicia każdego fotonu przez tylko jedną mikrościankę. Funkcja ρ_s jest określona wzorem

$$\rho_\mathrm{s} = \frac{DGF_\lambda}{4\langle \boldsymbol{l}, \boldsymbol{n}\rangle\langle \boldsymbol{v}, \boldsymbol{n}\rangle}, \tag{28.7}$$

w którym oprócz opisanych wcześniej wektorów jednostkowych $\boldsymbol{n}$, $\boldsymbol{l}$ i $\boldsymbol{v}$ występują trzy czynniki zależne od chropowatości powierzchni i od własności optycznych rozgraniczanych przez nią ośrodków.

Czynnik D (*normal distribution function*, *NDF*) opisuje statystyczny rozkład kierunków wektorów normalnych mikrościanek; jego wartość dla ustalonych wektorów $\boldsymbol{l}$ i $\boldsymbol{v}$ określa, jak wiele mikrościanek ma jednostkowy wektor normalny $\boldsymbol{h}$ o kierunku wektora $\boldsymbol{l}+\boldsymbol{v}$ — bo właśnie te mikrościanki należy wziąć pod uwagę. Kąt między wektorem $\boldsymbol{h}$ a wektorem normalnym powierzchni $\boldsymbol{n}$ oznaczymy symbolem $\vartheta_{\boldsymbol{h}}$; jest $\cos\vartheta_{\boldsymbol{h}} = \langle \boldsymbol{h}, \boldsymbol{n}\rangle$. Zakłada się, że dowolna prosta o kierunku wektora $\boldsymbol{n}$ przecina powierzchnię tylko w jednym punkcie. Wtedy te kosinusy dla wszystkich mikrościanek mają ten sam znak (dalej przyjmiemy, że dodatni, a więc kąty $\vartheta_{\boldsymbol{h}}$ są ostre). Ponieważ cała powierzchnia składa się z mikrościanek (i tylko z nich), musi być spełniony warunek

$$\int_{\boldsymbol{h}\in S_{\boldsymbol{n}+}} D(\boldsymbol{h}) \cos\vartheta_{\boldsymbol{h}}\, \mathrm{d}S = 1. \tag{28.8}$$

[24] Mikrościanki działają jak lustra, gdy ich średnice są co najmniej kilkakrotnie większe od długości fali świetlnej.

Przyjęty w klasycznym **modelu Cooka i Torrance'a** [46] rozkład kierunków wektorów normalnych mikrościanek jest opisany wzorem

$$D_{\mathrm{BS}}(\boldsymbol{h}) = \frac{\mathrm{e}^{-(\mathrm{tg}^2\,\vartheta_{\boldsymbol{h}})/m^2}}{\pi m^2 \cos^4\vartheta_{\boldsymbol{h}}}. \tag{28.9}$$

Opisuje on tzw. **rozkład Beckmanna-Spizzichino**. Chropowatość powierzchni rośnie ze wzrostem parametru m. Powierzchnia z takim rozkładem mikrościanek jest izotropowa. Efekt anizotropii (spowodowanej np. obecnością rys układających się w określonym kierunku) można otrzymać, przyjmując rozkład z dwoma parametrami, m_u, m_v:

$$D_{\mathrm{BSa}}(\boldsymbol{h}) = \frac{\mathrm{e}^{-(c^2/m_u^2+s^2/m_v^2)\,\mathrm{tg}^2\,\vartheta_{\boldsymbol{h}}}}{\pi m_u m_v \cos^4\vartheta_{\boldsymbol{h}}}. \tag{28.10}$$

We wzorze tym c i s to odpowiednio kosinus i sinus kąta między rzutem wektora $\boldsymbol{h}$ na płaszczyznę styczną do powierzchni a ustalonym wektorem $\boldsymbol{u}$ w tej płaszczyźnie. Dla powierzchni parametrycznej wektor ten może być dowolnie wybraną kombinacją liniową wektorów pochodnych cząstkowych parametryzacji (zobacz p. 21.5.4).

Często w grafice jest też używany **rozkład Trowbridge'a-Reitza**, określony wzorem

$$D_{\mathrm{TR}}(\boldsymbol{h}) = \frac{m^2}{\pi\left(1 + (m^2-1)\cos^2\vartheta_{\boldsymbol{h}}\right)^2} \tag{28.11}$$

z parametrem m, który określa chropowatość powierzchni. Anizotropowe uogólnienie tego rozkładu opisuje wzór

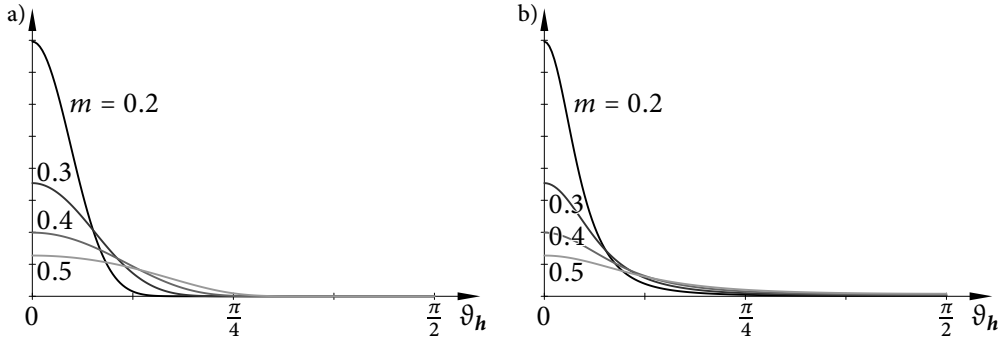
$$D_{\mathrm{TRa}}(\boldsymbol{h}) = \frac{1}{\pi m_u m_v\left(1 + \mathrm{tg}^2\,\vartheta_{\boldsymbol{h}}(c^2/m_u^2 + s^2/m_v^2)\right)^2 \cos^4\vartheta_{\boldsymbol{h}}}. \tag{28.12}$$

Symbole c i s oznaczają to samo co we wzorze opisującym funkcję D_{BSa}. Podstawiając $m_u = m_v = m$, otrzymamy izotropowy rozkład mikrościanek opisany wzorem (28.11). Ze wzrostem kąta między wektorami $\boldsymbol{h}$ a $\boldsymbol{n}$ rozkład Trowbridge'a-Reitza początkowo maleje szybciej, a potem zanika wolniej niż rozkład Beckmanna-Spizzichino (rys. 28.7), dając inny charakter chropowatości powierzchni.

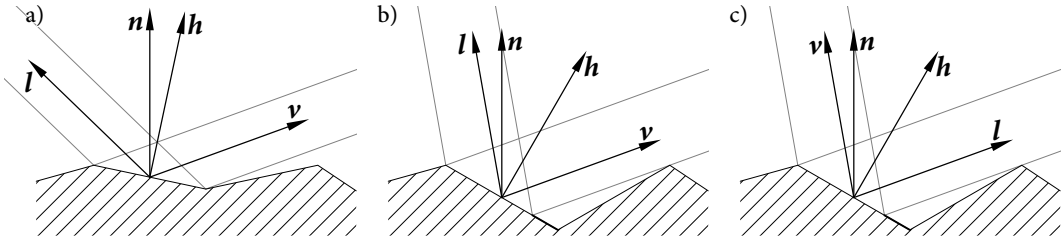
Powierzchnie przedmiotów mogą mieć więcej niż jedną rodzinę mikrościanek, które powstały na przykład w kolejnych etapach obróbki mechanicznej — frezowania, szlifowania, polerowania i porysowania. Każda taka rodzina ma inny rozkład kierunków wektorów normalnych; aby otrzymać obraz takiego przedmiotu, można przyjąć funkcję ρ_{s} z czynnikiem D opisanym wzorem

$$D(\boldsymbol{h}) = \sum_{i=1}^{n} a_i D_i(\boldsymbol{h}),$$

w którym występują dodatnie współczynniki $a_1, \ldots, a_n$ o sumie równej 1 i rozkłady $D_1, \ldots, D_n$ poszczególnych rodzin mikrościanek.



Rysunek 28.7. Rozkłady Beckmanna-Spizzichino (a) i Trowbridge'a-Reitza (b) z różnymi wartościami parametru chropowatości



Rysunek 28.8. Zasłanianie mikrościanek: a) cała mikrościanka jest oświetlona i widoczna, b) mikrościanka jest częściowo widoczna, c) mikrościanka jest częściowo oświetlona

Czynnik G (*geometric attenuation factor*, *GAF*) odpowiada za częściowe zasłanianie jednych mikrościanek przez drugie — przed obserwatorem, tj. widzianych z kierunku wektora $\boldsymbol{v}$, lub przed światłem, tj. z kierunku wektora $\boldsymbol{l}$. Znalezienie tego czynnika wymaga przyjęcia pewnej hipotezy na temat wzajemnego usytuowania mikrościanek. Najprostsza hipoteza [45] zakłada, że pary sąsiadujących mikrościanek tworzą **symetryczne rowki**, tzn. mikrościanki w każdej takiej parze mają jednakową wielkość, a dwusieczna kąta między ich wektorami normalnymi ma kierunek wektora $\boldsymbol{n}$ (rys. 28.8). Nie ma to wiele wspólnego z rzeczywistością, ale otrzymany na tej podstawie wzór jest użyteczny w praktyce. Ma on postać

$$G = \min\left\{1, \frac{2\langle \boldsymbol{h}, \boldsymbol{n}\rangle\langle \boldsymbol{v}, \boldsymbol{n}\rangle}{\langle \boldsymbol{v}, \boldsymbol{h}\rangle}, \frac{2\langle \boldsymbol{h}, \boldsymbol{n}\rangle\langle \boldsymbol{l}, \boldsymbol{n}\rangle}{\langle \boldsymbol{l}, \boldsymbol{h}\rangle}\right\}. \tag{28.13}$$

Czynnik Fresnela F_λ opisuje, jaka część fotonów padających na idealnie gładkie lustro ulega odbiciu; pozostałe fotony ulegają załamaniu, tj. przenikają przez powierzchnię lustra i, zależnie od przezroczystości ośrodka, do którego wpadły, przelatują przezeń lub zostają pochłonięte. Czynnik Fresnela zależy od kąta między wektorami $\boldsymbol{l}$ a $\boldsymbol{h}$ i od ilorazu współczynników załamania światła ośrodków dla fali świetlnej o długości λ.

Obiekty, których obrazy chcemy otrzymywać, są zbudowane z przewodników, półprzewodników albo dielektryków. W przewodnikach (i półprzewodnikach) światło indukuje

prąd, który natychmiast zamienia się w ciepło, dlatego przezroczyste mogą być tylko dielektryki lub *bardzo* cienkie warstwy przewodników. Okazuje się, że współczynniki załamania światła przewodników są zespolone; ich części urojone opisują szybkość tłumienia światła i w obliczeniach dwukierunkowej funkcji odbicia można je pominąć.

Czynnik Fresnela dla światła niespolaryzowanego [46] można wyrazić wzorem

$$F_\lambda = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1+\frac{\big(c(g+c)-1\big)^2}{\big(c(g-c)+1\big)^2}\right),$$

w którym $c = \cos\vartheta_{\boldsymbol{hl}} = \langle \boldsymbol{l}, \boldsymbol{h}\rangle$, $g = \sqrt{(\eta_{\lambda,2}/\eta_{\lambda,1})^2 + c^2 - 1}$, a symbole $\eta_{\lambda,1}$ i $\eta_{\lambda,2}$ oznaczają współczynnik załamania światła ośrodka, przez który światło dochodzi do powierzchni obiektu, i współczynnik załamania światła materiału, z którego jest wykonany ten obiekt.

W praktyce czynniki Fresnela są zastępowane wyrażeniami łatwiejszymi do obliczenia. Symbolami $F_{\lambda,0}$ i $F_{\lambda,\pi/2}$ oznaczymy czynniki Fresnela dla światła padającego na mikrościankę prostopadle oraz stycznie do niej. Zauważmy, że w pierwszym przypadku $\boldsymbol{l} = \boldsymbol{h}$, skąd wynika, że $c = 1$ i $g = \eta_{\lambda,2}/\eta_{\lambda,1}$, a w drugim mamy $c = 0$ i $g = \sqrt{(\eta_{\lambda,2}/\eta_{\lambda,1})^2 - 1}$. Stąd

$$F_{\lambda,0} = \left(\frac{\eta_{\lambda,2}-\eta_{\lambda,1}}{\eta_{\lambda,2}+\eta_{\lambda,1}}\right)^2, \quad F_{\lambda,\pi/2} = 1.$$

Przybliżenie Schlicka [48], opisane wzorem

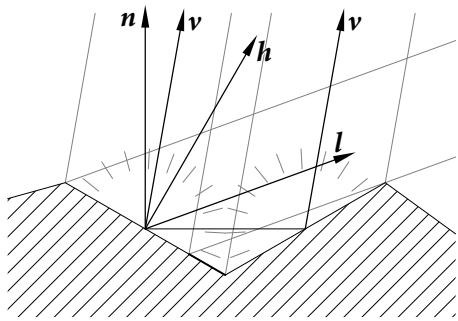
$$F_\lambda \approx F_{\lambda S} = F_{\lambda,0} + (F_{\lambda,\pi/2} - F_{\lambda,0})(1-\cos\vartheta_{\boldsymbol{hl}})^5 = F_{\lambda,0} + (1-F_{\lambda,0})(1-\langle \boldsymbol{l}, \boldsymbol{h}\rangle)^5, \qquad (28.14)$$

wymaga podania tylko funkcji $F_{\lambda,0}$, której argumentem jest długość fali świetlnej λ. Funkcję $F_{\lambda,0}$ zastąpimy wektorem $\boldsymbol{F}_{\lambda,0}$, którego trzy współrzędne są wartościami tej funkcji otrzymanymi z pomiarów współczynnika załamania światła czerwonego, zielonego i niebieskiego[25] przez badany materiał.

Zauważmy, że przestawienie wektorów $\boldsymbol{l}$ i $\boldsymbol{v}$ nie zmienia czynników D i G; również czynnik Fresnela nie zmienia wartości, bo kąty między wektorem $\boldsymbol{h}$ a wektorami $\boldsymbol{l}$ i $\boldsymbol{v}$ są równe. Dzięki temu funkcja ρ_s jest symetryczna, zgodnie z zasadą Helmholtza.

Opisująca rozproszone odbicie światła funkcja ρ_d może w najprostszym przypadku mieć tę samą wartość dla wszystkich wektorów $\boldsymbol{v}$ skierowanych „nad powierzchnię", ale to jest równoznaczne z przyjęciem, że powierzchnia ta jest *doskonale* gładka, jak gdyby wszystkie mikrościanki miały ten sam wektor normalny $\boldsymbol{h} = \boldsymbol{n}$, a przecież nie mają. Dlatego wygląd chropowatych powierzchni matowych może zauważalnie odbiegać od skutków użycia modelu Lamberta (zobacz np. Księżyc). **Model Orena i Nayara** [49] zakłada, że mikrościanki mają pewien ustalony rozkład kierunków wektorów normalnych i że odbicie światła w każdej mikrościance jest lambertowskie. Skutkiem obecności mikrościanek o różnych wektorach normalnych jest niejednakowa radiancja światła rozproszonego w różnych kierunkach.

[25] Do uzyskania obrazów o najwyższej jakości może jednak być potrzebna znacznie dokładniejsza reprezentacja widma światła oraz czynnika Fresnela, w postaci nawet kilkudziesięciu liczb odpowiadających różnym długościom fali świetlnej. Przejście od takiej reprezentacji do trzech liczb reprezentujących kolor piksela na ekranie jest końcowym krokiem obliczeń. Więcej wiadomości na ten temat można znaleźć w [35].



Rysunek 28.9. Odbicie rozproszone światła przez parę mikrościanek

Przyjęcie (podobnie jak w modelu Cooka i Torrance'a), że pary mikrościanek tworzą symetryczne rowki, umożliwiło obliczenie radiancji światła, które padając z kierunku wektora $\boldsymbol{l}$, zostaje odbite w kierunku wektora $\boldsymbol{v}$ przez jedną lub dwie mikrościanki z pary. W przykładzie na rysunku 28.9 obie mikrościanki są widoczne, przy czym bezpośrednio oświetlona jest tylko część jednej z nich. Bardziej szczegółowe rachunki można znaleźć w artykule [49]; ostatnim ich krokiem jest obliczenie całki względem rozkładu kierunków wektorów normalnych mikrościanek. Autorzy przyjęli rozkład normalny Gaussa, którego parametr σ określa chropowatość. Powierzchnia z tym rozkładem jest izotropowa. Ponieważ końcowy wynik opisujący dwukierunkową funkcję odbicia nie daje się przedstawić w postaci prostej formuły (do obliczenia całek trzeba było użyć kwadratur), twórcy modelu zaproponowali jego przybliżenie realizowane przez następujący wzór:

$$\rho_{\mathrm{d},1} = \frac{c_\lambda}{\pi}\left(C_1 + cC_2 \operatorname{tg}\beta + \big(1-|c|\big)C_3 \operatorname{tg}\frac{\alpha+\beta}{2}\right). \tag{28.15}$$

Występujące w nim symbole oznaczają

$$\alpha = \max\{\vartheta_{\boldsymbol{l}}, \vartheta_{\boldsymbol{v}}\}, \qquad \beta = \min\{\vartheta_{\boldsymbol{l}}, \vartheta_{\boldsymbol{v}}\},$$

$$C_1 = 1 - \frac{0.5\sigma^2}{\sigma^2+0.33}, \quad C_2 = \frac{0.45\sigma^2}{\sigma^2+0.09}d, \quad C_3 = \frac{0.125\sigma^2}{\sigma^2+0.09}\left(\frac{4\alpha\beta}{\pi^2}\right)^2,$$

$$c = \cos\varphi, \qquad d = \begin{cases} \sin\alpha & \text{jeśli } c \geqslant 0, \\ \sin\alpha - \left(\frac{2\beta}{\pi}\right)^3 & \text{w przeciwnym razie.} \end{cases}$$

Funkcja c_λ o wartościach w przedziale $[0,1]$ opisuje, jaki ułamek mocy światła o długości fali λ jest odbijany przez mikrościankę; podobnie jak czynnik Fresnela, zastąpimy ją przez wektor $\boldsymbol{c}_\lambda$ o trzech współrzędnych, r, g, b. Symbole $\vartheta_{\boldsymbol{l}}$ i $\vartheta_{\boldsymbol{v}}$ oznaczają kąty między wektorami $\boldsymbol{l}$ i $\boldsymbol{v}$ a wektorem $\boldsymbol{n}$: jest $\cos\vartheta_{\boldsymbol{l}} = \langle \boldsymbol{l}, \boldsymbol{n}\rangle$ i $\cos\vartheta_{\boldsymbol{v}} = \langle \boldsymbol{v}, \boldsymbol{n}\rangle$. Kąt φ jest mierzony między rzutami prostopadłymi wektorów $\boldsymbol{l}$ i $\boldsymbol{v}$ na płaszczyznę styczną do powierzchni. Jeśli oba te wektory

nie mają kierunku wektora $\boldsymbol{n}$, to

$$c = \frac{\langle \boldsymbol{l}, \boldsymbol{v} \rangle - \langle \boldsymbol{l}, \boldsymbol{n} \rangle \langle \boldsymbol{v}, \boldsymbol{n} \rangle}{\| \boldsymbol{l} - \langle \boldsymbol{l}, \boldsymbol{n} \rangle \boldsymbol{n} \| \| \boldsymbol{v} - \langle \boldsymbol{v}, \boldsymbol{n} \rangle \boldsymbol{n} \|}.$$

Jeśli wektor $\boldsymbol{l}$ lub $\boldsymbol{v}$ ma kierunek wektora $\boldsymbol{n}$, to $\beta = \operatorname{tg}\beta = C_3 = 0$, $d = \sin\alpha$ i nieokreślona liczba c jest nieistotna.

Funkcja $\rho_{\mathrm{d},1}$ jest składnikiem funkcji ρ_{d} opisującym pojedyncze odbicie światła w kierunku $\boldsymbol{v}$ od mikrościanek oświetlonych bezpośrednio. Uwzględnienie światła wysłanego w tym kierunku po dwóch kolejnych odbiciach (w obu mikrościankach tworzących każdą parę) wymaga dodania składnika

$$\rho_{\mathrm{d},2} = \frac{c_\lambda^2}{\pi} \frac{0.17\sigma^2}{\sigma^2 + 0.13} \left(1 - c\frac{4\beta^2}{\pi^2}\right), \tag{28.16}$$

przy czym dokładność przybliżenia przez ten wzór funkcji opisującej takie odbicia maleje ze wzrostem parametru σ. W praktyce często składnik ten bywa pomijany. Zauważmy, że podstawiając $\sigma = 0$, otrzymamy $C_1 = 1$, $C_2 = C_3 = \rho_{\mathrm{d},2} = 0$, wskutek czego powstanie lambertowski model odbicia światła.

Możemy sprawdzić, że przestawienie wektorów $\boldsymbol{l}$ i $\boldsymbol{v}$ w podanych wyżej wzorach nie zmienia wartości funkcji $\rho_{\mathrm{d},1}$ i $\rho_{\mathrm{d},2}$, a więc model Orena i Nayara jest zgodny z zasadą Helmholtza. Obliczenia przy jego użyciu są jednak dosyć kosztowne, dlatego w aplikacjach działających w czasie rzeczywistym (np. w grach) stosuje się go rzadziej niż model Lamberta.

Jeśli materiał jest przezroczysty (taki jak szkło), ale powierzchnia wykonanego z niego przedmiotu jest chropowata, to **dwukierunkowa funkcja przechodzenia światła** przez granicę ośrodków o współczynnikach załamania η_1 i η_2 (BTDF) jest opisana wzorem

$$\rho_{\mathrm{t}} = -\frac{DG(1 - F_\lambda)}{4\langle \boldsymbol{l}, \boldsymbol{n} \rangle \langle \boldsymbol{v}, \boldsymbol{n} \rangle} \frac{\eta_2^2}{\eta_1^2}, \tag{28.17}$$

w którym występują te same funkcje D, G i F_λ co we wzorze (28.7). Ponieważ czynnik Fresnela F_λ określa, jaka część fotonów odbija się od mikrościanki, czynnik $1 - F_\lambda$ opisuje fotony, które przez nią przechodzą. Czynniki D i G opisują rozkład kierunków wektorów normalnych mikrościanek i ich wzajemne zasłanianie. Aby obliczyć wartości wszystkich trzech funkcji, trzeba podać jednostkowy wektor normalny $\boldsymbol{h}$ mikrościanki, na której światło padające z kierunku $\boldsymbol{l}$ załamuje się w kierunku $\boldsymbol{v}$.

Dla załamania światła kąt między wektorami $\boldsymbol{l}$ a $\boldsymbol{v}$ jest rozwarty, czyli $\langle \boldsymbol{l}, \boldsymbol{v} \rangle < 0$; jeśli ta nierówność jest niespełniona, to $\rho_{\mathrm{t}} = 0$. W wyprowadzonym w podrozdziale A.1 wzorze (9.1) (w którym $\eta = \eta_1/\eta_2$) wektor $\boldsymbol{l}$ ma orientację przyjętą w specyfikacji GLSL-owej funkcji `refract`. Zgodnie z konwencją, w której wektor $\boldsymbol{l}$ jest zwrócony w stronę źródła światła, wzór ten, z wektorami $\boldsymbol{h}$ i $\boldsymbol{v}$ podstawionymi w miejsce $\boldsymbol{n}$ i $\boldsymbol{r}$, ma postać

$$\boldsymbol{v} = -\eta \boldsymbol{l} - \left(\sqrt{k} - \eta \langle \boldsymbol{l}, \boldsymbol{h} \rangle\right) \boldsymbol{h}.$$

Stąd, wiedząc, że wektor $\boldsymbol{h}$ jest jednostkowy, można go obliczyć, nie znając liczb k i $\langle \boldsymbol{l}, \boldsymbol{h} \rangle$:

$$\boldsymbol{h} = \frac{\pm 1}{\|\boldsymbol{v} + \eta \boldsymbol{l}\|}(\boldsymbol{v} + \eta \boldsymbol{l}) = \frac{\pm 1}{\|\eta_1 \boldsymbol{l} + \eta_2 \boldsymbol{v}\|}(\eta_1 \boldsymbol{l} + \eta_2 \boldsymbol{v}),$$

przy czym znak trzeba wybrać tak, aby było $\langle \boldsymbol{h}, \boldsymbol{n} \rangle > 0$. Dalej trzeba zbadać, czy znaki iloczynów skalarnych $\langle \boldsymbol{l}, \boldsymbol{h} \rangle$ i $\langle \boldsymbol{v}, \boldsymbol{h} \rangle$ są przeciwne, bo jeśli nie, to światło nie przechodzi przez mikrościankę, a więc także dla takiej pary wektorów $(\boldsymbol{l}, \boldsymbol{v})$ funkcja ρ_t ma wartość 0.

Do obliczenia czynnika G można użyć tej samej hipotezy, która umożliwiła jego otrzymanie dla odbicia światła, ale samo założenie, że mikrościanki tworzą symetryczne rowki, nie wyklucza zasłaniania pewnej części mikrościanki obserwowanej z kierunku $\boldsymbol{v}$ od światła padającego z kierunku $\boldsymbol{l}$ i jednoczesnego zasłaniania obserwatorowi (innej) jej części przez mikrościankę z sąsiedniej pary. Hipoteza o symetrycznych rowkach nie zakłada niczego o wielkości tej mikrościanki ani o kierunku jej wektora normalnego, a więc nie daje podstaw do obliczenia czynnika G. Ale jeśli przymknąć na to oko, to czynnik ten dla załamania światła może być opisany tym samym wzorem (28.13) co dla odbicia[26].

28.4.2. Implementacja oświetlenia przez źródła punktowe

Listing 28.9 przedstawia przykład realizacji w GLSL-u oświetlenia powierzchni przez *punktowe* źródła światła, z funkcją ρ_s określoną w modelu Cooka i Torrance'a oraz funkcją ρ_d przyjętą według Orena i Nayara. Blok z macierzami przekształceń i położeniem obserwatora i opisy źródeł światła są takie jak w aplikacjach opisanych w rozdziałach 10–26 i 32–36, choć obecnie wektor `direct` opisuje *moc* źródła światła. Zmieniony został opis materiału (w strukturze typu `Material`); teraz składa się on z liczb k_d, k_s i m oraz wektorów $\boldsymbol{c}_\lambda$ i $\boldsymbol{F}_{\lambda,0}$, których współrzędne r, g, b określają kolor materiału.

Listing 28.9. Szader fragmentów realizujący fizyczne modele oświetlenia przez źródła punktowe

GLSL

```
#version 450 core

#define PI 3.141592653

/* poniższe deklaracje jak na listingu 22.5 */
in FVertex { .... } In;
out vec4 out_Colour;
uniform TransBlock { .... } trb;
struct LSPar { .... };
uniform LSBlock { .... } light;
layout(binding=2) uniform sampler2DShadow shtex[MAX_NLIGHTS];
vec3 posDifference ( vec4 p, vec3 pos, out float dist ) { .... }
float attFactor ( vec3 att, float dist ) { .... }
float IsEnlighted ( int l ) { .... }
#define AGamma(colour) pow ( colour, vec3(256.0/563.0) )

```

[26] Zwracam uwagę, że dla załamania światła $|\langle \boldsymbol{v}, \boldsymbol{h} \rangle| \neq |\langle \boldsymbol{l}, \boldsymbol{h} \rangle|$.

```
17: struct Material {
18:     float kD, kS, m;
19:     vec3  cl, Fl0;
20:   };
21:
22: vec3 normal, tnormal;
23: Material mm;
24:
25: void GetMaterial ( vec2 tc ) { .... }
26:
27: vec3 OrenNayarRho ( vec3 n, vec3 v, vec3 l, float cnv, float cnl,
28:                     float s2, cl )
29: {
30:   float ca, cb, alpha, beta, api, bpi, tb, lv, c, d, C1, C2, C3, x, y;
31:
32:   if ( s2 == 0.0 )
33:     return cl/PI;
34:   C1 = 1.0-0.5*s2/(s2+0.33);
35:   if ( cnv > cnl ) { ca = cnl;  cb = cnv; }
36:               else { ca = cnv;  cb = cnl; }
37:   alpha = acos ( ca );  beta = acos ( cb );
38:   if ( cb < 1.0 ) {
39:     lv = dot ( l, v );
40:     c = (lv - ca*cb)/(length (l - cnl*n) * length ( v - cnv*n ));
41:     d = ca < 1.0 ? sqrt ( 1.0 - ca*ca ) : 0.0;
42:     if ( c >= 0.0 ) {
43:       bpi = (beta+beta)/PI;
44:       d -= bpi*bpi*bpi;
45:     }
46:     tb = sqrt ( 1.0 - cb*cb ) / cb;
47:     y = s2+0.09;
48:     C2 = 0.45*d*s2/y;
49:     api = (alpha+alpha)/PI;
50:     x = api*bpi;
51:     C3 = 0.125*s2*x*x/y;
52:     return cl*(C1 + c*C2*tb + (1.0-abs(c))*C3*tan(0.5*(alpha+beta)))/PI;
53:   }
54:   else
55:     return cl*C1/PI;
56: } /*OrenNayarRho*/
57:
58: vec3 CookTorranceRho ( vec3 n, vec3 v, vec3 l, float cnv, float cnl,
59:                        float m2, vec3 Fl0 )
60: {
61:   vec3  h, F;
62:   float cnh, c2nh, t2nh, cvh, D, g, G;
63:
```

```
64:   if ( m2 == 0.0 )
65:     return vec3 (0.0);
66:   h = normalize ( v + l );
67:   cnh = dot ( n, h );
68:   c2nh = cnh*cnh;
69:   t2nh = (1.0-c2nh)/c2nh;
70:   D = exp ( -t2nh/m2 ) / (PI*m2*c2nh*c2nh);
71:   cvh = dot ( v, h );
72:   g = (cnh+cnh)/cvh;
73:   if ( (G = g*cnv) > 1.0 )
74:     if ( (G = g*cnl) > 1.0 )
75:       G = 1.0;
76:   F = Fl0 + (vec3(1.0)-Fl0)*pow( 1.0-cvh, 5.0 );
77:   return D*G*F/(4.0*cnv*cnl);
78: } /*CookTorranceRho*/
79:
80: vec3 PBRLighting ( void )
81: {
82:   vec3  lv, vv, rhoD, rhoS, Colour, lp;
83:   float dist, cnv, cnl, m2, s;
84:   uint  i, mask;
85:
86:   vv = posDifference ( trb.eyepos, In.Position, dist );
87:   if ( dot ( vv, tnormal ) < 0.0 ) normal = -normal;
88:   cnv = dot ( normal, vv );
89:   Colour = vec3 ( 0.0 );
90:   m2 = mm.m*mm.m;
91:   for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
92:     if ( (light.mask & mask) != 0 ) {
93:       s = ((light.shmask & mask) != 0) ? IsEnlighted ( i ) : 1.0;
94:       if ( s > 0.0 ) {
95:         lv = posDifference ( light.ls[i].position, In.Position, dist );
96:         if ( (cnl = dot ( normal, lv )) > 0.0 ) {
97:           lp = s * light.ls[i].direct * cnl;
98:           if ( light.ls[i].position.w != 0.0 )
99:             lp *= attFactor ( light.ls[i].attenuation, dist );
100:          rhoD = mm.kD == 0.0 ? vec3(0.0) :
101:                 OrenNayarRho ( normal, vv, lv, cnv, cnl, 0.5*m2, mm.cl );
102:          rhoS = mm.kS == 0.0 ? vec3(0.0) :
103:                 CookTorranceRho ( normal, vv, lv, cnv, cnl, m2, mm.Fl0 );
104:          Colour += (mm.kD*rhoD + mm.kS*rhoS) * lp;
105:        }
106:      }
107:    }
108:  return clamp ( Colour, 0.0, 1.0 );
109: } /*PBRLighting*/
110:
```

```
111: void main ( void )
112: {
113:   normal = normalize ( In.Normal );
114:   tnormal = In.TNormal;
115:   GetMaterial ( In.TexCoord );
116:   out_Colour = vec4 ( AGamma ( PBRLighting () ), 1.0 );
117: } /*main*/
```

Procedura `main` w liniach 113–114 zapamiętuje w zmiennych globalnych jednostkowe wektory normalne powierzchni i przybliżającego ją trójkąta, a podprogram wywołany w linii 115 nadaje wartość zmiennej `mm` opisującej własności materiału. Mogą one zależeć od współrzędnych tekstury fragmentu.

Oświetlenie oblicza procedura `PBRLighting`, która w linii 86 oblicza wektor $\boldsymbol{v}$. Jeśli kąt między nim a wektorem normalnym trójkąta jest rozwarty, to w linii 87 następuje zmiana zwrotu wektora $\boldsymbol{n}$. Pętla w liniach 91–107 przebiega po źródłach światła. Jeśli i-te źródło jest włączone i fragment nie jest w cieniu, to w linii 95 szader oblicza wektor $\boldsymbol{l} = \boldsymbol{l}_i$, a następnie bada, czy źródło oświetla widoczną stronę powierzchni. W liniach 97–99 następuje obliczenie irradiancji. Wywołane w liniach 101 i 103 podprogramy obliczają wartości funkcji ρ_d i ρ_s, po czym (w linii 104) wartość zmiennej `Colour` jest zwiększana o składnik opisujący radiancję odbitego w kierunku $\boldsymbol{v}$ światła, które dochodzi od i-tego źródła.

Oba podprogramy obliczające składniki funkcji odbicia mają takie same pierwsze pięć parametrów: wektory $\boldsymbol{n}$, $\boldsymbol{v}$ i $\boldsymbol{l}$ oraz obliczone przez procedurę `PBRLighting` kosinusy kątów $\vartheta_{\boldsymbol{v}}$ i $\vartheta_{\boldsymbol{l}}$. Przedostatni parametr opisuje chropowatość powierzchni; jest on równy σ^2 albo m^2, przy czym $\sigma = m/\sqrt{2}$; liczba m jest podana w opisie materiału. Ostatni parametr jest to wektor $\boldsymbol{c}_\lambda$ albo $\boldsymbol{F}_{\lambda,0}$.

Jeśli $\sigma = 0$, to podprogram `OrenNayarRho`, nie tracąc czasu, podaje jako wynik wektor $\boldsymbol{c}_\lambda/\pi$, tak jak w modelu Lamberta. W przeciwnym razie porównuje w linii 35 kosinusy kątów $\vartheta_{\boldsymbol{v}}$ i $\vartheta_{\boldsymbol{l}}$, aby zbadać, który z nich jest kątem α, a który kątem β. Kąty te są obliczane w linii 37.

Jeśli $\cos\beta < 1$, czyli $0 < \beta \leqslant \alpha$, to w liniach 39–52 są obliczane współczynniki d, C_2 i C_3, które w przeciwnym razie nie są potrzebne do obliczenia funkcji $\rho_{\mathrm{d},1}$. W linii 46 podprogram na podstawie kosinusa kąta β oblicza jego tangens. Pozostałe instrukcje nie wymagają objaśnień. Podprogram nie oblicza składnika $\rho_{\mathrm{d},2}$, ale ufam, że dodanie instrukcji, które to robią, trudności Czytelnikom nie sprawi.

Podprogram `CookTorranceRho` nie nadaje się do wykonywania obrazów idealnie gładkich powierzchni lustrzanych, bo w nich ma być widoczne wyraźne odbicie otaczających obiektów, co da się osiągnąć tylko za pomocą śledzenia promieni lub w sposób opisany w rozdziale 20. Dlatego jeśli parametr m jest zerem, to podprogram przekazuje wynik $\mathbf{0}$. Jeśli $m \neq 0$, to zmienne `cnh`, `c2nh`, `t2nh` i `cvh` otrzymują kolejno wartości $\langle \boldsymbol{h}, \boldsymbol{n} \rangle = \cos\vartheta_{\boldsymbol{h}}$, $\cos^2\vartheta_{\boldsymbol{h}}$, $\mathrm{tg}^2\,\vartheta_{\boldsymbol{h}}$ i $\langle \boldsymbol{v}, \boldsymbol{h} \rangle = \langle \boldsymbol{l}, \boldsymbol{h} \rangle$. W linii 70 następuje obliczenie czynnika D według rozkładu Beckmanna-Spizzichino (28.9). Czynnik zasłaniania mikrościanek G jest obliczany w liniach 71–75. Instrukcja w linii 76 oblicza przybliżenie Schlicka czynnika Fresnela. Końcowe obliczenie wartości funkcji ρ_s następuje w linii 77.

Obrazki na górze rysunku 28.10 umożliwiają porównanie (na przykładzie czajnika z terakoty, $c_\lambda = (0.8, 0.55, 0.3)$) modelu oświetlenia Lamberta z modelem Orena i Nayara (z parametrem $\sigma = 0.5$). Niżej są narysowane (za pomocą modelu Cooka i Torrance'a) czajniki metalowe: stalowy ($\boldsymbol{F}_{\lambda,0} = (0.56, 0.57, 0.58)$, $m = 0.12$) i miedziany ($\boldsymbol{F}_{\lambda,0} = (0.85, 0.28, 0.15)$, $m = 0.25$), przy czym wszystkie czajniki są oświetlone tylko przez jedno źródło.

Rysunek 28.10. Skutki użycia różnych modeli oświetlenia

Jak dobierać parametry modelu? Szczegółowe informacje na temat współczynników załamania światła wielu różnych substancji, umożliwiające obliczanie czynników Fresnela znajdują się na stronie [62]. Dla przedmiotów metalowych powinno być $k_\mathrm{d} = 0$, ponieważ od przewodników światło odbija się tylko w sposób zwierciadlany; fotony, które przeszły przez mikrościanki zostaną pochłonięte, zanim zdążą przeniknąć z powrotem przez granicę ośrodków. Ale powierzchnie metalowe mogą być zanieczyszczone, na przykład zakurzone, zatłuszczone, zabłocone lub nawet zardzewiałe. Część światła odbija się od tych zanieczyszczeń w sposób rozproszony, co uzasadnia przyjmowanie $k_\mathrm{d} > 0$. Zanieczyszczenia i ślady korozji zazwyczaj są rozmieszczone nierównomiernie, dlatego parametry modelu nie powinny być stałe na całej powierzchni. Najlepiej jest brać je z tekstur. Dotyczy to czynników k_d i $k_\mathrm{s} = 1 - k_\mathrm{d}$, ale też parametru m funkcji rozkładu mikrościanek, wektorów $\boldsymbol{c}_\lambda$ i $\boldsymbol{F}_{\lambda,0}$ oraz wektora normalnego, którego zaburzenia reprezentują makroskopowe nierówności powierzchni. To oczywiście wymaga dokonania w treści szadera z listingu 28.9 odpowiednich przeróbek (w tym napisania odpowiedniej procedury `GetMaterial`), co pozostawiam Czytelnikom jako średnio zaawansowane ćwiczenie.

28.4.3. Implementacja oświetlenia przez otoczenie

Szadery realizujące modele oświetlenia przedstawione w tym podrozdziale lub nawet jeszcze dokładniejsze mogą trochę poprawić jakość obrazów, ale uwzględnienie *tylko* światła dochodzącego bezpośrednio od źródeł punktowych nie zapewni pełnego realizmu. Aby przedmiot wyglądał naturalnie, trzeba radiancję światła odbitego od jego powierzchni scałkować po wszystkich kierunkach, z których światło na nią pada. W książce [25] jest opisane rozwiązanie tego zadania, tj. procedury i szadery umożliwiające szybkie obliczanie potrzebnych całek. Jego opis (a zwłaszcza opis podstaw teoretycznych) pozostawia wiele do życzenia, ale najważniejsze pomysły użyte w rozwiązaniu przedstawionym niżej wziąłem stamtąd (i nie mogąc się powstrzymać, trochę pozmieniałem). Najpierw oczywiście przedstawię teorię.

Po wstawieniu do wzoru (28.2) wyrażeń opisujących składnik odblaskowy dwukierunkowej funkcji odbicia światła (28.7) i irradiancję światła dochodzącego z kierunku $\boldsymbol{l}$ całkowitą radiancję L_{rs} światła odbitego od punktu $\boldsymbol{p}$ powierzchni nieprzezroczystej w kierunku $\boldsymbol{v}$ w sposób odblaskowy wyrazimy wzorem

$$L_{\mathrm{rs}}(\boldsymbol{p},\boldsymbol{v}) = \int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} \frac{DGF_\lambda}{4\langle \boldsymbol{l},\boldsymbol{n}\rangle\langle \boldsymbol{v},\boldsymbol{n}\rangle} L(\boldsymbol{p}+\boldsymbol{l},-\boldsymbol{l})\langle \boldsymbol{l},\boldsymbol{n}\rangle \,\mathrm{d}S. \tag{28.18}$$

Pełne obliczanie występującej w nim całki jest zbyt kosztowne, aby GPU była w stanie mu podołać w trakcie animacji w czasie rzeczywistym, a korzystanie z preprocesingu jest niewykonalne z powodu zbyt dużej ilości danych, które trzeba byłoby zapamiętać. Konieczny jest więc kompromis: całkę zastąpimy jej (zgrubnym, ale wystarczająco dokładnym) przybliżeniem. Najpierw w wyrażeniu podcałkowym wyodrębnimy czynniki D i $L(\boldsymbol{p}+\boldsymbol{l},-\boldsymbol{l})$ i zastąpimy je wartością średnią ich iloczynu. W ten sposób dostaniemy wzór

$$L_{\mathrm{rs}}(\boldsymbol{p},\boldsymbol{v}) \approx \left(\frac{1}{2\pi}\int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} DL(\boldsymbol{p}+\boldsymbol{l},-\boldsymbol{l})\,\mathrm{d}S\right)\left(\int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} \frac{GF_\lambda}{4\langle \boldsymbol{v},\boldsymbol{n}\rangle}\,\mathrm{d}S\right). \tag{28.19}$$

Czynnik $\frac{1}{2\pi}$ jest odwrotnością miary zbioru całkowania (tj. półsfery $S_{\boldsymbol{n}+}$). Pomnożona przez niego pierwsza całka, którą oznaczę symbolem L_{i}, opisuje przefiltrowaną radiancję światła dochodzącego z otoczenia[27]. Drugą całką, nazwę ją M_λ, zajmiemy się dalej. Obliczanie tak przekształconej radiancji podzielimy na etapy preprocesingu i końcowy.

Aby zredukować ilość danych otrzymanych w preprocesingu pierwszej całki, uczynimy dwa założenia. Po pierwsze, czynnik D, który zależy od wektora $\boldsymbol{n}$ i wektora kierunku padania światła $\boldsymbol{l}$ (a także od chropowatości powierzchni), ma być jednoznacznie określony przez kąt $\vartheta_{\boldsymbol{h}}$ między wektorami $\boldsymbol{n}$ a $\boldsymbol{h}$, co oznacza przyjęcie, że powierzchnia jest izotropowa. Po drugie, wykonamy obliczenia tylko dla $\boldsymbol{v} = \boldsymbol{n}$, uznając, że to wystarczy. Przy tym ograniczeniu kąt $\vartheta_{\boldsymbol{h}}$ jest połową kąta $\vartheta_{\boldsymbol{l}}$ między wektorami $\boldsymbol{n}$ a $\boldsymbol{l}$, a całka jest funkcją wektora $\boldsymbol{n}$: $L_{\mathrm{i}} = L_{\mathrm{i}}(\boldsymbol{n})$. Czynnik D możemy przyjąć na podstawie wzoru (28.9) lub (28.11); w każdym

[27] Całka taka jak we wzorze (28.18) określa **splot sferyczny** funkcji L z funkcją, która jest iloczynem pozostałych czynników funkcji podcałkowej (porównaj z opisem splotu funkcji określonych na płaszczyźnie w podrozdz. 27.2). Zamiast niego użyjemy splotu funkcji L i D, opisanego przez pierwszą całkę we wzorze (28.19).

z nich występuje tylko jeden parametr chropowatości, m. Mając ustaloną (kostkową) teksturę radiancji światła dochodzącego ze wszystkich kierunków, możemy obliczyć całkę $L_\mathrm{i}(\boldsymbol{n})$ dla różnych wektorów jednostkowych $\boldsymbol{n}$ i przechować ją w nowej teksturze kostkowej, która będzie pełnić rolę analogiczną do opisanej w p. 28.3.2 tekstury irradiancji. Można to zrobić dla kilku wartości parametru m. Im większą ma on wartość, tym bardziej „rozmyta" jest przefiltrowana radiancja i tym mniejsza jest potrzebna rozdzielczość tej tekstury.

Listing 28.10 przedstawia szader fragmentów, którego zadaniem jest obliczenie w preprocesingu przefiltrowanej radiancji L_i. Szader ten współpracuje z szaderami wierzchołków i geometrii z listingów 28.1 i 28.2. Użyta w nim kwadratura korzysta z parametrycznego przedstawienia półsfery $S_{\boldsymbol{n}+}$ w układzie współrzędnych, w którym $\boldsymbol{n} = \boldsymbol{e}_3 = (0,0,1)$ (rys. 28.5):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}(\lambda,\vartheta) = \begin{bmatrix} \cos\lambda\sin\vartheta \\ \sin\lambda\sin\vartheta \\ \cos\vartheta \end{bmatrix}, \quad \lambda\in[0,2\pi),\ \vartheta\in[0,\pi/2].$$

Całkę z funkcji f na półsferze $S_{\boldsymbol{n}+}$ można obliczyć numerycznie na podstawie wzoru

$$\begin{aligned}\int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} f(\boldsymbol{l})\,\mathrm{d}S &= \int_0^{\pi/2}\left(\int_0^{2\pi} f\big(\boldsymbol{l}(\lambda,\vartheta)\big)\,\mathrm{d}\lambda\right)\sin\vartheta\,\mathrm{d}\vartheta \\ &\approx \frac{\pi^2}{NM}\sum_{j=1}^{M}\sum_{i=1}^{N} f\big(\boldsymbol{l}(\lambda_i,\vartheta_j)\big)\sin\vartheta_j, \quad \text{gdzie } \lambda_i = \pi\frac{2i-1}{N},\ \vartheta_j = \pi\frac{2j-1}{4M}.\end{aligned}$$

Aby go otrzymać, prostokąt $[0,2\pi)\times[0,\pi/2]$ podzieliłem na NM prostokątów i przyjąłem środek każdego z nich za węzeł kwadratury, której wszystkie współczynniki mają tę samą wartość $\pi^2/(NM)$. Aby uwzględnić czynnik $\frac{1}{2\pi}$ we wzorze (28.19), współczynniki te zastąpiłem liczbą $\pi/(2NM)$, przez którą w linii 43 jest mnożona suma wartości funkcji podcałkowej w węzłach kwadratury[28].

Mając liczby λ_i, ϑ_j, w linii 36 znajdujemy wektor $\tilde{\boldsymbol{l}} = (\cos\lambda_i\sin\vartheta_j, \sin\lambda_i\sin\vartheta_j, \cos\vartheta_j)$, a następnie poddajemy go przekształceniu przeprowadzającemu wektor $\boldsymbol{e}_3$ na wektor $\boldsymbol{n}$ odpowiadający środkowi teksela, w którym ma być zapamiętany wynik (i wektor $\tilde{\boldsymbol{l}}$ na $\boldsymbol{l}$); to przekształcenie jest identyczne, jak stosowane przez szader z listingu 28.3.

Na podstawie sinusa i kosinusa kąta $\vartheta_{\boldsymbol{l}} = \vartheta$ między wektorami $\boldsymbol{l}$ a $\boldsymbol{n}$ w liniach 30–31 są obliczane funkcje kąta $\vartheta_{\boldsymbol{h}}$ (który, przypomnijmy, jest tu połową kąta $\vartheta_{\boldsymbol{l}}$):

$$\cos^2\vartheta_{\boldsymbol{h}} = \frac{1+\cos\vartheta_{\boldsymbol{l}}}{2}, \quad \sin^2\vartheta_{\boldsymbol{h}} = \frac{\sin^2\vartheta_{\boldsymbol{l}}}{4\cos^2\vartheta_{\boldsymbol{h}}}, \quad \mathrm{tg}^2\,\vartheta_{\boldsymbol{h}} = \frac{\sin^2\vartheta_{\boldsymbol{h}}}{\cos^2\vartheta_{\boldsymbol{h}}}.$$

W linii następnej jest obliczany czynnik D według rozkładu Beckmanna-Spizzichino (28.9). Otrzymana wartość całki $L_\mathrm{i}(\boldsymbol{n})$ jest przekazywana na wyjście i zostaje zapamiętana w odpowiednim tekselu wynikowej tekstury kostkowej.

[28] To jest bardzo prosta kwadratura, ale w tym zastosowaniu sprawdziła się nadspodziewanie dobrze.

Listing 28.10. Szader fragmentów obliczający teksturę przefiltrowanej radiancji otoczenia

GLSL

```
#version 450 core

#define PI   3.141592653
#define N  200
#define M   50

in vec3 Normal;

layout(location=0) out vec4 out_Colour;

layout(binding=0) uniform samplerCube RadianceTxt;

uniform float roughness2;

void main ( void )
{
  int   i, j, k;
  float gamma, lambda, slambda, clambda, theta, stheta, ctheta,
        c2nh, s2nh, t2nh, D_BS;
  vec3  l, w, intrad, intradp;

  w = normalize ( Normal );
  w.z += Normal.z > 0.0 ? 1.0 : -1.0;
  gamma = 2.0 / dot ( w, w );
  intrad = vec3 ( 0.0 );
  for ( j = 1; j < 2*M; j += 2 ) {
    intradp = vec3 ( 0.0 );
    theta = float(j)/float(4*M)*PI;
    stheta = sin ( theta );  ctheta = cos ( theta );
    c2nh = 0.5*(1.0+ctheta);  s2nh = stheta*stheta/(4.0*c2nh);
    t2nh = s2nh/c2nh;
    D_BS = exp ( -t2nh/roughness2 ) / (PI*roughness2*c2nh*c2nh);
    for ( i = 1; i < 2*N; i += 2 ) {
      lambda = float(i)/float(N)*PI;
      slambda = sin ( lambda );  clambda = cos ( lambda );
      l = vec3 ( clambda*stheta, slambda*stheta, ctheta );
      l -= w*(gamma*dot ( w, l ));
      if ( Normal.z > 0.0 ) l = -l;
      intradp += texture ( RadianceTxt, l ).xyz * D_BS;
    }
    intrad += intradp*stheta;
  }
  out_Colour = vec4 ( intrad*(PI/(float(2*N*M))), 1.0 );
} /*main*/
```

Listing 28.11. Procedura całkowania radiancji otoczenia dla odbicia zwierciadlanego

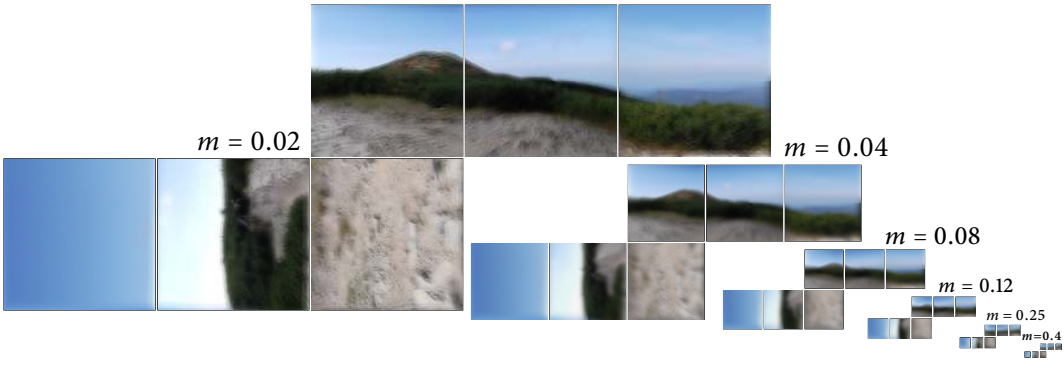
C

```
1: #define IBLSPECTXTSIZE 128
2: #define IBLSPECTXTMIN    4
3:
4: static const GLfloat roughness[6] = {0.02, 0.04, 0.08, 0.12, 0.25, 0.4};
5:
6: GLuint ConstructIBLSpecTexture ( GLuint prog_id, GLuint r2_loc,
7:                                  GLuint radtxt )
8: {
9:   GLuint fbo, status, itxt;
10:  int    i, wh;
11:
12:  glActiveTexture ( GL_TEXTURE0 );
13:  itxt = CreateCubeTexture ( IBLSPECTXTSIZE, GL_RGB32F, 5 );
14:  glTexParameteri ( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
15:  glTexParameteri ( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
16:  glGenFramebuffers ( 1, &fbo );
17:  glBindFramebuffer ( GL_FRAMEBUFFER, fbo );
18:  glDrawBuffer ( GL_COLOR_ATTACHMENT0 );
19:  glBindTexture ( GL_TEXTURE_CUBE_MAP, radtxt );
20:  glBindVertexArray ( empty_vao );
21:  glUseProgram ( prog_id );
22:  for ( i = 0, wh = IBLSPECTXTSIZE;  wh >= IBLSPECTXTMIN;  i++, wh /= 2 ) {
23:    glFramebufferTexture ( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, itxt, i );
24:    if ( (status = glCheckFramebufferStatus ( GL_FRAMEBUFFER)) !=
25:          GL_FRAMEBUFFER_COMPLETE )
26:      ExitOnError ( "ConstructIBLSpecTexture" );
27:    glViewport ( 0, 0, wh, wh );
28:    glUniform1f ( r2_loc, roughness[i]*roughness[i] );
29:    glDrawArrays ( GL_POINTS, 0, 1 );
30:    glFlush ();
31:  }
32:  glUseProgram ( 0 );
33:  glBindVertexArray ( 0 );
34:  glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
35:  glDeleteFramebuffers ( 1, &fbo );
36:  return itxt;
37: } /*ConstructIBLSpecTexture*/
```

Listing 28.11 przedstawia procedurę, której zadaniem jest utworzenie odpowiedniej tekstury i obliczenie całek przy użyciu programu zawierającego szader opisany wyżej. Jest to tekstura kostkowa wielopoziomowa; na każdym jej poziomie zostaną zapamiętane całki dla innego parametru m.

Kostkowa tekstura radiancji otoczenia, której identyfikator jest podany jako parametr, jest w liniach 12–13 przywiązywana do celu, dzięki czemu szader ma do niej dostęp poprzez zmienną `RadianceTxt`.

Po utworzeniu tekstury (przez procedurę z listingu 26.11) i pozaekranowego bufora ramki, w kolejnych przebiegach pętli, jako załącznik obrazu są (w linii 23) doczepiane kolejne poziomy tej tekstury, a następnie ustalane odpowiednie wymiary klatki, przy czym każdy kolejny poziom ma rozdzielczość dwukrotnie mniejszą od poprzedniego[29]. W linii 28 zmienna jednolita `roughness2` otrzymuje wartość m^2, po czym następuje rysowanie na danym poziomie, czyli obliczanie całek dla odpowiedniego parametru m. Na końcu procedura sprząta, tj. odczepia program szaderów i obiekt tablicy wierzchołków i likwiduje bufor ramki.



Rysunek 28.11. Tekstura przefiltrowanej radiancji otoczenia

Rysunek 28.11 przedstawia otrzymaną przy użyciu opisanej tu procedury teksturę przefiltrowanej radiancji na podstawie tekstury pokazanej na rysunku 28.4a. Ściany tekstury mają rozdzielczość 128 × 128 na poziomie 0 i 4 × 4 poziomie 5.

Teraz zajmiemy się obliczaniem drugiej całki we wzorze (28.19). Ją również uprościmy, zastępując czynnik Fresnela F_λ przybliżeniem Schlicka (28.14). Dostaniemy wtedy

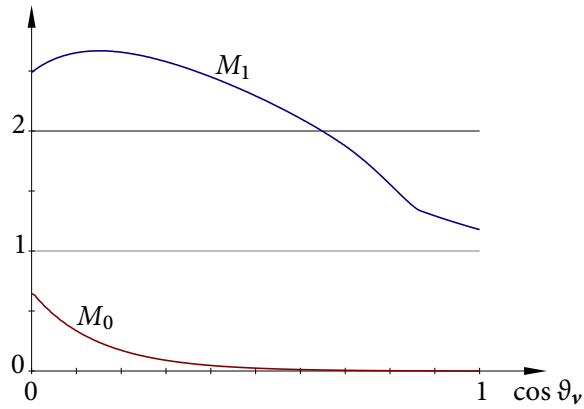
$$M_\lambda \approx \int_{\boldsymbol{l}\in S_{n+}} \frac{G\big(F_{\lambda,0} + (1-F_{\lambda,0})(1-\langle \boldsymbol{l},\boldsymbol{h}\rangle)^5\big)}{4\langle \boldsymbol{v},\boldsymbol{n}\rangle}\,\mathrm{d}S = M_0(\vartheta_{\boldsymbol{v}}) + F_{\lambda,0}M_1(\vartheta_{\boldsymbol{v}}), \tag{28.20}$$

$$\text{gdzie}\quad M_0(\vartheta_{\boldsymbol{v}}) = \int_{\boldsymbol{l}\in S_{n+}} \frac{G(1-\langle \boldsymbol{l},\boldsymbol{h}\rangle)^5}{4\langle \boldsymbol{v},\boldsymbol{n}\rangle}\,\mathrm{d}S, \quad M_1(\vartheta_{\boldsymbol{v}}) = \int_{\boldsymbol{l}\in S_{n+}} \frac{G\big(1-(1-\langle \boldsymbol{l},\boldsymbol{h}\rangle)^5\big)}{4\langle \boldsymbol{v},\boldsymbol{n}\rangle}\,\mathrm{d}S.$$

Umożliwia to obliczenie w preprocesingu całek M_0 i M_1, które są funkcjami tylko jednego parametru, czyli kąta $\vartheta_{\boldsymbol{v}}$ między wektorami $\boldsymbol{v}$ a $\boldsymbol{n}$, i zastosowanie ich do obliczania oświetlenia *wszystkich* przedmiotów wykonanych z materiałów, których dwukierunkowe funkcje odbicia światła są opisane wzorem (28.7). Obliczone wartości funkcji M_0 i M_1, których wykresy są pokazane na rysunku 28.12, najwygodniej jest zapamiętać w teksturze jednowymiarowej.

Listingi 28.12 i 28.13 przedstawiają szader obliczeniowy i korzystającą z niego procedurę, których zadaniem jest stablicowanie i zapamiętanie w teksturze całek M_0 i M_1 dla różnych kątów $\vartheta_{\boldsymbol{v}}$. Do obliczenia tych całek jest użyta taka sama kwadratura, jak ta do obliczenia

[29] Rozdzielczość tekstury na poziomie 0 jest całkowitą potęgą liczby 2, co nie jest konieczne, ale pozwala uprościć procedurę.



Rysunek 28.12. Wykresy całek M_0 i M_1 w funkcji kosinusa kąta ϑ_v

Listing 28.12. Szader obliczający funkcje M_0 i M_1

GLSL

```
#version 450

#define PI   3.141592653
#define N  200
#define M   50

layout(local_size_x=1) in;

layout(rg32f,binding=0) uniform image1D m0m1;

vec3 v;

vec2 Integrand ( vec3 l )
{
  vec3  h;
  float hdotl, vdoth, hdotn, sf, g, G;

  h = normalize ( l+v );
  hdotl = dot ( h, l );
  vdoth = dot ( v, h );
  g = (h.z+h.z)/vdoth;
  if ( v.z == 0.0 ) G = g;    /* cos theta v == 0.0 */
  else if ( (G = g*v.z) > 1.0 )
         if ( (G = g*l.z) > 1.0 )
           G = 1.0;
  sf = pow ( 1.0-hdotl, 5.0 );
  return G*vec2 ( sf, 1.0-sf );
} /*Integrand*/

```

```
30: void main ( void )
31: {
32:   int   i, j, k;
33:   float lambda, slambda, clambda, theta, stheta, ctheta, cthetav;
34:   vec2  intf, intfp;
35:   vec3  l;
36:
37:   k = int(gl_GlobalInvocationID.x);
38:   cthetav = float ( k ) / float ( gl_NumWorkGroups.x-1 );
39:   v = vec3 ( sqrt ( 1.0 - cthetav*cthetav ), 0.0, cthetav );
40:   intf = vec2 ( 0.0 );
41:   for ( j = 1; j < 2*M; j += 2 ) {
42:     intfp = vec2 ( 0.0 );
43:     theta = float(j)/float(4*M)*PI;
44:     stheta = sin ( theta );  ctheta = cos ( theta );
45:     for ( i = 1; i < N; i += 2 ) {
46:       lambda = float(i)/float(N)*PI;
47:       slambda = sin ( lambda );  clambda = cos ( lambda );
48:       l = vec3 ( clambda*stheta, slambda*stheta, ctheta );
49:       intfp += Integrand ( l );
50:     }
51:     intf += intfp*stheta;
52:   }
53:   if ( k > 0 )   /* cos theta v > 0.0 */
54:     intf /= cthetav;
55:   imageStore ( m0m1, k, vec4 ( PI*PI/float(2*N*M)*intf, 0.0, 0.0 ) );
56: } /*main*/
```

całki L_i. Półsferę $S_{\boldsymbol{n}+}$ przedstawiamy w układzie współrzędnych, w którym $\boldsymbol{n} = \boldsymbol{e}_3 = (0,0,1)$, a wektor $\boldsymbol{v}$ ma drugą współrzędną równą 0: $\boldsymbol{v} = (\sin\vartheta_{\boldsymbol{v}}, 0, \cos\vartheta_{\boldsymbol{v}})$.

Obie funkcje podcałkowe są ułamkami z mianownikiem $4\langle\boldsymbol{v},\boldsymbol{n}\rangle$. Jest to wyrażenie stałe, które możemy wynieść przed całkę, z wyjątkiem przypadku granicznego, gdy $\langle\boldsymbol{v},\boldsymbol{n}\rangle = 0$. Gdy kąt $\vartheta_{\boldsymbol{v}}$ jest bliski $\pi/2$, a kąt $\vartheta_{\boldsymbol{l}}$ jest mniejszy, czynnik G jest równy $2\langle\boldsymbol{h},\boldsymbol{n}\rangle\langle\boldsymbol{v},\boldsymbol{n}\rangle/\langle\boldsymbol{v},\boldsymbol{h}\rangle$, więc w tym przypadku można usunąć czynnik $\langle\boldsymbol{v},\boldsymbol{n}\rangle$ z licznika i mianownika i uniknąć dzielenia $0/0$. Ponadto jedna i druga funkcja podcałkowa dla każdego λ i ϑ ma takie same wartości w punktach (λ,ϑ) i $(2\pi-\lambda,\vartheta)$. Dzięki tej symetrii obliczenia można dwukrotnie przyspieszyć, ograniczając przedział całkowania parametru λ do $[0,\pi)$ i mnożąc współczynniki kwadratury przez 2. Uwzględniając czynnik 4 w mianowniku, przyjmiemy warunek kontynuacji wewnętrznej pętli w linii 45 o postaci `i < N` zamiast `i < 2*N`, a w linii 55 sumę wartości funkcji podcałkowej w węzłach kwadratury mnożymy przez $\pi^2/(2MN)$.

Parametr procedury `IntegrateSpecFLambda` (listing 28.13) jest identyfikatorem programu składającego się z opisanego tu szadera, a liczba przekazywana w instrukcji `return` jest identyfikatorem tekstury jednowymiarowej przechowującej wartości całek M_0 i M_1. Format `GL_RG32F` zapewnia, że teksele mają dwie składowe, przechowywane w zmiennych typu `float`. W liniach 7–11 procedura tworzy teksturę, której parametry umożliwiają właściwą

interpolację tekseli podczas rysowania, po czym udostępnia szaderowi (za pośrednictwem zmiennej `m0m1`) jej tablicę tekseli, wywołując w linii 12 procedurę `glBindImageTexture`. Pierwszy parametr procedury `glDispatchCompute` zapewnia, że każdy wątek szadera otrzymuje jeden teksel, w którego składowych `xy` ma zapamiętać wartości całek M_0 i M_1 obliczone dla jednego kąta $\vartheta_{\boldsymbol{v}}$ (jego kosinus szader oblicza w linii 38).

Listing 28.13. Procedura obliczania funkcji M_0 i M_1

C

```
1: #define FLTXTSIZE 256
2:
3: GLuint IntegrateSpecFLambda ( GLuint prog_id )
4: {
5:   GLuint fltxt;
6:
7:   glGenTextures ( 1, &fltxt );
8:   glBindTexture ( GL_TEXTURE_1D, fltxt );
9:   glTexStorage1D ( GL_TEXTURE_1D, 1, GL_RG32F, FLTXTSIZE );
10:  glTexParameteri ( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
11:  glTexParameteri ( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
12:  glBindImageTexture ( 0, fltxt, 0, GL_FALSE, 0, GL_WRITE_ONLY, GL_RG32F );
13:  glUseProgram ( prog_id );
14:  glDispatchCompute ( FLTXTSIZE, 1, 1 );
15:  glUseProgram ( 0 );
16:  glBindTexture ( GL_TEXTURE_1D, 0 );
17:  ExitIfGLError ( "IntegrateSpecFLambda" );
18:  return fltxt;
19: } /*IntegrateSpecFLambda*/
```

Mając przygotowane w preprocesingu tekstury przefiltrowanej radiancji otoczenia i stablicowanych całek M_0 i M_1, możemy rysować, ale trzeba przy tym użyć tych tekstur we właściwy sposób. Mamy oświetlić fragment powierzchni o wektorze normalnym $\boldsymbol{n}$ obserwowany z kierunku $\boldsymbol{v}$. Gdyby powierzchnia była idealnym lustrem, w stronę obserwatora odbiłoby się światło dochodzące z kierunku wektora $\boldsymbol{r} = 2\langle \boldsymbol{v}, \boldsymbol{n}\rangle \boldsymbol{n} - \boldsymbol{v}$. Obraz otoczenia odbity w lustrze nieidealnym (czyli w naszej powierzchni) jest rozmyty i tekstura przefiltrowanej radiancji przechowuje właśnie takie obrazy. Choć pierwszą całkę we wzorze (28.19) obliczyliśmy jako funkcję wektora $\boldsymbol{n}$ (przyjmując $\boldsymbol{v} = \boldsymbol{n}$, czyli także $\boldsymbol{r} = \boldsymbol{n}$), obliczając radiancję, jako argument tej funkcji podamy wektor $\boldsymbol{r}$.

Listing 28.14 przedstawia zmiany, które trzeba wprowadzić do szadera z listingu 28.9, aby dodać oświetlenie obiektu przez otoczenie. Jeśli dwukierunkowa funkcja odbicia światła zawiera składnik realizujący odbicie rozproszone zgodnie z modelem Orena i Nayara, to możemy go w tym miejscu zastąpić modelem Lamberta i użyć jednego ze sposobów przedstawionych w p. 28.3.2 — pokazany szader korzysta z wielomianowego przybliżenia irradiancji, reprezentowanego przez liczby przechowywane w bloku `Irrad`.

Listing 28.14. Szader fragmentów — oświetlenie przez otoczenie

GLSL

```
1: layout(binding=4,std430) buffer Irrad { float a[27]; } irrad;
2:
3: layout(binding=10) uniform samplerCube SpecRadianceTxt;
4: layout(binding=11) uniform sampler1D m0m1Txt;
5:
6: vec3 SpecularEnvLighting ( vec3 normal, vec3 vv, float cthetav )
7: {
8:   vec3 Li, r, Fl;
9:   vec2 M0M1;
10:
11:  r = -reflect ( vv, normal );
12:  Li = texture ( SpecRadianceTxt, r ).rgb;
13:  M0M1 = texture ( m0m1Txt, cthetav ).xy;
14:  Fl = M0M1.xxx + M0M1.y*mm.Fl0;
15:  return Li*Fl;
16: } /*SpecularEnvLighting*/
17:
18: vec3 PBRLighting ( void )
19: {
20:  vec3  lv, vv, Colour, rhoD, rhoS, lp;
21:  float dist, m2, cnv, cnl, s;
22:  uint  i, mask;
23:
24:  vv = posDifference ( trb.eyepos, In.Position, dist );
25:  if ( dot ( vv, tnormal ) < 0.0 ) normal = -normal;
26:  cnv = dot ( vv, normal );
27:  Colour = mm.kD > 0.0 ? mm.kD*mm.cl*LightPoly ( normal ) : vec3 ( 0.0 );
28:  if ( mm.kS > 0.0 )
29:    Colour = mm.kS*SpecularEnvLighting ( normal, vv, cnv );
30:  m2 = mm.m*mm.m;
31:  for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
32:    .... /* obliczanie oświetlenia przez źródła punktowe */
33:    .... /* tak jak na listingu 28.9 w liniach 94-109 */
34:  return clamp ( Colour, 0.0, 1.0 );
35: } /*PBRLighting*/
```

Tekstury przefiltrowanej radiancji i funkcji M_0 i M_1 są dostępne za pośrednictwem zmiennych `SpecRadianceTxt` i `m0m1Txt`. Zamiast nadawać zmiennej `Colour` wartość początkową **0**, procedura `PBRLighting` wywołuje procedurę `SpecularEnvLighting`, która w linii 11 oblicza wektor $\boldsymbol{r}$, w linii 12 oblicza wektor $\frac{1}{2\pi}L_\mathrm{i}(\boldsymbol{r})$, w linii 13 odczytuje z tekstury wartości funkcji $\frac{1}{4}M_0$ i $\frac{1}{4}M_1$ dla kąta $\vartheta_{\boldsymbol{v}}$, którego kosinus jest trzecim parametrem, w linii 14 oblicza przybliżenie całki M_λ i wreszcie w linii 15 oblicza przybliżoną wartość radiancji odbijanego w kierunku $\boldsymbol{v}$ światła dochodzącego z otoczenia.

Przed przystąpieniem do rysowania trzeba wykonać następujące instrukcje:

```
glActiveTexture ( GL_TEXTURE0 + 11 );
glBindTexture ( GL_TEXTURE_1D, fltxt );
glActiveTexture ( GL_TEXTURE0 + 10 );
glBindTexture ( GL_TEXTURE_CUBE_MAP, itxt );
glTexParameteri( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_BASE_LEVEL, .... );
glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 4, irrpoly );
ChooseMaterial ( .... );
```

Udostępniają one szaderowi dane potrzebne do oświetlenia obiektów przez otoczenie. Pierwsze dwie instrukcje przywiązują teksturę z wartościami funkcji M_0 i M_1 do zmiennej `m0m1Txt`. Kolejne trzy przywiązują do zmiennej `SpecRadianceTxt` teksturę przefiltrowanej radiancji i wybierają poziom tej tekstury przechowujący dane otrzymane z odpowiednim parametrem chropowatości m. Przedostatnia instrukcja przywiązuje bufor magazynowy z wielomianową reprezentacją irradiancji dla odbicia rozproszonego, a ostatnia wybiera opis materiału.

Uwaga: Skonstruowana zgodnie z podanym tu opisem tekstura przefiltrowanej radiancji jest wielopoziomowa; poszczególne jej poziomy odpowiadają różnym wartościom parametru chropowatości. W świecie marzeń szader mógłby użyć funkcji `textureLod`, której dodatkowy parametr (w porównaniu z funkcją `texture`) jawnie wybiera poziom tekstury, z którego mają być odczytane teksele (i takie rozwiązanie jest podane w książce [25]). Kłopot polega na tym, że dla tekstur kostkowych ten parametr jest ignorowany i wartość funkcji `textureLod` jest obliczana zawsze na podstawie poziomu podstawowego, którym domyślnie jest poziom o największej rozdzielczości[30]. Wybrać poziom rozmycia radiancji odpowiedni dla chropowatości powierzchni rysowanego przedmiotu może aplikacja, ustawiając poziom podstawowy tekstury (`GL_TEXTURE_BASE_LEVEL`) za pomocą procedury `glTexParameteri`. Niestety, nie jest w ten sposób możliwe uzyskanie dostępu do pozostałych poziomów tekstury kostkowej, co przydałoby się podczas rysowania powierzchni, której chropowatość nie jest stała. Dla takich powierzchni jedynym wyjściem wydaje się utworzenie osobnych tekstur jednopoziomowych dla potrzebnych wartości parametru chropowatości i udostępnienie ich szaderowi poprzez osobne ewaluatory.

Rysunek 28.13 przedstawia czajniki, stalowy i miedziany, przedstawione wcześniej na rysunku 28.10. Jak widać, dominujący wpływ na realizm obrazów obiektów, których powierzchnie odbijają światło w sposób zwierciadlany, ma oświetlenie przez otoczenie.

Jeśli przedmiot otoczony przez inne przedmioty jest mały, to można zaniedbać *ich* oświetlenie przez światło odbite od tego przedmiotu. Ale aby znaleźć globalne oświetlenie sceny będące skutkiem wielokrotnych odbić światła między na przykład ścianami pomieszczenia

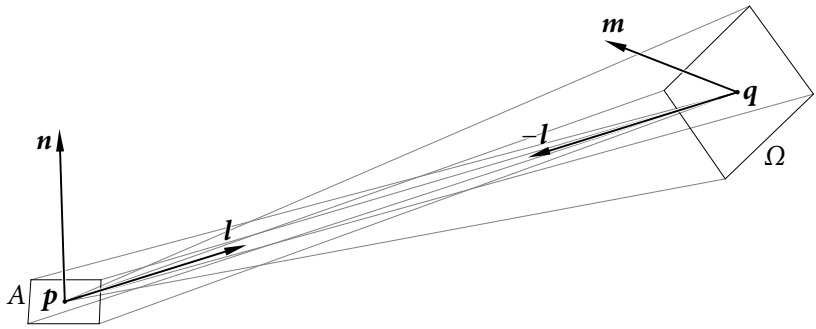
[30] Dla zwykłych tekstur dwuwymiarowych też tak jest, ale można to zmienić, wywołując procedurę `glTexParameteri` z parametrami `GL_TEXTURE_MIN_FILTER` i `GL_LINEAR_MIPMAP_LINEAR`. Niestety, dla tekstur kostkowych powoduje to błąd; uważam to za niedoróbkę standardu OpenGL lub błąd implementacji.

Rysunek 28.13. Czajniki metalowe oświetlone przez otoczenie i przez (punktowe) Słońce

i znajdującymi się w nim przedmiotami, trzeba rozwiązać numerycznie równanie bilansu energetycznego.

28.5. Równanie globalnego bilansu energetycznego

Wyprowadzimy równanie opisujące przepływ światła, które na powierzchniach obiektów w scenie wielokrotnie odbija się i załamuje, dzięki czemu może dotrzeć do wszystkich jej zakamarków.



Rysunek 28.14. Elementy powierzchni oświetlające się nawzajem

Rozważmy dwa elementy powierzchni A i Ω oraz dowolnie wybrane punkty $\boldsymbol{p}$ i $\boldsymbol{q}$ tych elementów (rys. 28.14), znajdujące się w odległości r. Jeśli średnice elementów są małe w porównaniu z ich odległością, to odległość każdego punktu elementu A od dowolnego punktu elementu Ω jest bliska r, różnica każdej takiej pary punktów ma kierunek bliski kierunku wektora $\boldsymbol{l} = \boldsymbol{q} - \boldsymbol{p}$, wreszcie kąt bryłowy elementu Ω widzianego z dowolnego punktu elementu A jest prawie taki sam i to samo dotyczy kątów bryłowych elementu A widzianego

z punktów elementu Ω. Wtedy strumień energetyczny światła dochodzącego[31] do elementu A z elementu Ω można wyrazić wzorem $L(\boldsymbol{q}, -\boldsymbol{l})\Omega_{\mathrm{s}}\psi_{\boldsymbol{q}A}$, w którym symbole Ω_{s} i $\psi_{\boldsymbol{q}A}$ oznaczają miarę skróconego elementu Ω widzianego z punktu $\boldsymbol{p}$ i miarę kąta bryłowego elementu A widzianego z punktu $\boldsymbol{q}$. Po podzieleniu strumienia przez pole elementu A otrzymamy irradiancję otoczenia punktu $\boldsymbol{p}$ światłem dochodzącym z elementu Ω (tj. z kierunku $\boldsymbol{l}$):

$$I(\boldsymbol{p}, \boldsymbol{l}) = L(\boldsymbol{q}, -\boldsymbol{l})\Omega_{\mathrm{s}}\psi_{\boldsymbol{q}A}/A.$$

Miara skróconego elementu Ω to $\Omega_{\mathrm{s}} = r^2\psi_{\boldsymbol{p}\Omega}$, a miara kąta $\psi_{\boldsymbol{q}A}$ jest równa $A|\cos\sphericalangle(\boldsymbol{n}, \boldsymbol{l})|/r^2$. Zatem

$$\begin{aligned} I(\boldsymbol{p}, \boldsymbol{l}) &= L(\boldsymbol{q}, -\boldsymbol{l})r^2\psi_{\boldsymbol{p}\Omega}\psi_{\boldsymbol{q}A}/A = L(\boldsymbol{q}, -\boldsymbol{l})r^2\psi_{\boldsymbol{p}\Omega}A|\cos\sphericalangle(\boldsymbol{n}, \boldsymbol{l})|/r^2/A \\ &= L(\boldsymbol{q}, -\boldsymbol{l})|\cos\sphericalangle(\boldsymbol{n}, \boldsymbol{l})|\psi_{\boldsymbol{p}\Omega}. \end{aligned}$$

Kąt bryłowy $\psi_{\boldsymbol{p}\Omega}$ odpowiada pewnemu wycinkowi sfery jednostkowej S, po której należy obliczyć całkę (28.2), aby uwzględnić oświetlenie elementu A z wszystkich stron. Stąd w punkcie $\boldsymbol{p}$ całkowita radiancja światła odbitego i załamanego w kierunku wektora $\boldsymbol{v}$ jest równa

$$L_{\mathrm{r}}(\boldsymbol{p}, \boldsymbol{v}) = \int_{\boldsymbol{l}\in S} \rho(\boldsymbol{p}; \boldsymbol{l}, \boldsymbol{v})L\big(\boldsymbol{q}(\boldsymbol{p}, \boldsymbol{l}), -\boldsymbol{l}\big)|\cos\sphericalangle(\boldsymbol{n}, \boldsymbol{l})|\,\mathrm{d}S.$$

Wartością funkcji $\boldsymbol{q}(\boldsymbol{p}, \boldsymbol{l})$ w tym wzorze jest punkt powierzchni (dowolnego obiektu sceny) widoczny z punktu $\boldsymbol{p}$ w kierunku wektora $\boldsymbol{l}$. Po wstawieniu otrzymanej całki do równania (28.1) powstaje równanie całkowe z niewiadomą funkcją L, opisujące bilans energetyczny światła w całej scenie:

$$L(\boldsymbol{p}, \boldsymbol{v}) = L_{\mathrm{e}}(\boldsymbol{p}, \boldsymbol{v}) + \int_{\boldsymbol{l}\in S} \rho(\boldsymbol{p}; \boldsymbol{l}, \boldsymbol{v})L\big(\boldsymbol{q}(\boldsymbol{p}, \boldsymbol{l}), -\boldsymbol{l}\big)|\cos\sphericalangle(\boldsymbol{n}, \boldsymbol{l})|\,\mathrm{d}S. \tag{28.21}$$

Aby otrzymać „prawdziwy" obraz oświetlonej sceny, trzeba dla każdego widocznego na obrazie punktu $\boldsymbol{p}$ powierzchni znaleźć wektor $\boldsymbol{v}$ w kierunku obserwatora (czyli w kierunku środka rzutowania perspektywicznego), obliczyć wartość funkcji $L(\boldsymbol{p}, \boldsymbol{v})$ i użyć jej do nadania koloru punktu na obrazie.

Metody numeryczne rozwiązywania równania (28.21), a także równań uwzględniających pochłanianie i rozpraszanie światła w nie całkiem przezroczystych ośrodkach (zobacz np. [35]) są fascynujące, ale w tej książce poprzestaniemy na jego wyprowadzeniu, zajmując się dalej tylko jego prostszym przypadkiem szczególnym. On też jest fascynujący.

[31] Zakładamy tu, że ośrodek, przez który przechodzi światło, na przykład powietrze, jest całkowicie przezroczysty. Opis, tj. model matematyczny skutków obecności mgły lub dymu, a także ośrodków, w których światło się rozprasza, takich jak mleczne szkło, jest bardziej skomplikowany.

29

*Lambertowski bilans energetyczny

Trudność w rozwiązywaniu równania (28.21) bierze się stąd, że dziedzina niewiadomej funkcji L jest iloczynem kartezjańskim sumy wszystkich powierzchni sceny i sfery jednostkowej, a więc zbiorem czterowymiarowym. Przyjęcie, że *wszystkie* te powierzchnie są lambertowskie (czyli idealnie gładkie i matowe) i nieprzezroczyste, przy czym oświetlona i możliwa do zobaczenia jest tylko jedna strona każdej powierzchni, umożliwia znaczne uproszczenie zadania, bo wtedy funkcje dane L_e i ρ i niewiadoma funkcja L zależą tylko od punktu na powierzchni.

Opisana w tym rozdziale implementacja metody bilansu energetycznego w możliwie dużym stopniu używa GPU do wszelkich obliczeń. Część z nich to preprocesing; jego zadaniem jest przygotowanie potrzebnych struktur danych i obliczenie współczynników macierzy układu równań liniowych, którego rozwiązanie daje przybliżenie funkcji L.

29.1. Globalne oświetlenie w modelu Lamberta

29.1.1. Równanie bilansu energetycznego

Zakładając, że wektor normalny każdej powierzchni w scenie jest skierowany w stronę, z której można tę powierzchnię oświetlać i oglądać, i dwukierunkowa funkcja odbicia ma po tej stronie wartość stałą (tj. niezależną od wektorów $\boldsymbol{l}$ i $\boldsymbol{v}$), możemy równanie (28.21) uprościć do postaci

$$L(\boldsymbol{p}) = L_\mathrm{e}(\boldsymbol{p}) + \rho(\boldsymbol{p}) \int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} L\big(\boldsymbol{q}(\boldsymbol{p},\boldsymbol{l})\big) \cos \measuredangle(\boldsymbol{n},\boldsymbol{l})\, \mathrm{d}S, \tag{29.1}$$

z całką po półsferze $S_{\boldsymbol{n}+}$ będącej zbiorem wektorów jednostkowych $\boldsymbol{l}$, takich że kosinus kąta między wektorami $\boldsymbol{n}$ (wektorem normalnym powierzchni w punkcie $\boldsymbol{p}$) a $\boldsymbol{l}$ jest nieujemny.

Źródłami światła w scenie mogą być elementy powierzchni, obecne mogą być też (albo tylko) punktowe źródła światła. Aby uwzględnić źródła punktowe, oświetlone przez nie elementy powierzchni traktujemy jak powierzchnie świecące, tj. przyjmujemy, że funkcja L_e jest sumą radiancji światła emitowanego przez powierzchnie i pomnożonej przez funkcję ρ irradiancji światła dochodzącego bezpośrednio od punktowych źródeł światła.

Całkę po półsferze $S_{\boldsymbol{n}+}$ możemy zamienić na całkę po wszystkich powierzchniach sceny. Niech punkt $\boldsymbol{q} = \boldsymbol{q}(\boldsymbol{p}, \boldsymbol{l})$ (położony na półprostej o początku $\boldsymbol{p}$ i kierunku wektora $\boldsymbol{l}$) należy do elementu powierzchni $\mathrm{d}\Omega$ o wektorze normalnym $\boldsymbol{m}$ (rys. 28.14). W polu widzenia punktu $\boldsymbol{p}$ element ten zajmuje kąt bryłowy

$$\mathrm{d}S = \frac{|\cos \measuredangle(\boldsymbol{m}, \boldsymbol{p}-\boldsymbol{q})|}{\|\boldsymbol{p}-\boldsymbol{q}\|^2}\,\mathrm{d}\Omega,$$

ale nie cały musi być widoczny z punktu $\boldsymbol{p}$. Dlatego trzeba do funkcji podcałkowej wprowadzić czynnik $v(\boldsymbol{p}, \boldsymbol{q})$ równy 1, jeśli między punktami $\boldsymbol{p}$ a $\boldsymbol{q}$ nie ma przeszkód, albo 0 jeśli są przeszkody (w szczególności, gdy $\cos \measuredangle(\boldsymbol{n}, \boldsymbol{q}-\boldsymbol{p}) < 0$ lub $\cos \measuredangle(\boldsymbol{m}, \boldsymbol{p}-\boldsymbol{q}) < 0$). Oznaczając symbolem Ω sumę wszystkich powierzchni sceny, możemy przekształcone równanie zapisać w postaci

$$L(\boldsymbol{p}) = L_{\mathrm{e}}(\boldsymbol{p}) + \rho(\boldsymbol{p}) \int_{\boldsymbol{q}\in\Omega} v(\boldsymbol{p}, \boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n}, \boldsymbol{q}-\boldsymbol{p}) \cos \measuredangle(\boldsymbol{m}, \boldsymbol{p}-\boldsymbol{q})}{\|\boldsymbol{p}-\boldsymbol{q}\|^2} L(\boldsymbol{q})\,\mathrm{d}\Omega. \qquad (29.2)$$

Równanie takiej postaci jest tzw. **równaniem całkowym Fredholma II rodzaju**; przed jego rozwiązywaniem wypada się upewnić, że rozwiązanie istnieje, do czego przyda się nam troszeczkę matematyki. Przyjmiemy, że powierzchnie, z których składa się scena, są ograniczone i można je podzielić skończenie wieloma gładkimi krzywymi (np. odcinkami) na kawałki, na których funkcje L_{e} i ρ są ciągłe[1]. Równanie (29.2) możemy zapisać w postaci

$$L = L_{\mathrm{e}} + \mathcal{K}L, \quad \text{albo} \quad (\mathcal{I} - \mathcal{K})L = L_{\mathrm{e}},$$

w której symbol $\mathcal{K} = \mathcal{D} \circ \mathcal{F}$ oznacza przekształcenie będące złożeniem dwóch przekształceń liniowych: operatora całkowego $\mathcal{F}$ określonego wzorem

$$(\mathcal{F}f)(\boldsymbol{p}) = \int_{\boldsymbol{q}\in\Omega} v(\boldsymbol{p}, \boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n}, \boldsymbol{q}-\boldsymbol{p}) \cos \measuredangle(\boldsymbol{m}, \boldsymbol{p}-\boldsymbol{q})}{\|\boldsymbol{p}-\boldsymbol{q}\|^2} f(\boldsymbol{q})\,\mathrm{d}\Omega$$

i przekształcenia $\mathcal{D}$, które dowolną funkcję f określoną w zbiorze Ω zamienia na iloczyn ρf,[2] a symbol $\mathcal{I}$ oznacza przekształcenie tożsamościowe (tj. $\mathcal{I}f = f$ dla każdej funkcji f). Operator $\mathcal{F}$ jest **wygładzający**, co oznacza w szczególności, że jeśli jego argument jest funkcją mierzalną[3] i ograniczoną na wspomnianych gładkich kawałkach powierzchni, to jego wartość jest funkcją ciągłą na każdym kawałku.

Zbiór funkcji ciągłych na rozważanych kawałkach powierzchni wyposażony w normę supremum jest przestrzenią Banacha[4]. W takiej przestrzeni określamy następujący **szereg**

[1] Dopuszczamy możliwość, że tylko pewna część gładkiej powierzchni świeci — wystarczy podzielić ją na część świecącą i resztę. Zakładamy, że funkcje L_{e} i ρ w każdym kawałku dają się rozszerzyć w sposób ciągły na brzeg tego kawałka. Wartości rozszerzonych w ten sposób funkcji na wspólnym brzegu kawałków uśredniamy.

[2] czyli w każdym punkcie $\boldsymbol{p} \in \Omega$ ma być $(\mathcal{D}f)(\boldsymbol{p}) = \rho(\boldsymbol{p}) f(\boldsymbol{p})$

[3] Rozważane tu funkcje spełniają ten warunek.

[4] Czyli zupełną unormowaną przestrzenią liniową. W przestrzeni zupełnej każdy ciąg zbieżny ma granicę, która jest elementem tej przestrzeni. Norma supremum funkcji rzeczywistej f o dziedzinie Ω jest to najmniejsza liczba x, taka że $|f(\boldsymbol{p})| \leqslant x$ dla każdego $\boldsymbol{p} \in \Omega$; oznaczamy ją symbolem $\|f\|_\infty$. Rozpatrujemy przestrzeń V złożoną z funkcji, które mogą być nieciągłe tylko w punktach ustalonych krzywych dzielących powierzchnie na kawałki. Przestrzeń z normą supremum *wszystkich* funkcji kawałkami ciągłych w zbiorze Ω nie jest zupełna.

Neumanna:

$$L_e + \mathcal{K}L_e + \mathcal{K}^2 L_e + \mathcal{K}^3 L_e + \cdots.$$

Jeśli ten szereg jest zbieżny, to jego granica jest funkcją kawałkami ciągłą i jest to poszukiwane rozwiązanie L równania całkowego (29.2).

Można udowodnić, co pominę[5], że jeśli maksymalna wartość funkcji ρ jest mniejsza niż $1/\pi$, to norma operatora $\mathcal{K}$ jest mniejsza niż 1 i wtedy szereg Neumanna jest zbieżny[6]. Jego kolejne składniki mają następującą interpretację: L_e opisuje światło wysłane przez świecące powierzchnie, $\mathcal{K}L_e$ odpowiada za światło, które po wysłaniu uległo jednemu odbiciu, $\mathcal{K}^2 L_e$ dwóm odbiciom itd. Ponieważ po każdym odbiciu światło jest słabsze, suma pierwszych kilkunastu lub nawet kilku składników szeregu jest dość dobrym przybliżeniem rozwiązania, uwzględniającym wybraną maksymalną liczbę odbić.

29.1.2. Metody dyskretyzacji

Równania całkowe takie jak (29.2) rozwiązuje się numerycznie, po dokonaniu **dyskretyzacji**. Polega ona na podzieleniu sceny (tj. zbioru Ω) na skończenie wiele **elementów** i założeniu szczególnej postaci funkcji, która ma być przybliżeniem dokładnego rozwiązania równania. Funkcję tę oznaczymy symbolem L_h.[7] Najprościej jest przyjąć, że elementy są trójkątami lub czworokątami i w każdym z nich funkcja L_h jest stała — wtedy mając n elementów, trzeba obliczyć n wartości tej funkcji.

Elementy oznaczymy symbolami $A_1, \ldots, A_n$; przyjmiemy, że są one płaskie i mają co najwyżej wspólne krawędzie. Wektor normalny elementu A_i oznaczę $\boldsymbol{n}_i$. Symbolem ϕ_i oznaczę funkcję, która ma wartość 1 w punktach należących do elementu A_i i 0 we wszystkich innych elementach[8]. Niech

$$L_h(\boldsymbol{p}) = \sum_{i=1}^{n} L_i \phi_i(\boldsymbol{p}), \quad L_{eh}(\boldsymbol{p}) = \sum_{i=1}^{n} L_{ei} \phi_i(\boldsymbol{p}).$$

Liczba L_i jest zatem wartością funkcji L_h w elemencie A_i, a funkcją L_{eh} o stałej wartości L_{ei} w elemencie A_i będziemy dalej w obliczeniach zastępować funkcję daną L_e.

Rozważymy dwa sposoby przekształcenia równania całkowego w układ n równań liniowych z n niewiadomymi. Pierwszy z nich to tzw. **metoda kolokacji**, w której w każdym elemencie wybieramy jeden punkt, na przykład środek ciężkości. Na niewiadome L_i nałożymy

[5] I słusznie!, jak zauważył mój kolega — ale zachęcam do spróbowania. Także studentów informatyki.

[6] Norma operatora liniowego $\mathcal{K}\colon V \to V$ jest to najmniejsza liczba y, taka że $\|\mathcal{K}f\|_\infty \leqslant y\|f\|_\infty$ dla wszystkich funkcji $f \in V$.

[7] Litera h w metodach numerycznych jest tradycyjnie używana do oznaczenia **parametru dyskretyzacji**, czyli największej średnicy elementu.

[8] Gwoli ścisłości przyjmujemy, że funkcja ϕ_i we wspólnym punkcie (brzegu) k elementów, jednym z których jest element A_i, ma wartość $\frac{1}{k}$. Dzięki temu w każdym punkcie dziedziny Ω suma wartości wszystkich funkcji ϕ_i jest równa 1.

warunek, że równanie (29.2) jest spełnione w tych punktach. Dla wybranego punktu $\boldsymbol{p}_i \in A_i$ dostaniemy wtedy równanie

$$L_i = L_\mathrm{e}(\boldsymbol{p}_i) + \rho(\boldsymbol{p}_i) \sum_{j=1}^{n} \left(\int_{\boldsymbol{q}\in A_j} v(\boldsymbol{p}_i, \boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n}_i, \boldsymbol{q}-\boldsymbol{p}_i) \cos \measuredangle(\boldsymbol{n}_j, \boldsymbol{p}_i-\boldsymbol{q})}{\|\boldsymbol{p}_i-\boldsymbol{q}\|^2} \,\mathrm{d}\Omega \right) L_j.$$

Wartość całki w nawiasie oznaczmy symbolem G_{ij}. Biorąc $L_{\mathrm{e}i} = L_\mathrm{e}(\boldsymbol{p}_i)$ dla wszystkich punktów $\boldsymbol{p}_i$, otrzymamy stąd układ równań liniowych, który możemy zapisać w postaci

$$L_i = L_{\mathrm{e}i} + \rho(\boldsymbol{p}_i) \sum_{j=1}^{n} G_{ij} L_j, \quad i = 1, \ldots, n.$$

Drugi sposób to tzw. **metoda Galerkina**; będziemy w niej poszukiwać takiej funkcji L_h, aby w każdym elemencie *wartość średnia* błędu (tzw. **residuum**, czyli różnicy stron równania (29.2) z funkcją L_h podstawioną w miejsce L) była równa 0. W tej metodzie dla każdego i całkujemy strony równania po elemencie A_i, a następnie dzielimy przez jego pole, otrzymując równanie

$$L_i = \frac{1}{A_i} \int_{\boldsymbol{p}\in A_i} L_\mathrm{e}(\boldsymbol{p}) + \rho(\boldsymbol{p}) \sum_{j=1}^{n} \left(\int_{\boldsymbol{q}\in A_j} v(\boldsymbol{p}, \boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n}_i, \boldsymbol{q}-\boldsymbol{p}) \cos \measuredangle(\boldsymbol{n}_j, \boldsymbol{p}-\boldsymbol{q})}{\|\boldsymbol{p}-\boldsymbol{q}\|^2} \,\mathrm{d}\Omega \right) L_j \,\mathrm{d}\Omega.$$

Zastępując funkcje ρ oraz L_e przez ich wartości średnie w elemencie A_i:

$$\rho_i = \frac{1}{A_i} \int_{\boldsymbol{p}\in A_i} \rho(\boldsymbol{p}) \,\mathrm{d}\Omega, \quad L_{\mathrm{e}i} = \frac{1}{A_i} \int_{\boldsymbol{p}\in A_i} L_\mathrm{e}(\boldsymbol{p}) \,\mathrm{d}\Omega,$$

otrzymamy (jeszcze trochę inne) równanie

$$L_i = L_{\mathrm{e}i} + \rho_i \sum_{j=1}^{n} \left(\frac{1}{A_i} \int_{\boldsymbol{p}\in A_i} \int_{\boldsymbol{q}\in A_j} v(\boldsymbol{p}, \boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n}_i, \boldsymbol{q}-\boldsymbol{p}) \cos \measuredangle(\boldsymbol{n}_j, \boldsymbol{p}-\boldsymbol{q})}{\|\boldsymbol{p}-\boldsymbol{q}\|^2} \,\mathrm{d}\Omega \,\mathrm{d}\Omega \right) L_j.$$

Mamy stąd układ równań liniowych

$$L_i = L_{\mathrm{e}i} + \rho_i \sum_{j=1}^{n} F_{ij} L_j, \quad i = 1, \ldots, n.$$

Występujące w nim liczby F_{ij} są nazywane **współczynnikami kształtu** (*form factors*).

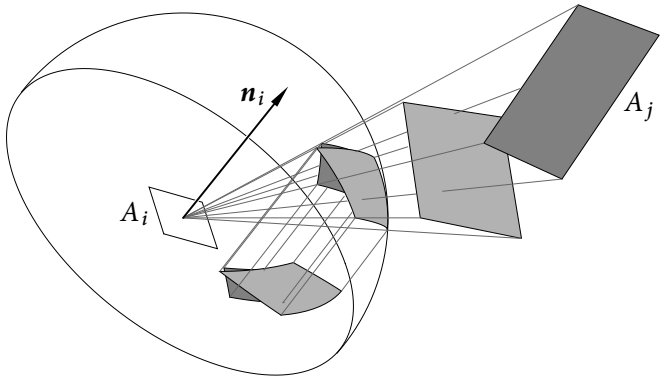
Możemy zauważyć, że $A_i F_{ij} = A_j F_{ji}$ dla każdego i, j, a ponadto $F_{ii} = 0$. W praktyce często współczynniki kształtu są zastępowane przez łatwiejsze do obliczenia liczby G_{ij}, które pojawiły się w metodzie kolokacji[9]. Liczby te nie muszą spełniać warunku $A_i G_{ij} = A_j G_{ji}$, a ponadto może się zdarzyć, że $G_{ij} = 0$, podczas gdy $F_{ij} \neq 0$, co może mieć negatywny

[9] Opisany dalej algorytm dokonuje uśredniania funkcji ρ i L_e w każdym elemencie, jest to więc implementacja metody Galerkina. Zastąpienie współczynników F_{ij} przez G_{ij} można rozumieć jako obliczenie F_{ij} za pomocą kwadratury opartej na jednym węźle — punkcie kolokacji.

wpływ na jakość otrzymanych obrazów. Całkę, której wartością jest liczba G_{ij}, można przekształcić tak, aby zbiorem całkowania był wycinek S_j półsfery $S_{\boldsymbol{n}+}$ zajmowany przez *widoczną z punktu* $\boldsymbol{p}_i$ *część* elementu A_j (porównaj równania (29.1) i (29.2)):

$$G_{ij} = \int_{\boldsymbol{q}\in A_j} v(\boldsymbol{p}_i, \boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n}_i, \boldsymbol{q}-\boldsymbol{p}_i) \cos \measuredangle(\boldsymbol{n}_j, \boldsymbol{p}_i-\boldsymbol{q})}{\|\boldsymbol{p}_i-\boldsymbol{q}\|^2} \,\mathrm{d}\Omega = \int_{\boldsymbol{l}\in S_j} \cos \measuredangle(\boldsymbol{n}_i, \boldsymbol{l}) \,\mathrm{d}S.$$

Całka ta jest równa polu obszaru otrzymanego przez dwukrotne rzutowanie widocznej części elementu A_j: dokonanie rzutu środkowego na półsferę jednostkową $S_{\boldsymbol{n}+}$ o środku $\boldsymbol{p}_i$, a potem rzutu prostopadłego otrzymanego wycinka S_j na płaszczyznę elementu A_i. Ten sposób obliczania współczynników G_{ij}, zwany **analogią Nusselta**, jest zilustrowany na rysunku 29.1. Obrazy wszystkich elementów widzianych z punktu $\boldsymbol{p}_i$ leżą w kole o promieniu 1, a zatem suma współczynników G_{ij} dla ustalonego i nie przekracza π.



Rysunek 29.1. Analogia Nusselta

Zamiast funkcji ρ o wartościach z przedziału $[0, 1/\pi)$ (zobacz podrozdz. 28.3) wygodniej jest przyjąć funkcję π razy większą (tj. o wartościach z przedziału $[0, 1)$), która określa ułamek mocy światła odbijanego przez poszczególne punkty powierzchni. Wtedy we wzorach definiujących współczynniki kształtu F_{ij} oraz G_{ij} należy wprowadzić dodatkowy czynnik $1/\pi$. Dla każdego i suma tak określonych współczynników kształtu jest mniejsza lub równa 1.

Uwaga: W literaturze na temat metody bilansu energetycznego funkcje dana i niewiadoma zazwyczaj opisują *emitancję* światła wypromieniowanego przez powierzchnie świecące i emitancję całkowitą — zastąpienie radiancji przez emitancję jest równoważne pomnożeniu stron równania (29.2) przez π (zobacz podrozdz. 28.1 i 28.3).

Oznaczmy symbolami $\boldsymbol{L}_\mathbf{e}$ i $\boldsymbol{L}$ wektory, których współrzędnymi są liczby $L_{\mathbf{e}i}$ oraz L_i. Niech D oznacza macierz diagonalną z liczbami ρ_i na przekątnej. Wszystkie te liczby odpowiadają ustalonej długości fali świetlnej i zamiast nich trzeba rozważać funkcje opisujące kolory światła i kolory (matowych) farb. Do reprezentowania kolorów możemy używać wektorów o trzech współrzędnych, r, g, b, opisujących składowe czerwoną, zieloną i niebieską.

Mamy zatem do rozwiązania trzy układy równań liniowych, każdy postaci

$$(I - DF)\boldsymbol{L} = \boldsymbol{L}_\mathrm{e}, \tag{29.3}$$

opisujące bilans energetyczny odpowiednio dla poszczególnych składowych. Macierz F jest we wszystkich tych układach ta sama, a za D podstawiamy macierze D_r, D_g, D_b utworzone ze współrzędnych kolorów farb (lub tekstur) nałożonych na poszczególne elementy.

Metoda eliminacji Gaussa *nie jest* odpowiednia do rozwiązywania takich układów równań z powodu zbyt dużej złożoności czasowej i pamięciowej. Znacznie lepiej sprawdzają się różne metody iteracyjne. W najprostszym przypadku, biorąc $\boldsymbol{L}_0 = \boldsymbol{L}_\mathrm{e}$ i obliczając

$$\boldsymbol{L}_k = \boldsymbol{L}_\mathrm{e} + DF\boldsymbol{L}_{k-1}, \quad k = 1, 2, 3, \ldots, \tag{29.4}$$

otrzymamy ciąg wektorów

$$\boldsymbol{L}_k = \boldsymbol{L}_\mathrm{e} + K\boldsymbol{L}_\mathrm{e} + \cdots + K^k\boldsymbol{L}_\mathrm{e},$$

reprezentujących kolejne sumy częściowe szeregu Neumanna dla reprezentowanego przez macierz $K = DF$ zdyskretyzowanego operatora $\mathcal{K}$ rozważanego wcześniej. Ciąg ten zbiega do poszukiwanego rozwiązania $\boldsymbol{L}$. Tyle matematyki.

Reszta jest liczeniem.

Marcin Kuczma

29.2. Implementacja

Implementacja składa się z preprocesingu, którego celem jest dokonanie dyskretyzacji, w tym odwzorowanie trójkątów w dziedzinę opisanej niżej **tekstury irradiancji**, powiązanie jej tekseli z poszczególnymi niewiadomymi i obliczenie potrzebnych macierzy, oraz koniecznego po każdej zmianie oświetlenia (np. włączeniu, wyłączeniu lub przemieszczeniu punktowego źródła światła) etapu rozwiązywania układu równań (29.3), którego wynikiem jest tekstura irradiancji reprezentująca bieżące oświetlenie.

Szader fragmentów używany do wykonywania końcowego obrazu powstał przez prostą modyfikację szadera z listingu 26.23 (zobacz też listing 28.5). Linia 62, w której jest dodawane podane w polu `ambient` (i przyjęte „na wyczucie") światło rozproszone w otoczeniu, została usunięta. Zamiast tego wartość przypisywana zmiennej `Colour` *przed* rozpoczęciem pętli ma być sumą radiancji światła emitowanego i światła odbitego przez dany punkt. Aby tak było, instrukcja w linii 59 na listingu została zamieniona na

```
Colour = MatEmission ( dot ( normal, vv ) ) +
             m.diffref.rgb * texture ( irradtex, In.IrrTxtCoord ).rgb;
```

Tekstura irradiancji, do której dostęp daje zmienna `irradtxt` typu `sampler2DRect`, jest określona w prostokącie. Wierzchołki rysowanych trójkątów mają dodatkowy atrybut o nazwie `IrrTxtCoord`, który zawiera współrzędne punktu tego prostokąta; szadery wierzchołków i geometrii muszą przekazać ten atrybut, aby szader fragmentów otrzymał wynik jego interpolacji. Teksele tekstury irradiancji odpowiadają elementom dyskretyzacji równa-

nia (29.2). Składowe r, g, b teksela odpowiadającego i-temu elementowi są współczynnikami w i-tym wierszu iloczynu macierzy F i macierzy $[\boldsymbol{L}_{\mathrm{r}T}, \boldsymbol{L}_{\mathrm{g}T}, \boldsymbol{L}_{\mathrm{b}T}]$, której kolumny są przybliżeniami rozwiązań układu równań (29.3) dla trzech składowych koloru, otrzymanymi po wykonaniu ustalonej liczby T iteracji wzoru (29.4). Tekstura irradiancji *pomija* zatem światło dochodzące *bezpośrednio* od źródeł punktowych.

Kolor obliczony przez szader fragmentów reprezentuje sumę radiancji światła wypromieniowanego przez punkt $\boldsymbol{p}$ powierzchni i pomnożonej przez $\rho(\boldsymbol{p})$ irradiancji światła, które dociera do tego punktu bezpośrednio ze źródeł punktowych (co jest opisane łącznie przez funkcję L_{e}) oraz pomnożonej przez $\rho(\boldsymbol{p})$ irradiancji światła, które dotarło do punktu $\boldsymbol{p}$ ze wszystkich widocznych z tego punktu powierzchni sceny. Dzięki temu, że ewaluator tekstury dokonuje interpolacji tekseli, to, że obliczona numerycznie irradiancja jest stała w każdym elemencie, staje się niewidoczne, a jakość otrzymanego obrazu uwzględnia szczegóły (opisujących funkcję ρ) tekstur nałożonych na obiekty i szczegóły kształtu cieni odpowiadających punktowym źródłom światła.

29.2.1. Podstawowe elementy implementacji

Fakt, że reprezentacja powierzchni obiektów, od których odbijają się fotony, składa się z płaskich trójkątów (których przetwarzanie jest specjalnością GPU) *nie oznacza* konieczności użycia tych trójkątów jako elementów dyskretyzacji. Duże trójkąty (np. na ścianach, suficie i podłodze pomieszczenia) i tak trzeba podzielić na mniejsze kawałki, a trójkąty bardzo małe lepiej jest połączyć, o ile to możliwe, w większe zespoły, które następnie zostaną podzielone na elementy. W tym celu zbiór trójkątów, z których składa się opis każdego obiektu, zostanie wstępnie podzielony na podzbiory, które nazwałem **płatami**. Trójkąty wchodzące w skład płata są połączone (przez wspólne krawędzie i wierzchołki) i są współpłaszczyznowe lub ich wektory normalne mają zbliżone kierunki, dzięki czemu można dokonać rzutowania prostopadłego na płaszczyznę tak, aby otrzymać trójkąty, których wnętrza są rozłączne[10].

Rzuty płatów na odpowiednie płaszczyzny zostają następnie poddane przekształceniom — podobieństwom geometrycznym — które rozmieszczają je na jednej płaszczyźnie. Prostokąt obejmujący rzuty wszystkich płatów jest dziedziną tekstury irradiancji, a jej teksele odpowiadają elementom dyskretyzacji równania (29.2), przy czym niektóre teksele są „nieużywane". Algorytm rozmieszczania płatów na płaszczyźnie ma na celu otrzymanie jak najmniejszej liczby takich tekseli.

Elementy dyskretyzacji są łączone w **makroelementy**, tj. zespoły elementów o nieco większych rozmiarach. Jest zatem n elementów i znacznie mniejsza liczba m makroelementów. Celem ich wprowadzenia jest oszczędność miejsca w pamięci i zmniejszenie widocznych na końcowych obrazach skutków niedokładnego obliczenia współczynników kształtu. Macierz

[10] Bliskość kierunków wektorów normalnych nie zapewnia możliwości takiego rzutowania; jeśli na przykład z tych trójkątów jest zbudowane przybliżenie powierzchni zwanej helikoidą, to rzuty trójkątów mogą się nakładać, co wymaga dokonania dodatkowego podziału zbioru trójkątów na mniejsze podzbiory. Opisana tu implementacja nie obsługuje tego przypadku, ale helikoidę można podzielić na płaty „ręcznie".

współczynników kształtu F we wzorze (29.4), będącym podstawą do obliczania oświetlenia, jest zastąpiona przez iloczyn opisanych niżej macierzy G i A.

Macierz A ma wymiary $m \times n$ i nieujemne współczynniki. Jej współczynnik a_{ij} jest dodatni, jeśli j-ty element należy do i-tego makroelementu. Wartość tego współczynnika jest ułamkiem pola makroelementu zajmowanego przez ten element, zatem suma współczynników w każdym wierszu jest równa 1. Niech $\boldsymbol{L} \in \mathbb{R}^n$ oznacza wektor, którego j-ta współrzędna (dla każdego j) jest wartością dowolnej funkcji L w j-tym elemencie. Iloczyn $A\boldsymbol{L}$ jest wektorem, którego i-ta współrzędna jest ważoną średnią wartości funkcji L w elementach należących do i-tego makroelementu.

Macierz G ma wymiary $n \times m$. Jej współczynnik G_{ij} jest współczynnikiem kształtu obliczonym na podstawie analogii Nusselta (rys. 29.1), przy czym element A_j, którego obraz w podwójnym rzutowaniu wyznacza wartość współczynnika, jest zastąpiony j-tym makroelementem. Sposób konstruowania tych macierzy jest opisany dalej.

Macierze D, G i A są rzadkie, tzn. mają niewiele współczynników niezerowych. Macierz D jest diagonalna, a jej współczynniki są wektorami o składowych r, g, b. Najwygodniej jest reprezentować ją za pomocą jednego bufora magazynowego zawierającego tablicę o długości n, w której elementach typu `vec4` są przechowywane współczynniki na diagonali[11]. Macierze G i A są reprezentowane w sposób opisany w podrozdziale G.4. Rozmieszczenie niezerowych współczynników macierzy G jest całkowicie nieregularne, natomiast macierz A ma w każdej kolumnie tylko jeden współczynnik niezerowy.

Na listingu 29.1 są pokazane podstawowe struktury danych używanych przez implementację. Struktura typu `SceneObject` zawiera opis obiektu, który umożliwia narysowanie go *bez* wykorzystania tekstury irradiancji; w szczególności są w nim pola `vao` i `vbo`, przechowujące identyfikatory obiektu tablicy wierzchołków i buforów z atrybutami wierzchołków, pola `nv` i `ntr` z liczbami wierzchołków i trójkątów oraz wskaźniki procedur (metod) rysowania (`redraw`) i sprzątania (`destroy`). Informacja zapisana w polu `index_type` to symbol typu indeksów w tablicy (`GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT` albo `GL_UNSIGNED_INT`), podawany w wywołanich procedury `glDrawElements` przez procedurę rysującą obiekt. Pola `mat0`, `nmat` i `txt0` identyfikują materiały i tekstury nałożone na obiekt.

Pole `previnst`, jeśli nie ma wartości `NULL`, przechowuje wskaźnik struktury opisującej inny „egzemplarz" takiego samego obiektu. Pozostałe pola obu obiektów mają takie same wartości (co oznacza, że mają wspólny obiekt tablicy wierzchołków, bufory i metody) z wyjątkiem pola `mm`, w którym jest pamiętana macierz przejścia od układu modelu do układu świata — każdy egzemplarz wypada umieścić gdzie indziej. Na przykład w scenie użytej do uruchomienia opisanej tu implementacji są trzy identyczne krzesła ustawione w różnych miejscach podłogi. Ponieważ każde z nich jest oświetlone inaczej, w strukturach danych implementacji metody bilansu energetycznego trójkąty każdej instancji muszą mieć osobne reprezentacje i w szczególności zajmować inne miejsca w dziedzinie tekstury irradiancji.

[11] W bloku o układzie `std430` elementy tablicy typu `vec3` są rozmieszczane tak, aby ich przesunięcia względem początku bufora były całkowitymi wielokrotnościami rozmiaru zmiennej typu `vec4`; można lepiej „upakować" dane w buforze, deklarując tablicę zmiennych typu `float` o długości $3n$, kosztem pewnego spowolnienia dostępu do danych. Ale przyjęcie typu `vec4` umożliwiło uproszczenie kodu szaderów.

Listing 29.1. Definicje struktur danych implementacji

C

```
typedef struct TriangDesc {
    GLuint  ind[3];
    GLint   matnum, trp;
    GLfloat nvect[3];
  } TriangDesc;

typedef struct TriangPatch {
    int   objid;
    int   nvclass, nvert, ftr, ntr;
    float x0, y0, w, h;
  } TriangPatch;

typedef struct SceneObject {
    struct SceneObject   *previnst;
    struct BalanceObject *bobj;
    char                 name[20];
    GLuint               vao, vbo[3];
    int                  nvert, ntr;
    GLenum               index_type;
    GLfloat              mm[16];
    int                  mat0, nmat, txt0;
    RedrawObject         redraw;
    DestroyObject        destroy;
    char                 active;
  } SceneObject;

typedef struct BalanceObject {
    SceneObject *obj;
    int         cnvert, ftrdesc, cntr, ntrpatch;
    GLint       txts;
    GLfloat     *vertpos, *txc, *txb;
    TriangDesc  *trdesc;
    TriangPatch *trpatch;
    float       eld;
    GLfloat     mmti[16];
  } BalanceObject;
```

Pole `bobj` przechowuje wskaźnik do struktury typu `BalanceObject`, będącej opakowaniem danych potrzebnych do implementacji metody bilansu energetycznego. Jej pole `obj` wskazuje opisaną wcześniej strukturę reprezentującą obiekt; struktury obu typów „wskazują na siebie nawzajem", przy czym reprezentacje pewnych obiektów, takich jak płomień świecy, mogą nie mieć struktury typu `BalanceObject`. Płomień, choć rysowany jako zbiór trójkątów, jest przezroczysty[12], a że jest mały, uznałem go za punktowe źródło światła.

[12] Prawdziwy płomień, choć to rozżarzona sadza, cienia nie rzuca. Zbadałem to przy użyciu lasera i świeczki.

Wartość pola `txts` służy do wybrania sposobu określania współrzędnych tekstury dla wierzchołków trójkątów obiektu. W polu `eld` jest pamiętana określona przez aplikację wielkość elementów otrzymanych z podziału powierzchni obiektu. Na przykład dla ścian pomieszczenia (pokoju o wymiarach $6 \times 4 \times 3$) elementy są kwadratami o boku 0.1 (jednostce długości w układzie modelu odpowiada 1 m, a więc bok elementu ma długość 10 cm), a dla stołu przyjąłem wielkość elementu 0.05 (czyli 5 cm). Pole `mmti` służy do zapamiętania transpozycji odwrotności macierzy przejścia od układu modelu do układu świata.

Struktury opisanych tu typów są przechowywane w tablicach. Tablice wskazywane przez pola `vertpos`, `txc` i `txb` struktury typu `BalanceObject` zawierają podane przez aplikację współrzędne położenia wierzchołków trójkątów (w układzie modelu) i współrzędne tekstury nałożonej na obiekt oraz nadawane podczas preprocesingu współrzędne tekstury irradiancji.

Opis trójkąta przechowywany w strukturze typu `TriangDesc` składa się z trójki indeksów wierzchołków (na podstawie których są obliczane indeksy do tablic wskazywanych przez pola `vertpos`, `txc` i `txb`), numeru materiału, numeru płata, do którego trójkąt został zaliczony (indeksu do tablicy `triangset`), i wektora normalnego płaszczyzny trójkąta.

Na opis płata w strukturze typu `TriangPatch` składają się: pole `objid`, które przechowuje numer obiektu (indeks do tablicy `objtab` struktur typu `BalanceObject`), pole `nvclass` opisane dalej, pole `nvert`, w którym pamiętana jest liczba wierzchołków trójkątów danego płata, pola `ftr` i `ntr`, w których jest zapisany indeks (do tablicy `trdesc`) pierwszego trójkąta i liczba trójkątów płata (opisy tych trójkątów zajmują w tablicy miejsca obok siebie), oraz pola `x0`, `y0`, `w` i `h` używane przez algorytm przydzielania wierzchołkom trójkątów położeń w dziedzinie tekstury irradiancji.

Listing 29.2. Opakowanie danych implementacji bilansu energetycznego

C

```
typedef struct {
    BalanceObject   objtab[MAX_OBJECTS], *current;
    int             nobj, ntrpatch, ntr, nvert, nelem, nmacroelem;
    float           ffnear, fffar;
    GLfloat         *vertpos, *txc, *txb;
    TriangDesc      *trdesc;
    TriangPatch     *trpatch;
    GLuint          tvao, tvbo[18];
    GLuint          irrtxt, fftxt[2];
    GLsizei         irrtxt_width, irrtxt_height;
    FFTransBl       fftr;
    GPUSparseMatrix avgmat, ffmat;
    GLuint          *VarMap, *VarBuf, *ObjIdBuf;
  } BalanceElements;
```

Opakowaniem całości danych używanych w opisanej tu implementacji metody bilansu energetycznego jest struktura o nazwie typu `BalanceElements` pokazana na listingu 29.2. Pierwsze jej pole jest tablicą struktur opisujących obiekty, których liczba jest pamiętana w polu `nobj`. Kolejne pola typu `int` przechowują liczby płatów, trójkątów, wierzchołków,

elementów i makroelementów. Pola `ffnear` i `fffar` przechowują zakres głębokości dla procedury obliczania współczynników kształtu. Pola `trdesc`, `triangset`, `vertpos`, `txb` i `txc` przechowują wskaźniki tablic, w których po zakończeniu preprocesingu znajdą się opisy wszystkich trójkątów i płatów. W kolejnych polach są pamiętane identyfikatory używanych przez implementację buforów i tekstur, a w polach `irrtxt_width` i `irrtxt_height` są przechowywane wymiary tekstury irradiancji.

Pole `fftr` opisanego w punkcie 29.2.6 typu `FFTransBl` służy do zapamiętania macierzy przekształceń używanych podczas obliczania współczynników kształtu. Pola `avgmat` i `ffmat` (zobacz listing G.13) są opakowaniami macierzy A i G opisanych na początku tego punktu. Pomocnicze tablice, opisane dalej, są wskazywane przez wartości pól `VarMap`, `VarBuf` i `ObjIdBuf`.

Będące częścią implementacji programy szaderów kompiluje i przygotowuje do pracy procedura `LoadBalanceShaders`, którą należy wywołać na początku działania aplikacji. Do ich sprzątania (tj. likwidacji) służy procedura `DeleteBalancePrograms`. Obie procedury nie mają parametrów; identyfikatory programów i odczytane z nich dane są pamiętane w statycznych zmiennych globalnych.

29.2.2. Wprowadzanie trójkątów

Listing 29.3 przedstawia nagłówki procedur, przy użyciu których aplikacja wprowadza trójkąty do struktur danych implementacji metody bilansu energetycznego. Pierwszy parametr tych procedur ma wskazywać zmienną opisanego wcześniej typu, która może być zadeklarowana statycznie lub utworzona przy użyciu procedury `malloc`. Proces wprowadzania danych musi się zacząć od wywołania procedury `BeginEnteringBalanceElements`. Po wprowadzeniu trójkątów wszystkich obiektów należy wywołać procedurę `EndEnteringBalanceElements`, której zadaniem jest dokończenie preprocesingu.

Listing 29.3. Procedury wprowadzania danych

```C
void BeginEnteringBalanceElements ( BalanceElements *belem,
                                    float ffnear, float fffar );
void EndEnteringBalanceElements ( BalanceElements *belem );

BalanceObject *BeginEnteringObjTriangles ( BalanceElements *belem,
                                           SceneObject *obj, GLint txts,
                                           int nvert, int ntr, float eld );
void EnterTriangles ( BalanceElements *belem, GLenum mode,
                      int nvert, GLfloat vpos[][3], GLfloat txc[][2],
                      int nind, GLenum itype, GLvoid *index, GLint matnum,
                      char onepatch );
void EndEnteringObjTriangles ( BalanceElements *belem );

void EnterNewObjElemInstance ( BalanceElements *belem, SceneObject *inst );
```

Wprowadzanie trójkątów każdego obiektu zaczyna się od wywołania procedury `BeginEnteringObjTriangles`, której drugi parametr wskazuje strukturę z (przygotowanym przez aplikację wcześniej) opisem obiektu. Wartość trzeciego parametru jest zapamiętywana w polu `txts` opisu obiektu. Czwarty i piąty parametr określają całkowite liczby wierzchołków i trójkątów, a szósty opisaną wcześniej wielkość elementu dyskretyzacji równania (29.2). Procedura `EndEnteringObjTriangles`, wywołana po wprowadzeniu wszystkich trójkątów obiektu, wykonuje związane z nimi obliczenia stanowiące pierwszą część preprocesingu. Te obliczenia wykonuje w całości CPU, a ponieważ kod wykonujących je procedur liczy kilkaset linii, zamiast listingu przedstawiam tylko krótki opis realizowanego przez nie algorytmu.

Trójkąty są wprowadzane przez jedno lub więcej wywołań procedury `EnterTriangles`. Tablicę `vertpos` ze współrzędnymi położeń *wszystkich* wierzchołków i tablicę `txc` ze współrzędnymi tekstury trzeba podać w pierwszym wywołaniu tej procedury (jeśli trójkąty nie są teksturowane, parametr `txc` ma mieć wartość `NULL`); w kolejnych wywołaniach należy podawać tylko indeksy (w tablicach wskazywanych przez parametr `index`). Parametr `mode` może mieć wartość `GL_TRIANGLES`, `GL_TRIANGLE_STRIP` lub `GL_TRIANGLE_FAN`. Zawartość tablicy indeksów ma być taka sama jak zawartość bufora z indeksami wierzchołków używanego podczas rysowania trójkątów za pomocą procedury `glDrawElements`. Można podać indeksy typu `GLubyte`, `GLushort` lub `GLuint`; parametr `type` ma mieć wartość `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT` albo `GL_UNSIGNED_INT`. Jeśli parametr `mode` ma wartość `GL_TRIANGLE_STRIP`, to w tablicy indeksów można umieścić indeksy o wartości specjalnej, odpowiednio `0xFF`, `0xFFFF` albo `0xFFFFFFFF`, która spowoduje restart prymitywu (p. 12.4.2). „Wewnętrznie" indeksy są przechowywane w zmiennych typu `GLuint`.

Parametr `matnum` jest numerem materiału, którego opis (zawierający wektor `dirref`, którego współrzędne $r, g, b \in [0,1)$ są ułamkami mocy odbijanego światła czerwonego, zielonego i niebieskiego, lub numer tekstury pełniącej analogiczną rolę) musi być przygotowany przed wywołaniem procedury `BeginEnteringObjTriangles`. Poszczególne trójkąty obiektu mogą być wykonane z różnych materiałów; trójkąty z każdego materiału wprowadza się osobnym wywołaniem procedury `EnterTriangles`.

Jeśli parametr `onepatch` ma wartość `true`, to z trójkątów wprowadzonych w danym wywołaniu powstanie jeden płat. Zbiór trójkątów wprowadzonych z tym parametrem o wartości `false` zostanie podzielony na płaty w sposób opisany w następnym punkcie.

Procedura `EnterNewObjElemInstance` służy do wprowadzenia kolejnych instancji obiektu wprowadzonego wcześniej. Parametr `inst` wskazuje strukturę typu `SceneObject`, której pole `previnst` wskazuje pierwszą instancję obiektu wcześniej wprowadzonego (przy użyciu procedur opisanych wyżej) do struktury danych bilansu energetycznego. Procedura ta tworzy kopię wyników preprocesingu tej pierwszej instancji.

29.2.3. Preprocesing trójkątów obiektu

Płat jest zbiorem trójkątów połączonych wspólnymi wierzchołkami i takim, że po dokonaniu rzutu prostopadłego na odpowiednio wybraną płaszczyznę powstaje zbiór trójkątów mogących mieć wspólne tylko wierzchołki lub krawędzie. Pierwszym etapem preprocesingu trój-

kątów należących do obiektu jest ich klasyfikacja, tj. podział na podzbiory, które po zrzutowaniu na odpowiednie płaszczyzny dają trójkąty o zgodnej orientacji[13]. Płaty powstaną z podziału znalezionych klas na mniejsze podzbiory.

Dla każdego trójkąta jest więc obliczany (w układzie modelu) jego wektor normalny, jako iloczyn wektorowy wektorów dwóch krawędzi. Klasyfikacja jest dokonywana za pomocą sześciu wektorów, $\boldsymbol{e}_1$, $\boldsymbol{e}_2$, $\boldsymbol{e}_3$, $-\boldsymbol{e}_1$, $-\boldsymbol{e}_2$ i $-\boldsymbol{e}_3$, gdzie wektor $\boldsymbol{e}_i$ ma i-tą współrzędną równą 1 i pozostałe dwie zerowe[14]. Trójkąt zostaje zaliczony do tej klasy, z której wektorem jego wektor normalny tworzy najmniejszy kąt (czyli ma największy iloczyn skalarny). Następnie tablica trójkątów zostaje posortowana ze względu na numery klas.

Poszczególnym płatom zostaną przydzielone inne (być może odległe) miejsca w dziedzinie tekstury irradiancji. Dlatego wierzchołki, które są wspólne dla trójkątów należących do różnych klas są kopiowane tak, aby każdy wierzchołek w tablicach wskazywanych przez pola `vertpos` i `txc` struktury opisującej obiekt był wierzchołkiem trójkątów tylko z jednej klasy. Odpowiednio do tego są modyfikowane indeksy wierzchołków w opisujących trójkąty strukturach typu `TriangDesc` (w tablicy wskazywanej przez pole `trdesc` struktury `BalanceObject`).

Dla każdej klasy jest określony **graf sąsiedztwa**: trójkąty są jego wierzchołkami, a krawędzie grafu łączą te trójkąty, które mają wspólny wierzchołek. Każdy płat odpowiada pewnej składowej spójnej grafu sąsiedztwa.

Do znajdowania składowych spójnych jest użyty algorytm łączenia drzew w lesie zbiorów rozłącznych, opisany w rozdziale 21 książki [54]. W skrócie: dla zbioru trójkątów danej klasy tworzony jest pomocniczy graf bez krawędzi, którego wierzchołkami są poszczególne trójkąty. Ten graf i grafy otrzymane z niego są lasami, tj. grafami bez cykli. Zatem, początkowo wszystkie składowe spójne grafu pomocniczego reprezentują jednoelementowe zbiory trójkątów. Po wykryciu wspólnego *wierzchołka trójkątów* reprezentowanych przez *wierzchołki grafu pomocniczego* należące do różnych składowych spójnych dodawana jest krawędź łącząca te składowe. Identyfikatory składowych danych na początku są numerami (tj. indeksami do tablicy) trójkątów; po połączeniu dwóch składowych jeden z tych numerów „przechodzi" na wynik połączenia, a drugi numer przestaje być identyfikatorem składowej.

Składowe spójne grafu pomocniczego są drzewami; dla każdego wierzchołka grafu jest pamiętany numer *pewnego* wierzchołka (czyli numer pewnego trójkąta), przy czym jeśli to jest numer tego samego wierzchołka, to jest on korzeniem drzewa i jego numer jest identyfikatorem składowej spójnej, a w przeciwnym przypadku jest to numer kolejnego wierzchołka na ścieżce do korzenia. Gdy dwa drzewa mają być połączone, korzeń jednego z nich jest „podczepiany" pod korzeń drugiego (przez przypisanie jego numeru; w ten sposób powstaje nowa krawędź). Dla każdego wierzchołka grafu jest pamiętana tzw. **ranga**, będąca oszacowaniem wysokości drzewa, którego ten wierzchołek jest korzeniem. Zawsze drzewo, którego korzeń ma rangę mniejszą, staje się poddrzewem korzenia drugiego drzewa, a jeśli rangi są

[13] Przypominam, że z tych trójkątów składają się powierzchnie nieprzezroczystych brył, przy czym orientacja każdego trójkąta (określona przez kolejność wierzchołków) jest taka, że jego wektor normalny jest skierowany na zewnątrz bryły.

[14] Można wprowadzić więcej klas, dołączając na przykład wektory jednostkowe o kierunkach przekątnych sześcianu. Skutki tego są warte eksperymentalnego zbadania.

równe, to ranga wierzchołka, który staje się korzeniem drzewa połączonego, jest zwiększana o 1. Podczas badania, czy dane dwa wierzchołki należą do różnych drzew, i podczas łączenia jest też dokonywana tzw. **kompresja ścieżki**, tj. przebudowa drzew, dzięki której ich wysokości pozostają bardzo małe.

Drzewa (składowe spójne) w lesie otrzymanym na końcu reprezentują płaty; dla każdego trójkąta w polu `trp` zapamiętywany jest identyfikator składowej, do której należy ten trójkąt. Tablica trójkątów jest sortowana według tych numerów, dzięki czemu wszystkie opisy trójkątów każdego płata znajdują się w tablicy obok siebie i w strukturze typu `TriangPatch` będącej elementem tablicy wskazywanej przez pole `trpatch` wystarczy zapamiętać położenie (indeks) pierwszego trójkąta i liczbę trójkątów płata opisanego przez tę strukturę.

Kolejny krok preprocesingu kończy przygotowania do przydzielania wierzchołkom trójkątów współrzędnych w dziedzinie tekstury irradiancji. Dla każdego płata jest wyznaczana w przestrzeni płaszczyzna, na którą płat ma być rzutowany. Dla zbioru punktów będących rzutami wierzchołków trójkątów płata znajdowana jest otoczka wypukła (przy użyciu opisanego w rozdziale 33 książki [54] **algorytmu Grahama**). Następnie znajdowany jest taki obrót w płaszczyźnie, aby opisany na otoczce wypukłej prostokąt o bokach równoległych do osi układu współrzędnych miał najmniejsze pole. Współrzędne x i y rzutów wierzchołków są skalowane, przy czym czynnik skalowania każdej współrzędnej jest odwrotnością wartości parametru `eld` podanego dla obiektu, zmodyfikowaną w taki sposób, aby różnice minimalnych i maksymalnych współrzędnych x i y były liczbami całkowitymi. Po obróceniu i przeskalowaniu rzuty wierzchołków są przesuwane tak, aby najmniejsze współrzędne x i y otrzymanych punktów były równe 1. Otoczka wypukła rzutu płata na płaszczyznę zostaje wpisana w prostokąt, którego szerokość i wysokość pozostawiają margines o szerokości 1.[15] Wymiary tego prostokąta zostają zapamiętane w polach `w` i `h` struktury `TriangPatch` opisującej płat. W tablicy `txb` zapamiętywane są współrzędne rzutów wierzchołków, obróconych, przeskalowanych i przesuniętych. Dalsze etapy preprocesingu następują po wprowadzeniu i przetworzeniu w opisany wyżej sposób trójkątów wszystkich obiektów sceny.

29.2.4. Obliczanie współrzędnych w dziedzinie tekstury irradiancji

Gdy opisy trójkątów i płatów poszczególnych obiektów są już utworzone w sposób przedstawiony w poprzednim punkcie, kolejnym krokiem jest przepisanie ich do tablic wspólnych dla wszystkich obiektów. Po obliczeniu sum liczb trójkątów, płatów i wierzchołków następuje rezerwacja tablic o odpowiednich długościach; ich adresy są przypisywane polom `trdesc`, `trpatch`, `vertpos`, `txc` i `txb` opakowania danych (typu `BalanceElements`, zobacz listing 29.2), a potem dane z tablic poszczególnych obiektów są przepisywane do tablic wspólnych. Do indeksów wierzchołków w opisach trójkątów każdego obiektu jest dodawana suma liczb wierzchołków obiektów przepisanych wcześniej. Do indeksu pierwszego trójkąta każdego płata jest dodawana suma liczb trójkątów obiektów przepisanych wcześniej. Po przepisaniu danych każdego obiektu jego tablice są zwalniane, a polom `vertpos`, `txc`, `txb` i `trdesc` opisu obiektu są przypisywane adresy początków tablic wspólnych. Pole `trpatch`

[15] Cel wprowadzenia tego marginesu wyjaśni się dalej.

opisu obiektu otrzymuje adres pierwszego elementu tablicy wspólnej opisów płatów zawierającego opis płata tego obiektu.

Sposób rozmieszczania płatów na płaszczyźnie polega na zbudowaniu drzewa binarnego, którego liśćmi są płaty. Drzewo jest budowane od dołu: początkowo każdy płat jest też korzeniem drzewa w lesie. Korzenie te są umieszczane w kolejce priorytetowej. Dla każdego płata, a także dla wierzchołków drzewa tworzonych później, dane są wymiary prostokąta otaczającego (ustalone po znalezieniu i odpowiednim obróceniu otoczki wypukłej). Wymiary te są liczbami całkowitymi, przy czym jeśli wysokość jest większa niż szerokość, to się je przestawia. Większy priorytet ma wierzchołek, którego prostokąt ma mniejszą szerokość, a jeśli szerokości są równe, to ten, którego wysokość jest mniejsza. Kolejka priorytetowa jest zrealizowana za pomocą kopca (zobacz rozdział 6 książki [54]).

W pętli z kolejki priorytetowej są wyjmowane dwa wierzchołki, będące korzeniami drzew w lesie, po czym jest tworzony nowy wierzchołek, pod który te wierzchołki są podczepiane. Prostokąt otaczający nowego wierzchołka ma szerokość równą większej z szerokości prostokątów podczepianych wierzchołków, a wysokość równą sumie wysokości tych prostokątów. Jeśli tak otrzymana wysokość jest większa od szerokości, to wymiary te są przestawiane. Nowy wierzchołek (korzeń nowego drzewa) jest wstawiany do kolejki priorytetowej. Działanie pętli kończy się, gdy w kolejce priorytetowej pozostał tylko jeden wierzchołek, będący oczywiście korzeniem całego drzewa. Szerokość w i wysokość h prostokąta otaczającego korzeń, zapamiętywane w polach `irrtxt_width` i `irrtxt_height`, są wymiarami (w tekselach) tekstury irradiancji, która będzie utworzona później.

Rozmieszczanie płatów w dziedzinie tekstury irradiancji następuje podczas przeszukiwania drzewa w głąb; porównanie wymiarów prostokąta otaczającego dany wierzchołek z wymiarami prostokąta otaczającego prostokąt wierzchołka „wyżej" jest podstawą do podjęcia decyzji, czy należy przestawić szerokość z wysokością, aby pierwszy prostokąt zmieścił się w drugim. Dla każdego wierzchołka są obliczane współrzędne przesunięcia jego dolnego lewego narożnika względem narożnika prostokąta otaczającego korzeń drzewa. Po dojściu do liścia (reprezentującego płat) współrzędne przesunięcia są zapamiętywane w polach `x0` i `y0` opisu płata (struktury typu `TriangPatch`).

Jeśli liczba przestawień wymiarów po drodze od korzenia do liścia reprezentującego płat była nieparzysta, to dla każdego wierzchołka trójkątów tego płata dwie liczby zapamiętane w elemencie tablicy `txb` są przestawiane. Następnie dodawane są do nich wartości pól `x0` i `y0` opisu płata i wtedy wierzchołek otrzymuje swoje położenie w dziedzinie tekstury irradiancji — atrybut `IrrTxtCoord`. Przykładowy wynik rozmieszczania płatów opisaną tu metodą jest pokazany na rysunku 29.2 (s. 816).

29.2.5. Konstrukcja elementów i makroelementów

Kolejne etapy preprocesingu są wykonywane z użyciem GPU. Po wykonaniu obliczeń opisanych w poprzednich punktach trzeba określić, które teksele tekstury irradiancji odpowiadają elementom dyskretyzacji równania całkowego. W tym celu wszystkie trójkąty trzeba narysować w pomocniczym pozaekranowym buforze ramki.

Listing 29.4. Procedura tworzenia obiektu tablicy wierzchołków

C

```
1: static void SetupBElemVAO ( BalanceElements *belem )
2: {
3:   GLuint *ind;
4:   int    i, j, k, ntr;
5:
6:   glGenVertexArrays ( 1, &belem->tvao );
7:   glBindVertexArray ( belem->tvao );
8:   glGenBuffers ( 4, belem->tvbo );
9:   glBindBuffer ( GL_ARRAY_BUFFER, BUF_VERTPOS );       /* belem->tvbo[0] */
10:  glBufferData ( GL_ARRAY_BUFFER, belem->nvert*3*sizeof(GLfloat),
11:                 belem->vertpos, GL_STATIC_DRAW );
12:  glEnableVertexAttribArray ( 0 );
13:  glVertexAttribPointer ( 0, 3, GL_FLOAT, GL_FALSE,
14:                           3*sizeof(GLfloat), (GLvoid*)0 );
15:  glBindBuffer ( GL_ARRAY_BUFFER, BUF_TXC );          /* belem->tvbo[1] */
16:  glBufferData ( GL_ARRAY_BUFFER, belem->nvert*2*sizeof(GLfloat),
17:                 belem->txc, GL_STATIC_DRAW );
18:  glEnableVertexAttribArray ( 1 );
19:  glVertexAttribPointer ( 1, 2, GL_FLOAT, GL_FALSE,
20:                           2*sizeof(GLfloat), (GLvoid*)0 );
21:  glBindBuffer ( GL_ARRAY_BUFFER, BUF_TXB );          /* belem->tvbo[2] */
22:  glBufferData ( GL_ARRAY_BUFFER, belem->nvert*2*sizeof(GLfloat),
23:                 belem->txb, GL_STATIC_DRAW );
24:  glEnableVertexAttribArray ( 2 );
25:  glVertexAttribPointer ( 2, 2, GL_FLOAT, GL_FALSE,
26:                           2*sizeof(GLfloat), (GLvoid*)0 );
27:  ntr = belem->ntr;
28:  if ( !(ind = malloc ( 3*ntr*sizeof(GLuint) )) )
29:    ExitOnError ( "SetupBElemVAO" );
30:  for ( i = k = 0;  i < ntr;  i++ )
31:    for ( j = 0; j < 3; j++ )
32:      ind[k++] = belem->trdesc[i].ind[j];
33:  glBindBuffer ( GL_ELEMENT_ARRAY_BUFFER, BUF_ELIND );  /*belem->tvbo[3] */
34:  glBufferData ( GL_ELEMENT_ARRAY_BUFFER, ntr*3*sizeof(GLuint),
35:                 ind, GL_STATIC_DRAW );
36:  free ( ind );
37:  ExitIfGLError ( "SetupBElemVAO" );
38: } /*SetupBElemVAO*/
```

Listing 29.4 przedstawia procedurę, której zadaniem jest utworzenie obiektu tablicy wierzchołków używanego nie tylko w preprocesingu, ale także podczas wykonywania końcowych obrazów sceny. W celu poprawienia czytelności kodu poszczególne elementy tablicy `tvbo`, w której pamiętane są identyfikatory buforów w pamięci GPU używanych przez implementację, zostały ukryte pod znaczącymi nazwami makrodefinicji; zamiast na przykład wyrażenia `belem->tvbo[0]` w całym kodzie źródłowym implementacji napisałem bardziej znaczący identyfikator `BUF_VERTPOS`. Zatem, do buforów ukrytych pod nazwami `BUF_`-

VERTPOS, BUF_TXC i BUF_TXB są wpisywane odpowiednio współrzędne położeń wierzchołków trójkątów w przestrzeni (w układzie modelu każdego obiektu), współrzędne tekstury nałożonej na obiekty i współrzędne wierzchołków w dziedzinie tekstury irradiancji. Bufor BUF_ELIND zawiera przepisane z opisów trójkątów indeksy wierzchołków.

Preprocesing jest wykonywany przy użyciu kilkunastu programów szaderów, zarówno realizujących potok przetwarzania grafiki, jak i takich z szaderami obliczeniowymi. Mają one wiele zmiennych jednolitych o identycznych nazwach i spełnianej roli, dlatego zamiast deklarować je osobno w domyślnych blokach zmiennych jednolitych, umieściłem je wszystkie w przedstawionym na listingu 29.5 bloku zmiennych jednolitych w buforze doczepionym do odpowiedniego punktu dowiązania w celu GL_UNIFORM_BUFFER. Zamiast procedur z rodziny glUniform* użyłem własnych procedur przypisujących potrzebne wartości zmiennym w tym bloku. Mają one nazwy CTLUniformi, CTLUniformui i CTLUniformf.

Listing 29.5. Blok zmiennych jednolitych szaderów preprocesingu

GLSL

```
uniform CtlBlock {
    int   stage, step, width, height, N, H, nrows, ncols, first, txts;
    uint  nelem, p0, mi, nnz;
    bool  reverse;
    float C;
    vec3  colour;
  } ctl;
```

Większość danych przetwarzanych w celu przygotowania obliczeń bilansu energetycznego jest uporządkowana jednym z dwóch sposobów: pewne informacje trzeba zapamiętać dla wszystkich tekseli tekstury irradiancji (także tych „nieużywanych"), a inne tylko dla tekseli „używanych", tj. związanych z elementami dyskretyzacji. Elementy są ponumerowane (od 0 do $n-1$); dla każdego z nich trzeba zapamiętać współrzędne jego teksela na teksturze irradiancji i numer makroelementu, do którego element należy, a w osobnych tablicach punkt kolokacji, wektor normalny i wektor wartości funkcji ρ opisujących odbijanie światła czerwonego, zielonego i niebieskiego. Jeszcze dwie tablice są potrzebne do przechowania dla wszystkich elementów wektorów o składowych r, g, b światła emitowanego oraz wektorów o składowych r, g, b wartości obliczonego rozwiązania równania bilansu energetycznego.

Dla każdego teksela tekstury irradiancji, o wymiarach $w \times h$ znalezionych w poprzednim etapie preprocesingu, trzeba zapamiętać dwie liczby: numer elementu (indeks do tablic z informacjami opisanymi wyżej) wraz z numerem trójkąta, któremu teksel odpowiada, albo liczbę oznaczającą, że teksel jest nieużywany. W zasadzie można by użyć do tego tekstury z tekselami o składowych typu uint, ale w implementacji użyłem bufora magazynowego z jednowymiarową tablicą elementów typu uvec2[16]. W kodzie w C bufor ten opatrzyłem nazwą BUF_VMAP.

[16] Dziedziną tekstury dostępnej przez ewaluator typu usampler2DRect jest prostokąt $[0, w] \times [0, h]$, gdzie w i h to jej wymiary w tekselach, co ułatwia dostęp do konkretnych tekseli. Zdecydowałem się użyć bufora magazynowego zamiast takiej tekstury po odkryciu, że próby przesłania z pamięci GPU do CPU tekseli o składowych 32-bitowych (typu uint) powodują błąd OpenGL-a. Jeśli to nie jest błąd implementacji standardu, to nie umiem tego wyjaśnić. *Zresztą żeglarz nie dyskutuje ze skałami, tylko je omija.* C.N. Parkinson.

Indeksy do większości tablic w buforach magazynowych używanych przez opisane niżej procedury i szadery są obliczane jednym z dwóch sposobów: indeks jest albo numerem elementu dyskretyzacji (czyli numerem niewiadomej od 0 do $n-1$ w układzie równań (29.3)), albo numerem teksela obliczanym ze wzoru $i = yw + x$, gdzie $x \in \{0, \ldots, w-1\}$ i $y \in \{0, \ldots, h-1\}$ to współrzędne teksela. W podanym niżej opisie będą używane określenia **tablica indeksowana numerem elementu** oraz **tablica indeksowana współrzędnymi teksela**. Zatem bufor BUF_VMAP zawiera tablicę indeksowaną współrzędnymi teksela.

Listing 29.6 przedstawia procedurę wywoływaną po utworzeniu tablic z opisami wszystkich wierzchołków i trójkątów sceny i w szczególności po nadaniu wierzchołkom położeń w dziedzinie tekstury irradiancji. Bezpośrednio przed jej wywołaniem należy też, za pomocą procedury z listingu 29.4, utworzyć obiekt tablicy wierzchołków.

Listing 29.6. Procedura TexturePatchVertices

C

```
static void TexturePatchVertices ( BalanceElements *belem )
{
  GLuint bfbo, mbuf;
  GLint  *buf;
  int    i, w, h, ntr, nelem, nmacroelem;

  w = belem->irrtxt_width;  h = belem->irrtxt_height;  ntr = belem->ntr;
  BUF_VARBUF = NewStorageBuffer ( w*h*4*sizeof(GLuint), 0 );
  BUF_VMAP = NewStorageBuffer ( w*h*2*sizeof(GLuint), 9 );
  glGenFramebuffers ( 1, &bfbo );
  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, bfbo );
  glFramebufferParameteri ( GL_DRAW_FRAMEBUFFER,
                            GL_FRAMEBUFFER_DEFAULT_WIDTH, w );
  glFramebufferParameteri ( GL_DRAW_FRAMEBUFFER,
                            GL_FRAMEBUFFER_DEFAULT_HEIGHT, h );
  if ( glCheckFramebufferStatus ( GL_DRAW_FRAMEBUFFER ) !=
         GL_FRAMEBUFFER_COMPLETE )
    ExitOnError ( "TexturePatchVertices 0" );
  CTLUniformi ( belem, CTL_WIDTH, belem->irrtxt_width );
  CTLUniformi ( belem, CTL_HEIGHT, belem->irrtxt_height );
  glViewport ( 0, 0, w, h );
  glUseProgram ( bprog_id[1] );  /* listing 29.7 */
  CTLUniformi ( belem, CTL_STAGE, 0 );
  COMPUTE ( w*h, 1, 1 );
  BUF_MATBUF = NewStorageBuffer ( ntr*sizeof(GLuint), 11 );
  if ( !(buf = malloc ( ntr*sizeof(GLint) )) )
    ExitOnError ( "TexturePatchVertices 1" );
  for ( i = 0; i < ntr; i++ )
    buf[i] = belem->trdesc[i].matnum;
  glBufferSubData ( GL_SHADER_STORAGE_BUFFER, 0, ntr*sizeof(GLint), buf );
  free ( buf );
  BUF_ABUF = NewStorageBuffer ( w*h*4*sizeof(GLfloat), 6 );
  glUseProgram ( bprog_id[0] );  /* listingi 29.8, 29.9 i 29.10 */
```

```
34:   for ( i = 0; i < belem->nobj; i++ ) {
35:     bobj = &belem->objtab[i];
36:     CTLUniformi ( belem, CTL_FIRST, bobj->ftrdesc );
37:     CTLUniformi ( belem, CTL_TXTS, bobj->txts );
38:     glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
39:     glDrawElements ( GL_TRIANGLES, 3*bobj->cntr, GL_UNSIGNED_INT,
40:                      (GLvoid*)(bobj->ftrdesc*3*sizeof(GLuint)) );
41:     glFlush ();
42:     glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
43:     glDrawElements ( GL_TRIANGLES, 3*bobj->cntr, GL_UNSIGNED_INT,
44:                      (GLvoid*)(bobj->ftrdesc*3*sizeof(GLuint)) );
45:     glFlush ();
46:   }
47:   BUF_OBJID = NewStorageBuffer ( w*h*sizeof(GLuint), 8 );
48:   LoadPatchRectSizes ( belem );
49:   glUseProgram ( bprog_id[2] );  /* listingi 29.12 i 29.13 */
50:   glBindVertexArray ( empty_vao );
51:   glDrawArraysInstanced ( GL_TRIANGLE_STRIP, 0, 4, belem->ntrpatch );
52:   glFlush ();
53:   glBindVertexArray ( 0 );
54:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
55:   glDeleteFramebuffers ( 1, &bfbo );
56:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, BUF_VERTPOS );
57:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 2, BUF_TXB );
58:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 3, BUF_ELIND );
59:   BUF_TBUF = NewStorageBuffer ( w*h*4*sizeof(GLfloat), 4 );
60:   glUseProgram ( bprog_id[1] );  /* listing 29.7 */
61:   CTLUniformi ( belem, CTL_STAGE, 1 );
62:   COMPUTE ( w, h, 1 );
63:   belem->nelem = nelem = FindNElem ( belem, w*h );
64:   if ( !(belem->VarBuf = malloc ( (nelem*5+w*h*2)*sizeof(GLuint) )) )
65:     ExitOnError ( "TexturePatchVertices 2" );
66:   belem->ObjIdBuf = &belem->VarBuf[nelem*4];
67:   belem->VarMap = &belem->ObjIdBuf[nelem];
68:   NetSort ( belem, w*h );
69:   CTLUniformi ( belem, CTL_STAGE, 4 );
70:   COMPUTE ( nelem, 1, 1 );
71:   PrefixSum ( belem, nelem );
72:   glGetNamedBufferSubData ( BUF_VARBUF, 0, nelem*4*sizeof(GLuint),
73:                             belem->VarBuf );
74:   glGetNamedBufferSubData ( BUF_VARBUF, (nelem-1)*4*sizeof(GLuint),
75:                             sizeof(GLuint), &nmacroelem );
76:   belem->nmacroelem = ++nmacroelem;
77:   mbuf = NewStorageBuffer ( nelem*sizeof(GLuint), 7 );
78:   CTLUniformi ( belem, CTL_STAGE, 6 );
79:   COMPUTE ( nelem, 1, 1 );
80:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 8, 0 );
81:   glDeleteBuffers ( 1, &BUF_OBJID );
```

```
82:   BUF_OBJID = mbuf;
83:   glGetNamedBufferSubData ( BUF_VARBUF, 0, nelem*4*sizeof(GLuint),
84:                             belem->VarBuf );
85:   glGetNamedBufferSubData ( BUF_OBJID, 0, nelem*sizeof(GLuint),
86:                             belem->ObjIdBuf );
87:   ExitIfGLError ( "TexturePatchVertices" );
88: } /*TexturePatchVertices*/
```

W liniach 8 i 9 są tworzone bufory magazynowe; bufor BUF_VARBUF zawiera tablicę zmiennych typu uvec4, której elementy początkowo odpowiadają tekselom tekstury irradiancji (stąd podana długość tablicy, równa wh), a po zakończeniu preprocesingu będą odpowiadać elementom dyskretyzacji.

Pozaekranowy bufor ramki tworzony w liniach 10–18 nie ma załączników. Aby można go było użyć, trzeba podać jego wymiary w pikselach[17], co robi procedura wywołana w liniach 12–15. W liniach 19–20 zmiennym jednolitym width i height w bloku CtlBlock są przypisywane wymiary tekstury irradiancji. Program szaderów wywołany w linii 24 nadaje wartości początkowe elementom tablic (indeksowanych współrzędnymi teksela) w buforach BUF_VARBUF i BUF_VARMAP (listing 29.7, linie 118–119); pod nazwą RESTART_IND jest ukryta liczba $2^{32} - 1$; taka wartość zmiennej *nie jest* numerem elementu, makroelementu, płata ani trójkąta.

W linii 25 jest rezerwowany bufor magazynowy z tablicą, do której w liniach 26–30 są przesyłane numery materiałów wszystkich trójkątów. W buforze utworzonym przez instrukcję w linii 32 znajdą się uśrednione wartości (składowych r, g, b) funkcji ρ opisującej odbijanie światła. Tablica w tym buforze jest indeksowana współrzędnymi teksela.

Listing 29.7. Szader obliczeniowy pierwszego programu preprocesingu

GLSL

```
1: #version 450 core
2:
3: #define RESTART_IND 0xFFFFFFFF
4: #define EPS0 0.001
5: #define EPS1 0.01
6: #define MG   0.1
7:
8: layout(local_size_x=1) in;
9:
10: layout(std430,binding=0) buffer VarBuf { uvec4 tvar[]; } vbuf;
11: layout(std430,binding=1) buffer TSPointBuf { float pt[]; } tsbuf;
12: layout(std430,binding=2) buffer TXPointBuf { vec2 tt[]; } txbuf;
13: layout(std430,binding=3) buffer TIndBuf { int ind[]; } tind;
14: layout(std430,binding=4) buffer CPointBuf { vec4 cp[]; } cpbuf;
15: layout(std430,binding=7) buffer VOIdBuf { uint id[]; } vobjid;
16: layout(std430,binding=8) buffer ObjIdBuf { uint id[]; } objid;
17: layout(std430,binding=9) buffer VMap { uvec2 px[]; } vmap;
18:
```

[17] Jeśli bufor ramki ma załączniki, to te wymiary są brane z nich.

```
19: uniform CtlBlock { .... } ctl;  /* listing 29.5 */
20:
21: vec2 trv[13];
22:
23: int SHClip ( vec2 cpp, vec2 cpn, int n, int k, int l ) { .... }
24:
25: float PolygonArea ( int n )
26: {
27:   float a;
28:   int   i;
29:
30:   a = (trv[0].x-trv[n-1].x)*(trv[0].y+trv[n-1].y);
31:   for ( i = 1; i < n; i++ )
32:     a += (trv[i].x-trv[i-1].x)*(trv[i].y+trv[i-1].y);
33:   return abs ( 0.5*a );
34: } /*PolygonArea*/
35:
36: vec2 CentralPoint ( int n )
37: {
38:   vec2 c;
39:   int  i;
40:
41:   for ( c = trv[0], i = 1;  i < n;  i++ )
42:     c += trv[i];
43:   return c / float(n);
44: } /*CentralPoint*/
45:
46: bool ProcessTexel ( int x, int y )
47: {
48:   int   z, trID, ind[3], i, n, i0, i1, i2;
49:   float a;
50:   mat3  m;
51:   vec2  c, d;
52:   vec3  v0, v1, v2, bc;
53:
54:   z = y*ctl.width + x;
55:   cpbuf.cp[z] = vec4(0.0);
56:   if ( (trID = int(vbuf.tvar[z].y)) == RESTART_IND )
57:     return false;
58:   for ( i = 0; i < 3; i++ ) {
59:     ind[i] = tind.ind[3*trID+i];
60:     trv[i] = txbuf.tt[ind[i]];
61:   }
62:   if ( (n = SHClip ( vec2(float(x),0.0), vec2(1.0,0.0), 3, 0, 7 )) < 3 )
63:     return false;
64:   if ( (n = SHClip ( vec2(0.0,float(y)), vec2(0.0,1.0), n, 7, 0 )) < 3 )
65:     return false;
```

```
66:   if ( (n = SHClip ( vec2(float(x+1),0.0), vec2(-1.0,0.0), n, 0, 7 )) < 3 )
67:     return false;
68:   if ( (n = SHClip ( vec2(0.0,float(y+1)), vec2(0.0,-1.0), n, 7, 0 )) < 3 )
69:     return false;
70:   a = PolygonArea ( n );
71:   if ( a < EPS0 ) return false;
72:   c = CentralPoint ( n );  d = c - vec2 ( float(x), float(y) );
73:   if ( a < EPS1 && (d.x < MG || d.x > 1.0-MG || d.y < MG || d.y > 1.0-MG) )
74:     return false;
75:   cpbuf.cp[z].w = a;
76:   i0 = ind[0];  i1 = ind[1];  i2 = ind[2];
77:   m = mat3 ( vec3(txbuf.tt[i0],1.0),
78:              vec3(txbuf.tt[i1],1.0),
79:              vec3(txbuf.tt[i2],1.0) );
80:   bc = inverse ( m ) * vec3 ( c, 1.0 );
81:   v0 = vec3 ( tsbuf.pt[3*i0], tsbuf.pt[3*i0+1], tsbuf.pt[3*i0+2] );
82:   v1 = vec3 ( tsbuf.pt[3*i1], tsbuf.pt[3*i1+1], tsbuf.pt[3*i1+2] );
83:   v2 = vec3 ( tsbuf.pt[3*i2], tsbuf.pt[3*i2+1], tsbuf.pt[3*i2 +2] );
84:   cpbuf.cp[z].xyz = bc[0]*v0 + bc[1]*v1 + bc[2]*v2;
85:   return true;
86: } /*ProcessTexel*/
87:
88: void CompSwap ( uint x )
89: {
90:   uint  i, j, l, h2;
91:   uvec4 s;
92:
93:   h2 = ctl.H >> 1;  l = x % h2;  x /= h2;  i = x*ctl.H+l;
94:   if ( (j = ctl.reverse ? (x+1)*ctl.H-l-1 : i+h2) < ctl.N ) {
95:     if ( vbuf.tvar[i].z > vbuf.tvar[j].z )
96:       { s = vbuf.tvar[i];  vbuf.tvar[i] = vbuf.tvar[j];  vbuf.tvar[j] = s; }
97:   }
98: } /*CompSwap*/
99:
100: void PrefixSum ( int i )
101: {
102:   uint ii, m0, m1, ia, ib;
103:
104:   ii = i+i;  m0 = 0x01 << ctl.step;  m1 = m0-1;
105:   ia = (ii &  m0) | m1;
106:   if ( (ib = ia + (i & m1) + 1) < ctl.N )
107:     vbuf.tvar[ib].x += vbuf.tvar[ia].x;
108: } /*PrefixSum*/
109:
110: void main ( void )
111: {
112:   int   x, y, z;
```

```
113:   uvec4 tv;
114:
115:   z = int(gl_GlobalInvocationID.x);
116:   switch ( ctl.stage ) {
117: case 0:
118:     vbuf.tvar[z] = uvec4(0,RESTART_IND,RESTART_IND,0);
119:     vmap.px[z] = uvec2(RESTART_IND,RESTART_IND);
120:     return;
121: case 1:
122:     y = int(gl_GlobalInvocationID.y);
123:     if ( !ProcessTexel ( z, y ) )
124:       vbuf.tvar[y*ctl.width + z] =
125:           uvec4(0,RESTART_IND,RESTART_IND,0);
126:     return;
127: case 2:
128:     y = z + ctl.N/2;
129:     if ( (ctl.N & 0x00000001) != 0 ) y++;
130:     vbuf.tvar[z].x += vbuf.tvar[y].x;
131:     return;
132: case 3:
133:     CompSwap ( gl_GlobalInvocationID.x );
134:     return;
135: case 4:
136:     if ( z == 0 )
137:       vbuf.tvar[z].x = 0;
138:     else
139:       vbuf.tvar[z].x = vbuf.tvar[z].z > vbuf.tvar[z-1].z ? 1 : 0;
140:     return;
141: case 5:
142:     PrefixSum ( z );
143:     return;
144: case 6:
145:     x = int(vbuf.tvar[z].w & 0xFFFF);  y = int(vbuf.tvar[z].w >> 16);
146:     vobjid.id[z] = objid.id[y*ctl.width + x];
147:     return;
148:   }
149: } /*main*/
```

W pętli w liniach 34–46 procedura `TexturePatchVertices` rysuje trójkąty kolejnych obiektów w dziedzinie tekstury irradiancji za pomocą programu wybranego w linii 33. Przed rysowaniem obiektu zmiennym `first` i `txts` są przypisywane numer pierwszego trójkąta obiektu i wartość zmiennej `txts` z opisu trójkąta. Rysowanie odbywa się dwukrotnie: za pierwszym razem są rysowane krawędzie trójkątów, a za drugim są wypełniane ich wnętrza; dodatkowe rysowanie krawędzi ma na celu pokrycie obszaru każdego płata tekselami[18]. Wywołana w liniach 41 i 45 procedura `glFlush` zapobiega „wymieszaniu" odcinków i trójkątów; bez tego OpenGL może zmieniać kolejność rysowania, czego skutkiem może być

[18] Do pełnego pokrycia *może* brakować nielicznych tekseli, z czym jednak można się pogodzić.

błędne uznanie wielu tekseli za nieużywane. Marginesy o szerokości jednego teksela wokół płatów zapewniają, że teksele żadnego płata nie „zabrudzą" obrazu płatów sąsiednich. Szader wierzchołków przypisuje zmiennej `gl_Position` wektor współrzędnych wierzchołka w dziedzinie tekstury, przekształconych do układu kostki standardowej. Położenie wierzchołka w przestrzeni jest ignorowane.

Listing 29.8. Szader wierzchołków drugiego programu preprocesingu

GLSL

```
#version 450 core

layout(location=1) in vec2 TxtCoord;
layout(location=2) in vec2 in_Position;
out vec2 TexCoord;

uniform CtlBlock { .... } ctl;  /* listing 29.5 */

void main ( void )
{
  TexCoord = TxtCoord;
  gl_Position = vec4 ( 2.0*in_Position.x/ctl.width-1.0,
                       2.0*in_Position.y/ctl.height-1.0, 0.0, 1.0 );
} /*main*/
```

Zależnie od wartości zmiennej jednolitej `ctl.txts` szader geometrii przekazuje na wyjście współrzędne tekstury wierzchołka otrzymane na wejściu albo współrzędne wzięte z tablicy `txtc` — w scenie użytej do uruchomienia implementacji obiektem pokrytym teksturą, dla której współrzędne są generowane w ten sposób, jest lampa zawieszona pod sufitem. Do atrybutów każedgo wierzchołka szader dołącza też numer trójkąta otrzymany w zmiennej `gl_PrimitiveIDIn`.

Listing 29.9. Szader geometrii drugiego programu preprocesingu

GLSL

```
#version 450 core

layout(triangles) in;
layout(triangle_strip,max_vertices=3) out;

in  vec2 TexCoord[];
out vec2 TxtCoord;

uniform CtlBlock { .... } ctl;  /* listing 29.5 */

const vec2 txtc[3] = { vec2(1.0,0.0), vec2(0.0,1.0), vec2(0.0,0.0) };

void main ( void )
{
  int i;
```

```
16:
17:   for ( i = 0; i < 3; i++ ) {
18:     if ( ctl.txts == 0 )
19:       TxtCoord = TexCoord[i];
20:     else
21:       TxtCoord = txtc[i];
22:     gl_PrimitiveID = gl_PrimitiveIDIn;
23:     gl_Position = gl_in[i].gl_Position;
24:     EmitVertex ();
25:   }
26:   EndPrimitive ();
27: } /*main*/
```

Szader fragmentów w linii 25 na podstawie współrzędnych fragmentu zaokrąglonych do liczb całkowitych oblicza indeks do tablic, w których zapisuje wyniki: elementowi tablicy `vbuf.tvar` (w buforze `BUF_VARBUF`) przypisuje wektor złożony z liczby 1, numeru trójkąta i dwóch zer, a elementowi tablicy `abuf.a` (w buforze `BUF_ABUF`) przypisuje wektor zdolności powierzchni do odbijania światła wzięty z opisu materiału lub, jeśli na obiekt jest nałożona tekstura, z tekstury. Filtrowanie tekstury przez ewaluator zapewnia jej uśrednienie w obszarze teksela.

Listing 29.10. Szader fragmentów drugiego programu preprocesingu

GLSL

```
1: #version 450 core
2:
3: #define MAX_TEXTURES   4
4: #define MAX_MATERIALS 20
5:
6: in vec2 TxtCoord;
7:
8: layout(std430,binding=0) buffer VarBuf { uvec4 tvar[]; } vbuf;
9: layout(std430,binding=6) buffer AlBuf { vec4 a[]; } abuf;
10: layout(std430,binding=11) buffer MatBuf { int matnum[]; } mbuf;
11:
12: layout(binding=0) uniform sampler2D tex[MAX_TEXTURES];
13:
14: uniform CtlBlock { .... } ctl;  /* listing 29.5 */
15:
16: struct Material { .... };  /* listing 18.1 */
17:
18: uniform MatBlock { int mtn;  Material mat[MAX_MATERIALS];  } mat;
19:
20: void main ( void )
21: {
22:   int x, y, z, n;
23:
```

```
24:   x = int(gl_FragCoord.x);  y = int(gl_FragCoord.y);
25:   z = y*ctl.width + x;
26:   vbuf.tvar[z] = uvec4 ( 1, gl_PrimitiveID, 0, 0 );
27:   n = mbuf.matnum[gl_PrimitiveID+ctl.first];
28:   if ( mat.mat[n].txtnum >= 0 )
29:     abuf.a[z] = texture ( tex[mat.mat[n].txtnum], TxtCoord );
30:   else
31:     abuf.a[z] = mat.mat[n].diffref;
32:   discard;
33: } /*main*/
```

Bufor BUF_OBJID rezerwowany w linii 47 zawiera tablicę, w której dla każdego teksela tekstury irradiancji będzie zapamiętany numer obiektu, do którego należy trójkąt, do którego obrazu należy ten teksel. Szadery „widzą" ten bufor jako blok magazynowy ObjIdBuf. Zadaniem wywołanej w następnej linii procedury LoadPatchRectSizes (listing 29.11) jest określenie wielkości makroelementów dla każdego płata i przesłanie ich do bufora BUF_PRS, dostępnego jako blok Pr szaderom z listingów 29.12 i 29.13.

Pola struktury typu PatchRect są identyczne jak pola zadeklarowanej w liniach 7–11 na listingu 29.12 struktury o tej samej nazwie, a ponieważ wszystkie te pola są zmiennymi 32-bitowymi, w bloku o układzie std430 są również upakowane bez przerw i można całą zawartość bufora przesłać w jednym wywołaniu procedury glBufferSubData w linii 38.

Listing 29.11. Procedura LoadPatchRectSizes

C

```
1: #define MACROEL_DSIZE 5
2:
3: static int FindMacroElSize ( int ps )
4: {
5:   int d0, d1, r;
6:
7:   if ( ps > MACROEL_DSIZE )
8:     for ( d0 = MACROEL_DSIZE-1, d1 = MACROEL_DSIZE; ; d0--, d1++ ) {
9:       r = ps % d1;
10:      if ( r == 0 || 4*r > 3*d1 )
11:        return d1;
12:      r = ps % d0;
13:      if ( r == 0 || 4*r > 3*d0 )
14:        return d0;
15:    }
16:  return ps;
17: } /*FindMacroElSize*/
18:
19: void LoadPatchRectSizes ( BalanceElements *belem )
20: {
21:   TriangPatch *ts;
22:   PatchRect   *pr;
```

```
23:   int          i, bsize, step;
24:
25:   step = sizeof(PatchRect);
26:   bsize = belem->ntrpatch*step;
27:   if ( !(pr = malloc ( bsize )) )
28:     ExitOnError ( "LoadPatchRectSizes" );
29:   for ( i = 0; i < belem->ntrpatch; i++ ) {
30:     ts = &belem->trpatch[i];
31:     pr[i].w = ts->w;  pr[i].h = ts->h;
32:     pr[i].x0 = ts->x0;  pr[i].y0 = ts->y0;
33:     FindMacroElSize ( (int)ts->w-2, &pr[i].melw );
34:     FindMacroElSize ( (int)ts->h-2, &pr[i].melh );
35:     pr[i].objID = ts->objid;
36:   }
37:   BUF_PRS = NewStorageBuffer ( bsize, 3 );
38:   glBufferSubData ( GL_SHADER_STORAGE_BUFFER, 0, bsize, pr );
39:   free ( pr );
40:   ExitIfGLError ( "LoadPatchRectSizes" );
41: } /*LoadPatchRectSizes*/
```

Prostokąt, w który otoczka wypukła obrazu trójkątów płata w dziedzinie tekstury jest wpisana z marginesem o szerokości 1, jeśli jest duży, zostaje podzielony na mniejsze makroelementy; pomocnicza procedura `FindMacroElSize`, osobno dla szerokości i wysokości prostokąta, dobiera szerokość i wysokość płata, przy czym wielkość domyślna 5 może być zwiększona lub zmniejszona, jeśli reszta z dzielenia wymiaru płata przez wymiar makroelementu jest za mała (celem jest podzielenie każdego płata na makroelementy o zbliżonych rozmiarach). Małe płaty zostaną pokryte przez jeden makroelement.

Instrukcja procedury `TexturePatchVertices` w linii 51 rysuje prostokąty otaczające płaty przy użyciu programu zbudowanego z szaderów przedstawionych na listingach 29.12 i 29.13. Szader wierzchołków oblicza współrzędne wierzchołka prostokąta na podstawie numeru instancji i numeru wierzchołka, po czym w liniach 28–29 przekształca je do układu kostki standardowej. Szader fragmentów w linii 18 oblicza indeks do jednowymiarowych tablic w buforach `BUF_VARBUF` i `BUF_OBJID`. Jeśli składowa `x` wektora odpowiadającego tekselowi tekstury irradiancji ma wartość 1, to ten teksel należy do obrazu trójkąta (tj. był „zamalowany” przez wcześniej użyty program szaderów). Wtedy składowej `z` zostaje przypisana liczba przyjęta jako tymczasowy identyfikator makroelementu. Liczba ta jest utworzona z przesuniętego o 16 pozycji numeru płata i numeru prostokąta tworzącego w obszarze płata wzór szachownicy. W 16-bitowych połówkach składowej `w` zostają zapamiętane współrzędne teksela.

Dalsze obliczenia są prowadzone przy użyciu szadera obliczeniowego z listingu 29.7. W linii 59 procedura `TexturePatchVertices` tworzy bufor magazynowy przywiązany do punktu 4, dostępny jako blok `CPointBuf`. Służy on do tymczasowego przechowania (w tablicy indeksowanej współrzędnymi teksela) położeń punktów kolokacji, które później będą przepisane do tablicy (indeksowanej numerem elementu) w buforze `BUF_CP`. Instrukcja w linii 62 powoduje wykonanie etapu 1 obliczeń. Szader wywołuje wtedy procedurę

Listing 29.12. Szader wierzchołków trzeciego programu preprocesingu

GLSL

```
#version 450 core

flat out int patchID;

struct PatchRect {
    float w, h, x0, y0;
    int   melw, melh;
    int   objID;
  };

layout(std430,binding=3) buffer Pr { PatchRect pr[]; } pr;

uniform CtlBlock { .... } ctl;  /* listing 29.5 */

void main ( void )
{
  vec2 vertex;
  int  id;

  patchID = id = gl_InstanceID;
  switch ( gl_VertexID ) {
 case 0: vertex = vec2 ( pr.pr[id].x0, pr.pr[id].y0 );              break;
 case 1: vertex = vec2 ( pr.pr[id].x0+pr.pr[id].w, pr.pr[id].y0 );  break;
 case 2: vertex = vec2 ( pr.pr[id].x0, pr.pr[id].y0+pr.pr[id].h );  break;
default: vertex =
     vec2 ( pr.pr[id].x0+pr.pr[id].w, pr.pr[id].y0+pr.pr[id].h );   break;
  }
  gl_Position = vec4 ( 2.0*vertex.x/ctl.width-1.0,
                       2.0*vertex.y/ctl.height-1.0, 0.0, 1.0 );
} /*main*/
```

Listing 29.13. Szader fragmentów trzeciego programu preprocesingu

GLSL

```
#version 450 core

flat in int patchID;

struct PatchRect { .... };  /* listing 29.12 */

layout(std430,binding=0) buffer VarBuf { uvec4 tvar[]; } vbuf;
layout(std430,binding=8) buffer ObjIdBuf { int id[]; } objid;
layout(std430,binding=3) buffer Pr { PatchRect pr[]; } pr;

uniform CtlBlock { .... } ctl;  /* listing 29.5 */
```

```
12:
13: void main ( void )
14: {
15:   int x, y, z, a, b, c;
16:
17:   x = int(gl_FragCoord.x);  y = int(gl_FragCoord.y);
18:   z = y*ctl.width + x;
19:   if ( vbuf.tvar[z].x == 1 ) {
20:     a = (x-int(pr.pr[patchID].x0)-1) / pr.pr[patchID].melw;
21:     b = (int(pr.pr[patchID].w)-2) / pr.pr[patchID].melw;
22:     c = (y-int(pr.pr[patchID].y0)-1) / pr.pr[patchID].melh;
23:     vbuf.tvar[z].z = (patchID << 16) + ((a + b*c) << 1) + ((a^c) & 0x01 );
24:     vbuf.tvar[z].w = (y << 16) + x;
25:     objid.id[z] = pr.pr[patchID].objID;
26:   }
27:   discard;
28: } /*main*/
```

`ProcessTexel` (listing 29.7, linie 46–86). Jeśli teksel nie należy do obrazu żadnego trójkąta, to w linii 57 następuje powrót. Jeśli należy, to znajdowane jest przecięcie teksela — kwadratu o boku 1 z trójkątem; jeśli przecięcie jest puste, to teksel, mimo „zamalowania" podczas rysowania trójkątów, będzie nieużywany.

Pętla w liniach 58–61 przepisuje do tablic roboczych `ind` i `trv` indeksy i współrzędne wierzchołków trójkąta w dziedzinie tekstury irradiancji. W liniach 62–69 następują cztery wywołania procedury `SHClip`, która realizuje algorytm Sutherlanda-Hodgmana obcinania wielokątów. Kod tej procedury, podobny do procedur na listingach 19.14 i F.14, pominąłem.

Procedura `PolygonArea` wywołana w linii 70 oblicza pole znalezionej części wspólnej teksela i trójkąta. Jeśli jest ono za małe, to teksel również zostaje uznany za nieużywany. W przeciwnym razie w linii 72 jest wywołana procedura obliczająca środek ciężkości zbioru wierzchołków części wspólnej; punkt ten posłuży do znalezienia (na trójkącie w przestrzeni) punktu kolokacji elementu dyskretyzacji, ale jeśli jest on położony zbyt blisko brzegu teksela (w odległości mniejszej niż `MG`, tj. 0.1), to teksel też będzie nieużywany.

Pole części wspólnej z trójkątem teksela, który ma być używany, w linii 75 jest zapamiętywane. W liniach 77–80 jest znajdowany (i przypisywany zmiennej `bc`) wektor współrzędnych barycentrycznych (w układzie określonym przez wierzchołki trójkąta) punktu kolokacji na podstawie współrzędnych kartezjańskich w dziedzinie tekstury irradiancji. Współrzędne te są następnie użyte do znalezienia kombinacji afinicznej wierzchołków trójkąta w przestrzeni, która zostaje zapamiętana (w linii 84) jako punkt kolokacji elementu dyskretyzacji.

Wywołana przez procedurę `TexturePatchVertices` w linii 63 funkcja `FindNElem` oblicza sumę składowych `x` wektorów w tablicy indeksowanej numerem elementu w buforze `BUF_VARBUF`; teksele używane mają tam zapamiętaną liczbę 1, a nieużywane 0. Algorytm sumowania jest opisany w podrozdziale G.1 (obliczenia na GPU są realizowane przez instrukcje w liniach 128–130 szadera na listingu 29.7). Obliczona suma jest liczbą n elementów dyskretyzacji.

Procedura NetSort wywołana w linii 68 sortuje tablicę wektorów w buforze BUF_VARBUF w kolejności rosnących identyfikatorów makroelementów. Algorytm sortowania, opisany w podrozdziale G.3, jest realizowany przez GPU, przy czym komparatorem porównującym i przestawiającym elementy tablicy jest instrukcja w liniach 95–96 (listing 29.7). W wyniku sortowania elementy należące do każdego makroelementu znajdą się w tablicy obok siebie. Wektory odpowiadające tekselom nieużywanym mają przypisany identyfikator makroelementu 0xFFFFFFFF, tj. $2^{32}-1$. W posortowanej tablicy te wektory znajdą się na końcu i w dalszych obliczeniach zawierająca je część tablicy będzie nieużywana, a od tej chwili tablica w buforze BUF_VARBUF jest indeksowana numerem elementu. Sortowanie ostatecznie ustala kolejność niewiadomych w układzie równań (29.3), tj. nadaje im numery od 0 do $n-1$.

Kolejny etap obliczeń, wykonywany po wykonaniu instrukcji w linii 70 przez instrukcje szadera w liniach 136–139, polega na przypisaniu składowej x każdego wektora w buforze BUF_VARBUF liczby 0 albo 1; jedynka jest przypisywana wtedy, gdy tymczasowy identyfikator makroelementu, przechowywany w składowej z, jest większy niż identyfikator makroelementu w poprzednim wektorze. Procedura wywołana w linii 71 powoduje obliczenie ciągu sum prefiksowych tego ciągu zer i jedynek (zobacz podrozdz. G.2). W ten sposób makroelementy otrzymują swoje numery, od 0 do $m-1$. Liczba $m-1$ jest odczytywana z bufora w liniach 74–75, po czym w linii 76 następuje obliczenie i zapamiętanie liczby m.

Etap 6, realizowany przez instrukcje szadera w liniach 145–146 po wykonaniu instrukcji w linii 79, dla każdego elementu dyskretyzacji przepisuje numer obiektu, przez którego trójkąt teksel został „zamalowany", z tablicy indeksowanej współrzędnymi teksela do tablicy indeksowanej numerem elementu. Bufor BUF_OBJID, zawierający tę pierwszą tablicę, jest w linii 81 zwalniany, a w linii 82 zastępowany przez bufor z nową tablicą.

Instrukcje w liniach 83–86 odczytują dane z buforów do tablic w pamięci CPU, co umożliwia ich oglądanie; bardzo mi to pomogło w uruchomieniu implementacji i w wykonaniu ilustracji do tego rozdziału.

Pokazana na listingu 29.14 procedura ComputeMatricesDandA realizuje kolejny etap preprocesingu, którego wynikiem są macierze D i A opisane w p. 29.2.1. Macierze te są konstruowane przy użyciu szaderów z listingów 29.15 i 29.16. Bufor BUF_ALBMAT, rezerwowany przez instrukcję w linii 6, zawiera tablicę wektorów o składowych r, g, b, opisujących odbijanie światła przez elementy — tablica ta reprezentuje diagonalę macierzy D. Z powodów opisanych w p. 29.2.1 elementy tej tablicy są typu vec4.

Listing 29.14. Procedura konstrukcji macierzy D i A

C

```
1: static void ComputeMatricesDandA ( BalanceElements *belem )
2: {
3:   int nelem, nmacroelem;
4:
5:   nelem = belem->nelem;  nmacroelem = belem->nmacroelem;
6:   BUF_ALBMAT = NewStorageBuffer ( nelem*4*sizeof(GLfloat), 7 );
7:   BUF_CP = NewStorageBuffer ( nelem*4*sizeof(GLfloat), 5 );
8:   glUseProgram ( bprog_id[5] );    /* listing 29.15 */
```

```
 9:   COMPUTE ( nelem, 1, 1 );
10:   belem->avgmat.m = nmacroelem;
11:   belem->avgmat.n = belem->avgmat.nnz = nelem;
12:   belem->avgmat.lmax = 0;
13:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 4, BUF_CP );
14:   belem->avgmat.buf[0] =
15:       NewStorageBuffer ( (nelem+nmacroelem+1)*sizeof(GLuint), 5 );
16:   belem->avgmat.buf[1] = NewStorageBuffer ( nelem*sizeof(GLfloat), 6 );
17:   glUseProgram ( bprog_id[3] );  /* listing 29.16 */
18:   CTLUniformi ( belem, CTL_NROWS, nmacroelem );
19:   CTLUniformi ( belem, CTL_NCOLS, nelem );
20:   CTLUniformi ( belem, CTL_STAGE, 0 );
21:   COMPUTE ( nelem, 1, 1 );
22:   CTLUniformi ( belem, CTL_STAGE, 1 );
23:   COMPUTE ( nmacroelem, 1, 1 );
24:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 8, BUF_OBJID );
25:   glUseProgram ( 0 );
26:   ExitIfGLError ( "ComputeMatricesDandA" );
27: } /*ComputeMatricesDandA*/
```

Bufor `BUF_CP` zawiera indeksowaną numerem elementu tablicę, w której mają być przechowywane punkty kolokacji. Punkty te zostały obliczone przez szader z listingu 29.7 i zapamiętane w tablicy indeksowanej współrzędnymi teksela w buforze `BUF_TBUF`, a teraz zostaną przepisane do nowego bufora.

Dygresja: To obliczenie *nie jest* związane z konstrukcją macierzy i powinno być wykonane w części preprocesingu realizowanej przez procedurę `TexturePatchVertices`, najlepiej przez szader z listingu 29.7. Ale wtedy szader ten musiałby mieć dostęp do dziewięciu buforów magazynowych, co przekracza minimalny limit gwarantowany przez specyfikację [1]. Zainstalowane na moim sprzęcie implementacje standardu dopuszczają większe limity, ale chciałbym, by opisane tu procedury działały także tam, gdzie producent sprzętu i sterowników nie był aż tak szczodry[19]. Ponieważ nie chciało mi się pisać osobnego szadera, dopisałem to zadanie szaderowi z listingu 29.15. Czytelników zachęcam do zrobienia tego lepiej.

Pokazany na listingu 29.15 szader, którego każdy wątek odpowiada jednemu elementowi dyskretyzacji, w linii 21 oblicza indeks do tablic indeksowanych współrzędnymi teksela, a potem w linii 22 zapamiętuje w buforze `BUF_VMAP` indeks elementu i numer trójkąta, do którego należy element, w linii 23 przepisuje wektor odbijania światła z tablicy indeksowanej współrzędnymi teksela do tablicy z diagonalą macierzy D i w linii 24 przepisuje współrzędne punktu kolokacji w elemencie do bufora `BUF_CP`.

Macierz A ma m wierszy i n kolumn, przy czym w każdej kolumnie ma tylko jeden niezerowy współczynnik, a dzięki posortowaniu elementów zgodnie z numerami makroele-

[19] Na takiej „szczodrości" można się sparzyć: po wymianie karty graficznej na nowszą i mocniejszą odkryłem, że maksymalna liczba wątków w lokalnej grupie roboczej szadera obliczeniowego zmalała z 1536 do określonego przez specyfikację minimum 1024.

mentów wszystkie niezerowe współczynniki w każdym wierszu tej macierzy występują obok siebie. Macierz ta jest reprezentowana w sposób opisany w podrozdziale G.4, za pomocą trzech tablic przechowywanych w dwóch buforach. Procedura `ComputeMatricesDandA` rezerwuje te bufory, wykonując instrukcje w liniach 14–16, wcześniej zapamiętawszy w polach opakowania macierzy wymiary m i n oraz liczbę n jej niezerowych współczynników.

Listing 29.15. Szader obliczeniowy konstrukcji macierzy D

GLSL

```
1: #version 450 core
2:
3: layout(local_size_x=1) in;
4:
5: layout(std430,binding=0) buffer VarBuf { uvec4 tvar[]; } vbuf;
6: layout(std430,binding=4) buffer CPointBuf { vec4 cp[]; } cpbuf;
7: layout(std430,binding=5) buffer VCPointBuf { vec4 cp[]; } vcpbuf;
8: layout(std430,binding=6) buffer AlBuf { vec4 a[]; } abuf;
9: layout(std430,binding=7) buffer AlbMat { vec4 a[]; } albm;
10: layout(std430,binding=9) buffer VMap { uvec2 px[]; } vmap;
11:
12: uniform CtlBlock { .... } ctl;  /* listing 29.5 */
13:
14: void main ( void )
15: {
16:   int x, y, z;
17:   uvec4 tv;
18:
19:   z = int(gl_GlobalInvocationID.x);
20:   tv = vbuf.tvar[z];  x = int(tv.w & 0xFFFF);  y = int(tv.w >> 16);
21:   x = y*ctl.width + x;
22:   vmap.px[x] = uvec2 ( z, tv.y );
23:   albm.a[z] = abuf.a[x];
24:   vcpbuf.cp[z] = cpbuf.cp[x];
25: } /*main*/
```

Etap 0 obliczeń wykonywanych przez szader z listingu 29.16 ma za zadanie wypełnić tablice `r` (w liniach 23–28) i `c` (w linii 29). Liczba przypisywana w linii 30 elementowi tablicy `a` jest polem przecięcia obrazu trójkąta z kwadratem jednostkowym zajmowanym przez teksel. Współczynnik a_{ij} macierzy A ma być proporcjonalny do tego pola, a suma współczynników w każdym wierszu ma być równa 1. Aby to osiągnąć, w etapie 1 szader dla każdego wiersza (czyli makroelementu) oblicza (w liniach 33–35) sumę pól w wierszu, a w następnej pętli dzieli pola przez tę sumę. Jest tu użyty sekwencyjny algorytm sumowania; nie warto go zastępować algorytmem równoległym, bo macierz ma dość dużo wierszy (przetwarzanych równolegle, więc GPU i tak ma zajęcie), a maksymalna liczba składników jest rzędu kilkudziesięciu.

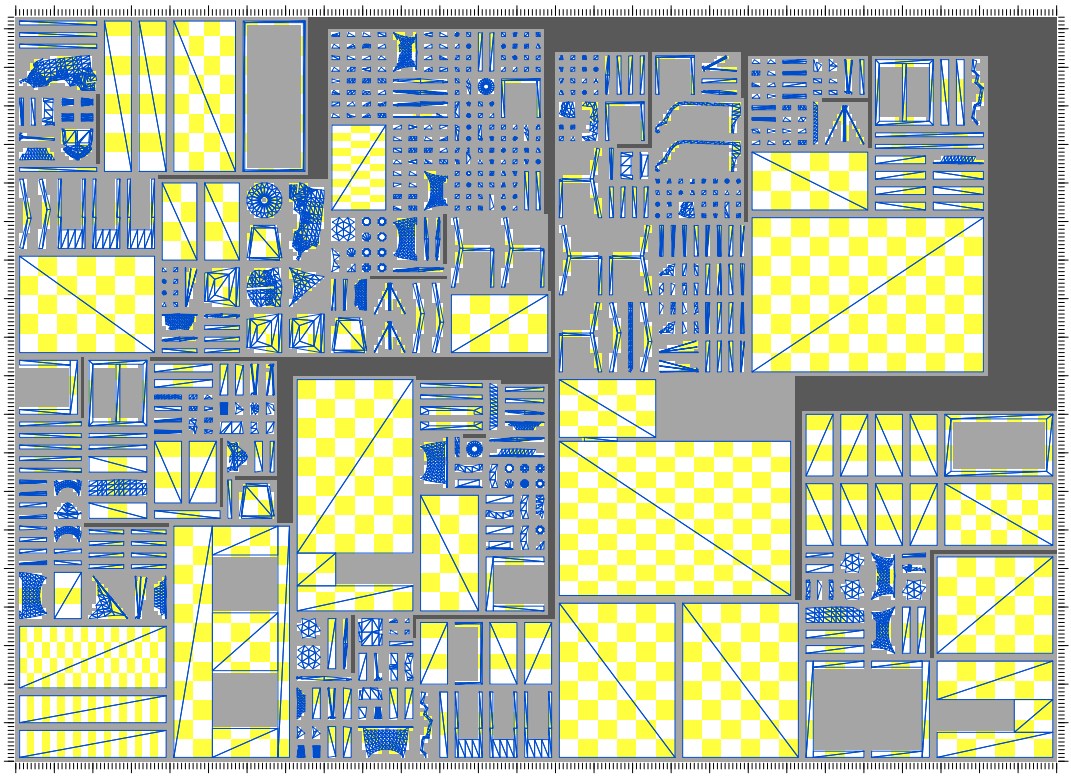
Wyniki opisanych wyżej obliczeń są zilustrowane na rysunkach 29.2, 29.3 i 29.4. Scena użyta do uruchomienia implementacji składa się z jedenastu obiektów opisanych łącznie

Listing 29.16. Szader obliczeniowy konstrukcji macierzy A

GLSL

```
#version 450 core

layout(local_size_x=1) in;

layout(std430,binding=0) buffer VarBuf { uvec4 tvar[]; } vbuf;
layout(std430,binding=4) buffer CPointBuf { vec4 cp[]; } cpbuf;
layout(std430,binding=5) buffer RowsCols { uint rc[];} rc;
layout(std430,binding=6) buffer Coeff { float a[]; } a;
layout(std430,binding=7) buffer VOIdBuf { uint id[]; } vobjid;
layout(std430,binding=8) buffer ObjIdBuf { uint id[]; } objid;

uniform CtlBlock { .... } ctl;  /* listing 29.5 */

void main ( void )
{
  uint  i, j, x, y;
  uvec4 tv;
  float s;

  i = gl_GlobalInvocationID.x;
  switch ( ctl.stage ) {
case 0:
    if ( i == 0 )
      rc.rc[0] = 0;
    else if ( i == ctl.nrows )
      rc.rc[ctl.nrows] = uint(ctl.ncols);
    else if ( vbuf.tvar[i].x > vbuf.tvar[i-1].x )
      rc.rc[vbuf.tvar[i].x] = i;
    rc.rc[ctl.nrows+1+i] = i;
    a.a[i] = cpbuf.cp[i].w;
    return;
case 1:
    s = a.a[rc.rc[i]];
    for ( j = rc.rc[i]+1; j < rc.rc[i+1]; j++ )
      s += a.a[j];
    for ( j = rc.rc[i]; j < rc.rc[i+1]; j++ )
      a.a[j] /= s;
    return;
  }
} /*main*/
```

przez 5729 trójkątów. Zbiór trójkątów został podzielony na 586 płatów, których otoczki wypukłe zmieściły się w prostokącie o wymiarach 270×193; a więc takie są wymiary dziedziny tekstury irradiancji.

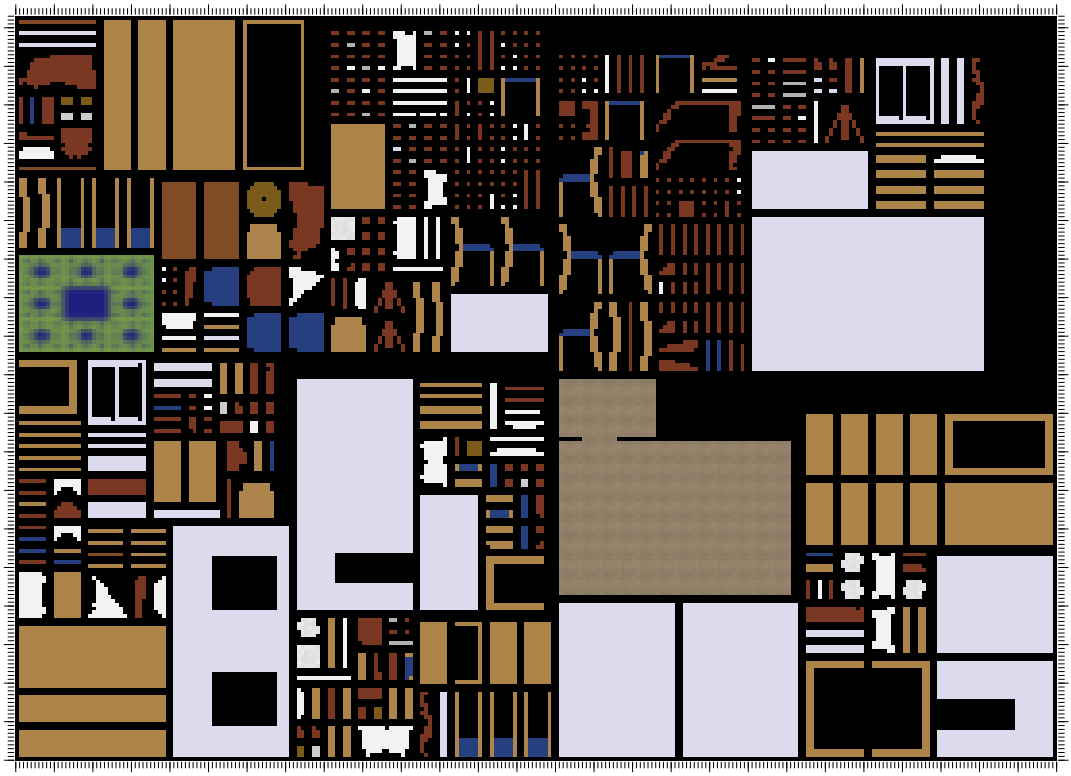


Rysunek 29.2. Mapa dyskretyzacji: rozmieszczenie trójkątów, elementów i makroelementów

Cienkie linie na rysunku 29.2 obrazują krawędzie trójkątów. Kolorem szarym są zaznaczone teksele nieużywane, przy czym teksele należące do prostokątów otaczających płaty są jaśniejsze. Istnieją płaty złożone z trójkątów tak małych, że cały płat został pokryty przez pojedynczy teksel; w tym przypadku powstał makroelement złożony z jednego elementu. Są też płaty duże. Ich obszary zostały podzielone na makroelementy uwidocznione za pomocą wzorów szachownic pokrywających prostokąty otaczające płaty; przedstawiony algorytm wygenerował 1858 makroelementów. Metodę rozmieszczania trójkątów można oczywiście ulepszyć, ale w tym przykładzie prostokąty pokrywają 90% dziedziny tekstury irradiancji, co jest wynikiem przyzwoitym. Tekseli używanych (odpowiadających elementom dyskretyzacji) jest 26382, czyli 50.6% wszystkich tekseli.

Na rysunku 29.3 jest pokazana zdolność poszczególnych elementów do odbijania światła. Kolory na obrazie są określone przez wektory na diagonali macierzy D, wzięte z opisów materiałów albo obliczone (przez ewaluatory wywołane w linii 29 szadera z listingu 29.10) w procesie filtrowania nałożonych na trójkąty tekstur.

Rysunek 29.4 przedstawia wizualizację wyników preprocesingu opisanego w tym punkcie. Trójkąty na obrazkach w górnej połowie rysunku mają kolory odpowiadające klasom, do których zostały zaliczone, i płatom. Na obrazach na dole są uwidocznione za pomocą tekstur elementy dyskretyzacji i makroelementy.



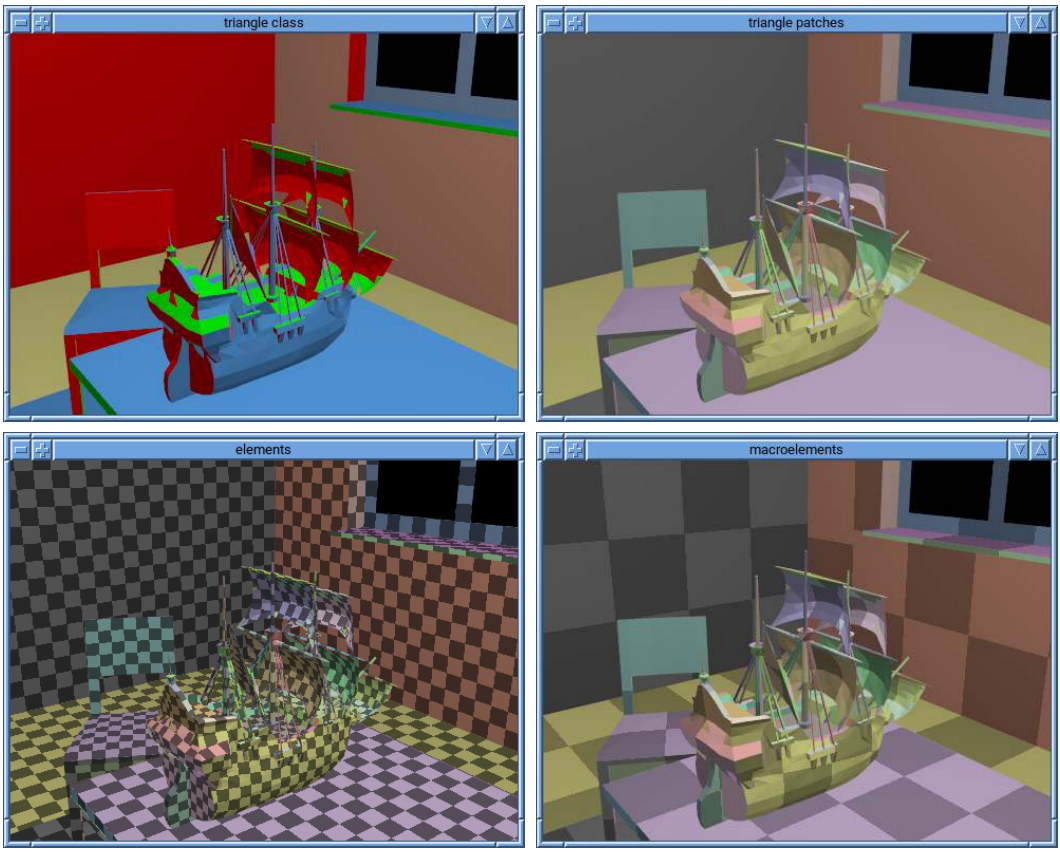
Rysunek 29.3. Mapa dyskretyzacji: zdolność odbijania światła przez elementy

29.2.6. Obliczanie współczynników kształtu

Do obliczenia współczynników G_{ij} macierzy G wykorzystamy GPU; w tym celu dla każdego elementu dyskretyzacji A_i, którego punkt kolokacji oznaczymy $\boldsymbol{p}_i$, trzeba wykonać pięć obrazów sceny w rzutach perspektywicznych o środku $\boldsymbol{p}_i$ na ściany prostopadłościennej kostki „zbudowanej nad" elementem A_i (rys. 29.5). Obrazy wykonamy przy włączonym rozstrzyganiu widoczności, przy czym wartości przypisywane pikselom (zamiast kolorów) będą numerami makroelementów. Każdy piksel ma swoją wagę, która na podstawie analogii Nusselta jest polem odpowiadającego mu obszaru w kole o polu 1.[20] Współczynnik G_{ij} jest (w przybliżeniu) sumą wag tych pikseli, które po narysowaniu sceny mają wartość j.

Mając n elementów, trzeba zatem wykonać obrazy sceny w $5n$ rzutach perspektywicznych; rozdzielczość tych obrazów determinuje dokładność otrzymanych wyników. Zauważmy, że jeśli makroelementy są bardzo małe, to z obrazów o zbyt małej rozdzielczości możemy nawet otrzymać 0 zamiast niektórych dodatnich współczynników G_{ij}. Przyjęcie większej rozdzielczości spowalnia obliczenia. Mamy tu więc klasyczny dylemat wyboru kompromisu między czasem obliczeń a dokładnością wyniku.

[20] Jest tu dokonane przekształcenie wspomniane na s. 787: współczynniki macierzy D mają wartości z przedziału $[0,1)$ zamiast $[0,1/\pi)$, a sumy współczynników w wierszach macierzy G nie mogą być większe niż 1 zamiast π.



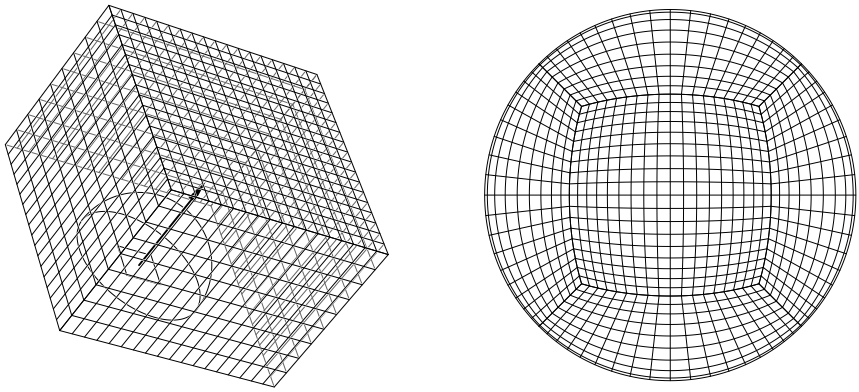
Rysunek 29.4. Podział zbioru trójkątów na klasy i płaty
i podział płatów na elementy dyskretyzacji i makroelementy

Aby przyspieszyć obliczenia, opisany dalej szader używany do obliczania współczynników kształtu, pracując w lokalnej grupie roboczej, korzysta ze **zmiennych współdzielonych** (*shared variables*); są one przechowywane w pamięci podręcznej, do której GPU ma szybki dostęp. Według specyfikacji OpenGL-a wielkość tej pamięci to co najmniej 32 kB. Rozdzielczość obrazów używanych do obliczania współczynników kształtu wybrałem tak, aby zmieścić się w tym limicie. W tym celu na górnej ścianie kostki przyjąłem rozdzielczość 36×36 pikseli, a na ścianach bocznych 36×18 pikseli[21]. W sumie na ścianach kostki będzie więc $N = 36 \cdot 36 + 4 \cdot 36 \cdot 18 = 3888$ pikseli.

Parametrem do wyboru jest proporcja długości boków bocznych ścian kostki. Wysokość ściany w jednostkach, w których szerokość jest równa 2, oznaczymy literą C. Liczbę C warto przyjąć tak, aby wartość wyrażenia

$$E = \sum_i (w_i - w_{\text{ave}})^2,$$

[21] Dla pewnych scen to może być za mało, ale w niektórych przypadkach wystarczy nawet 12×12 i 12×6.



Rysunek 29.5. Kostka służąca do obliczania współczynników kształtu

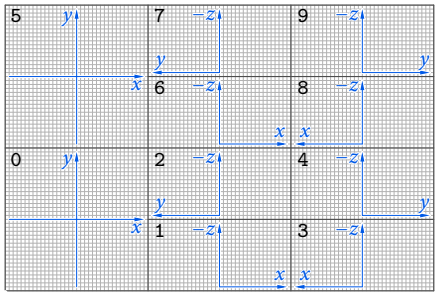
w którym w_i oznacza wagę i-tego piksela, a $w_{\text{ave}} = 1/N$ oznacza wagę średnią, była najmniejsza. To jest osiągane dla $C \approx 1.543$.

Wykonywanie $5n$ obrazów tej samej sceny w różnych rzutach perspektywicznych można częściowo zrównoleglić. OpenGL umożliwia wykonanie wielu obrazów jednocześnie, w różnych klatkach. Według specyfikacji można rysować jednocześnie w co najwyżej 16 klatkach, nie jest więc możliwe równoległe wykonanie $5n$ obrazów. Ale można narysować scenę n razy, za każdym razem w pięciu różnych rzutach. Co więcej, dla każdego rzutu wykonamy dwa obrazy, przy użyciu dwóch różnych macierzy przejścia od układu współrzędnych obserwatora do układu kostki standardowej, skonstruowanych dla dwóch różnych zakresów głębokości. Ma to na celu poprawienie dokładności algorytmu rozstrzygania widoczności.

Średnica (maksymalna odległość punktów) sceny użytej do uruchomienia implementacji metody bilansu energetycznego to prawie 9 jednostek układu świata (metrów), podczas gdy pewne detale są ponad 3 rzędy wielkości mniejsze (rzędu 0.005, tj. 5 mm). Rachunek na s. 346 pokazuje, że algorytm widoczności dla takich danych ma marne szanse na poprawne działanie. Aby to naprawić, przedział $[-f, -n]$ występujących w scenie współrzędnych z przetwarzanych punktów w układach obserwatora podzielimy na podprzedziały $[-f, -s]$ i $[-s, -n]$, przyjmując za s średnią geometryczną krańców przedziału. Możemy sprawdzić, że wtedy dokładność reprezentacji (przechowywanych w buforze głębokości) liczb ζ reprezentujących głębokości punktów brył widzenia otrzymanych z $n = 0.0025$, $s = 0.16$, $f = 10.24$ umożliwia rozróżnianie głębokości (w przestrzeni) różniących się o więcej niż $3.8 \cdot 10^{-5}$.

Na rysunku 29.6 jest pokazany sposób rozmieszczenia klatek w załącznikach bufora ramki używanego do obliczania współczynników kształtu i zaznaczone są też kierunki osi pomocniczego układu współrzędnych, którego początkiem jest punkt kolokacji $\boldsymbol{p}_i$, a wersor osi z ma kierunek wektora normalnego trójkąta, na którym leży i-ty element, i przeciwny zwrot. W klatkach o numerach 0–4 mają powstawać obrazy obiektów położonych blisko punktu $\boldsymbol{p}_i$ (których współrzędna z jest mniejsza niż s), a w pozostałych klatkach obrazy obiektów bardziej oddalonych.

Procedura `LoadFFViewports` (listing 29.17) wywołuje procedurę `glViewportArrayv`, aby wprowadzić dziesięć klatek, w których będą powstawać obrazy. Makrodefinicje `FS` i `HS`



Rysunek 29.6. Klatki obrazów wykonywanych w celu obliczania współczynników kształtu

ukrywają liczby 36.0 i 18.0 określające wymiary klatek i ich położenia. Inaczej niż w przypadku procedury `glViewport`, wymiary i położenia klatek muszą być podane jako liczby zmiennopozycyjne.

Listing 29.17. Procedura `LoadFFViewports`

C

```
1: void LoadFFViewports ( FFTransBl *fftrans )
2: {
3:   static const GLfloat viewports[10*4] =
4:     { 0.0,0.0,FS,FS, FS,0.0,FS,HS, FS,HS,FS,HS, 2*FS,0.0,FS,HS,
5:       2*FS,HS,FS,HS, 0.0,FS,FS,FS, FS,FS,FS,HS, FS,FS+HS,FS,HS,
6:       2*FS,FS,FS,HS, 2*FS,FS+HS,FS,HS };
7:
8:   glViewportArrayv ( 0, 10, viewports );
9: } /*LoadFFViewports*/
```

Ponieważ liczba elementów dyskretyzacji jest duża (rzędu kilkudziesięciu tysięcy), obliczanie współczynników kształtu będzie dokonane w **blokach**. Rozmiar bloku ma symboliczną nazwę `FFBLOCKSIZE`; przyjąłem, że jest to 1024. W razie zwiększenia liczby elementów (wskutek zmniejszenia ich średnic lub dodania do sceny nowych obiektów), nawet bardzo potężne GPU mogą nie pomieścić danych „w jednym kawałku”.

Na listingu 29.18 jest przedstawiona procedura, której zadaniem jest obliczenie współczynników kształtu i utworzenie reprezentacji macierzy G z tymi współczynnikami. W tym miejscu podaję ogólny opis jej działania i wywoływanych przez nią procedur, zostawiając szczegóły na później. Procedura `SetupObjMMti`, wywołana w linii 9, dla każdego obiektu oblicza transpozycję odwrotności macierzy przekształcenia modelu i zapisuje ją w polu `mmti` struktury typu `BalanceObject` będącej częścią reprezentacji obiektu.

Procedury `ComputeFFWeights` (listing 29.19) i `ConstructFFPMatrices` (listing 29.21) mają za zadanie obliczyć wagi pikseli i skonstruować macierz przejścia od układu obserwatora do układu kostki standardowej dla dziesięciu rzutów perspektywicznych, z którymi będą wykonywane obrazy dla każdego elementu.

Listing 29.18. Procedura `ComputeFormFactors`

C

```
void ComputeFormFactors ( BalanceElements *belem )
{
  GLuint          fffbo, ffbuf[4], *ffmatblbuf;
  GPUSparseMatrix *ffbldesc;
  GLfloat         *elcp, cp[4], nv[3], *mm, *mmti;
  int             nelem, nblocks, N, t, b, i, j, k;
  GLuint          *vardata, inval = RESTART_IND_UINT, on;

  SetupObjMMti ( belem );
  ComputeFFWeights ( belem, ffbuf );
  ConstructFFPMatrices ( &belem->fftr, belem->ffnear, belem->fffar );
  N = 3*FFTXTSIZE*FFTXTSIZE;  /* 3*36*36 */
  ffbuf[1] = NewStorageBuffer ( ((FFBLOCKSIZE+2)*N+1)*sizeof(GLuint), 7 );
  ffbuf[2] = NewStorageBuffer ( (FFBLOCKSIZE*N+1)*sizeof(GLuint), 4 );
  ffbuf[3] = NewStorageBuffer ( (FFBLOCKSIZE*N+1)*sizeof(GLuint), 5 );
  fffbo = PrepareFFFramebuffer ( belem );
  glBindImageTexture ( 0, belem->fftxt[0], 0, GL_FALSE, 0, GL_READ_ONLY,
                       GL_R32UI );
  LoadFFViewports ( &belem->fftr );
  nelem = belem->nelem;
  if ( !(elcp = malloc ( nelem*4*sizeof(GLfloat) )) )
    ExitOnError ( "ComputeFormFactors 1" );
  glGetNamedBufferSubData ( BUF_CP, 0, nelem*4*sizeof(GLfloat), elcp );
  glBindVertexArray ( belem->tvao );
  nblocks = (nelem+FFBLOCKSIZE-1) / FFBLOCKSIZE;
  if ( !(ffmatblbuf = malloc ( 2*nblocks*sizeof(GLuint) )) ||
       !(ffbldesc = malloc ( nblocks*sizeof(GPUSparseMatrix) )) )
    ExitOnError ( "ComputeFormFactors 2" );
  memset ( ffbldesc, 0, nblocks*sizeof(GPUSparseMatrix) );
  glGenBuffers ( 2*nblocks, ffmatblbuf );
  for ( b = j = 0;  b < nblocks;  b++, j += 2 ) {
    ffbldesc[b].buf[0] = ffmatblbuf[j];
    ffbldesc[b].buf[1] = ffmatblbuf[j+1];
  }
  vardata = belem->VarBuf;
  on = RESTART_IND_UINT;
  mm = mmti = NULL;
  glEnable ( GL_DEPTH_TEST );
  glDisable ( GL_DEPTH_CLAMP );
  for ( b = i = k = 0;  i < nelem;  b++ ) {
    glClearNamedBufferData ( ffbuf[1], GL_R8UI, GL_RED,
                             GL_UNSIGNED_INT, &inval );
    for ( j = 0;  j < FFBLOCKSIZE && i < nelem;  j++, i++, k += 4 ) {
      t = vardata[k+1];
      if ( on != belem->ObjIdBuf[i] ) {
```

```
46:         on = belem->ObjIdBuf[i];
47:         mm = belem->objtab[on].obj->mm;
48:         mmti = belem->objtab[on].mmti;
49:       }
50:       M4x4MultMP3f ( cp, mm, &elcp[k] );
51:       M4x4MultMV3f ( nv, mmti, belem->trdesc[t].nvect );
52:       DrawViewFromElem ( belem, bprog_id[7], bprog_id[8],
53:                          fftrbbp, fftrbofs, j, cp, nv );
54:     }
55:     ffbldesc[b].m = j;
56:     ffbldesc[b].n = belem->nmacroelem;
57:     GPUComputeFF ( belem, ffbuf, j, &ffbldesc[b] );
58:   }
59:   AssembleFFMatrix ( belem, nblocks, ffbldesc );
60:   free ( elcp );
61:   glBindVertexArray ( 0 );
62:   glUseProgram ( 0 );
63:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
64:   glDeleteFramebuffers ( 1, &fffbo );
65:   glDeleteTextures ( 2, belem->fftxt );
66:   glDeleteBuffers ( 4, ffbuf );
67:   glDeleteBuffers ( 2*nblocks, ffmatblbuf );
68:   free ( ffmatblbuf );
69:   free ( ffbldesc );
70:   ExitIfGLError ( "ComputeFormFactors" );
71: } /*ComputeFormFactors*/
```

W liniach 13–15 są tworzone potrzebne podczas obliczeń bufory magazynowe. Procedura `PrepareFFFramebuffer` (listing 29.22) przygotowuje pozaekranowy bufor ramki z odpowiednimi załącznikami i podaje jego identyfikator.

Tekstura o identyfikatorze zapamiętanym w zmiennej `belem->fftxt[0]` jest załącznikiem bufora ramki, w którego tekselach program wykonujący obrazy zapamięta numery makroelementów. Obraz tej tekstury musi być widoczny dla szadera, który skopiuje dane z niej do bufora magazynowego do dalszego przetwarzania, dlatego w liniach 17–18 jest on przywiązywany do odpowiedniego punktu dowiązania.

W linii 23 z bufora `BUF_CP` odczytywane są położenia punktów kolokacji poszczególnych elementów, obliczone we wcześniejszej fazie preprocesingu i dane w układach modeli poszczególnych obiektów.

W linii 25 obliczana jest liczba bloków, na które zostanie podzielone obliczenie macierzy G. Reprezentacja każdego z tych bloków, taka jak reprezentacje innych macierzy rzadkich, jest przechowywana w dwóch buforach z trzema tablicami. Identyfikatory tych buforów są rezerwowane w linii 30, a w pętli w liniach 31–34 są one zapamiętywane w opakowaniach reprezentacji każdego bloku.

Pętla w liniach 40–58 przebiega przez wszystkie bloki. Procedura `glClearNamedBufferData` kasuje bufor magazynowy, tzn. nadaje wszystkim zmiennym typu `uint` w tablicy w tym

buforze wartość 0xFFFFFFFF. **Uwaga:** parametry wydają polecenie nadania wartości 0xFF wszystkim *bajtom* w tym buforze, co jest niezbyt eleganckim trikiem, ale odnosi pożądany skutek. Podanie drugiego parametru GL_R32UI skutkowało w moim laptopie skasowaniem 20 najbardziej znaczących bitów podanej liczby. Wygląda to na błąd w sterowniku[22].

Pętla w liniach 43–54 przebiega przez kolejne elementy dyskretyzacji w bloku. W linii 44 zmiennej t jest przypisywany indeks trójkąta, na którym leży element. Jeśli numer bieżącego obiektu jest inny niż numer obiektu, do którego należy element przetwarzany w poprzednim przebiegu pętli, to zmiennym mm i mmti są przypisywane adresy tablic z macierzą przekształcenia modelu i transpozycją jej odwrotności. W liniach 50 i 51 są obliczane współrzędne punktu kolokacji i wektora normalnego trójkąta w układzie świata[23].

Procedura DrawViewFromElem (listing 29.23) wykonuje obrazy sceny widzianej z punktu kolokacji na j-tym elemencie w bloku. Punkt ten i wektor normalny elementu są podane jako ostatnie dwa parametry. Dane z obrazów są przepisywane do bufora magazynowego. Po zakończeniu wewnętrznej pętli następuje wywołanie procedury GPUComputeFF (listing 29.29), która oblicza współczynniki kształtu dla całego bloku przy użyciu szadera obliczeniowego.

Procedura AssembleFFMatrix (listing 29.32) „składa" całą macierz G z obliczonych wcześniej bloków, po czym następuje sprzątanie, tj. likwidacja użytego bufora ramki, tekstur i buforów magazynowych z reprezentacjami tych bloków.

Procedura ComputeFFWeights (listing 29.19) w linii 6 tworzy bufor magazynowy z tablicą, w której będą przechowywane wagi pikseli. Ma się w niej pomieścić $(B+1)N$ liczb zmiennopozycyjnych, gdzie N jest liczbą pikseli (równą 3888), a B oznacza wielkość bloku (czyli 1024). Wagi zostaną obliczone i zapamiętane na pierwszych N miejscach w tablicy. W celu obliczenia współczynników kształtu dla j-tego elementu w bloku te N liczb zostanie posortowane według zapamiętanych w pikselach numerów makroelementów i skopiowane do tablicy, zaczynając od miejsca $(j+1)N$, po czym wagi odpowiadające każdemu makroelementowi zostaną zsumowane.

Listing 29.19. Procedura obliczania wag pikseli

C

```
1:  static void ComputeFFWeights ( BalanceElements *belem, GLuint ffbuf[4] )
2:  {
3:    int n, N;
4:
5:    N = 3*FFTXTSIZE*FFTXTSIZE;  /* 3*36*36 */
6:    ffbuf[0] = NewStorageBuffer ( (FFBLOCKSIZE+1)*N*sizeof(GLfloat), 3 );
7:    glUseProgram ( bprog_id[6] );  /* listing 29.20 */
8:    SetCTLUniformf ( belem, CTL_C, CF );
9:    SetCTLUniformi ( belem, CTL_STAGE, 0 );
10:   COMPUTE ( FFTXTSIZE/2, FFTXTSIZE/2, 1 )
11:   SetCTLUniformi ( belem, CTL_STAGE, 1 );
```

[22] Zobacz cytat w przypisie na s. 799.

[23] Przechowywanie współrzędnych punktów kolokacji w układach modeli umożliwia wykonanie części preprocesingu opisanej w p. 29.2.2–29.2.5 tylko raz, bez potrzeby powtarzania go po każdej zmianie macierzy przekształcenia modelu.

```
12:   SetCTLUniformi ( belem, CTL_H, N );
13:   COMPUTE ( N, 1, 1 )
14:   SetCTLUniformi ( belem, CTL_STAGE, 2 );
15:   for ( n = N; n > 1; n = (n+1)/2 ) {
16:     SetCTLUniformi ( belem, CTL_N, n );
17:     COMPUTE ( n/2, 1, 1 )
18:   }
19:   SetCTLUniformi ( belem, CTL_STAGE, 3 );
20:   SetCTLUniformi ( belem, CTL_N, N );
21:   COMPUTE ( N, 1, 1 )
22:   ExitIfGLError ( "ComputeFFWeights" );
23: } /*ComputeFFWeights*/
```

W liniach 10, 13, 17 i 21 następuje uruchamianie programu z szaderem z listingu 29.20, który realizuje kolejne etapy obliczania wag. Etap 2, który realizuje algorytm sumowania parami, jest wykonywany w $\lceil \log_2 N \rceil$ przebiegach pętli w liniach 15–18.

Zbadajmy teraz szader. Do obliczenia wag pikseli służą następujące wzory: niech $h = 2/36$ — jest to długość boków piksela na górnej ścianie kostki (rys. 29.5); piksele na ścianach bocznych mają wymiary $h \times Ch$. Jeśli środek piksela na górnej ścianie kostki ma współrzędne (x, y, C), to jego wagę możemy obliczyć ze wzoru

$$w \approx \frac{C^2}{\pi h^2 d^2}, \quad \text{gdzie } d = x^2 + y^2 + C^2,$$

a dla piksela na bocznej ścianie, którego środek ma współrzędne $(1, y, z)$, zastosujemy wzór

$$w \approx \frac{Cz}{\pi h^2 d^2}, \quad \text{gdzie } d = 1 + y^2 + z^2.$$

Zamiast tych wzorów w procedurze `ComputeWeights` są zapisane (w liniach 24 i 32) instrukcje obliczania wyrażeń C/d^2 i z/d^2, z pominięciem wspólnego czynnika wszystkich wag $C/(\pi h^2)$. Symetria kostki i podziału jej ścian na piksele umożliwia obliczenie wag pikseli tylko na jednej czwartej górnej ściany kostki i na połowie ściany bocznej; w liniach 25 oraz 33–34 obliczone wyrażenia są zapisywane w tablicy zgodnie z tą symetrią. Ponieważ podane wyrażenia opisują wagi w przybliżeniu[24], suma ich wartości *nie jest* równa 1. Dlatego w ostatnich trzech etapach szader oblicza sumę obliczonych wyrażeń i dzieli przez nią je wszystkie, otrzymując wagi o sumie równej 1.

Listing 29.20. Szader obliczający wagi pikseli

GLSL

```
1: #version 450 core
2:
3: #define HW 18
4:
```

[24] Pole krzywoliniowego czworokąta w kole, które jest wagą piksela na ścianie kostki, jest całką, która tu jest przybliżana przez kwadraturę opartą na jednym węźle. Sumowanie tych wag w celu obliczenia współczynników kształtu jest numerycznym całkowaniem.

```
5: layout(local_size_x=1) in;
6:
7: layout(std430,binding=3) buffer FFWeightBuf { float w[]; } wbuf;
8:
9: uniform CtlBlock { .... } ctl;  /* listing 29.5 */
10:
11: #define WPIX(I,J) wbuf.w[(J)*6*HW+(I)]
12:
13: void ComputeWeights ( uvec3 u )
14: {
15:   vec3  p;
16:   float d, c, s, f;
17:   uint  i0, i1, i2, i3, j0, j1;
18:
19:   i0 = HW-1-u.x;  i1 = HW+u.x;
20:   j0 = HW-1-u.y;  j1 = HW+u.y;
21:   p = vec3 ( (float(u.x)+0.5)/float(HW),
22:              (float(u.y)+0.5)/float(HW), ctl.C );
23:   d = dot ( p, p );
24:   f = ctl.C/(d*d);
25:   WPIX(i0,j0) = WPIX(i0,j1) = WPIX(i1,j0) = WPIX(i1,j1) = f;
26:   i0 = 3*HW-1-u.x;  i1 = 3*HW+u.x;
27:   i2 = i0+2*HW;  i3 = i1+2*HW;
28:   j0 = u.y;  j1 = HW+u.y;
29:   p = vec3 ( 1.0, (float(u.x)+0.5)/float(HW),
30:              ctl.C*(float(u.y)+0.5)/float(HW) );
31:   d = dot ( p, p );
32:   f = p.z/(d*d);
33:   WPIX(i0,j0) = WPIX(i1,j0) = WPIX(i2,j0) = WPIX(i3,j0) =
34:   WPIX(i0,j1) = WPIX(i1,j1) = WPIX(i2,j1) = WPIX(i3,j1) = f;
35: } /*ComputeWeights*/
36:
37: void main ( void )
38: {
39:   uvec3 inv;
40:   uint  i;
41:
42:   inv = gl_GlobalInvocationID;
43:   switch ( ctl.stage ) {
44: case 0:
45:     ComputeWeights ( inv );
46:     return;
47: case 1:  /* kopiowanie do sumowania */
48:     wbuf.w[ctl.H+inv.x] = wbuf.w[inv.x];
49:     return;
50: case 2:  /* sumowanie parami */
51:     if ( (i = inv.x+(ctl.N+1)/2) < ctl.N )
```

```
52:       wbuf.w[ctl.H+inv.x] += wbuf.w[ctl.H+i];
53:     return;
54: case 3:  /* normalizacja */
55:     wbuf.w[inv.x] /= wbuf.w[ctl.N];
56:     return;
57: default:
58:     return;
59:   }
60: } /*main*/
```

Etap 1 kopiuje N liczb z początku tablicy na kolejne N miejsc. Etap 2 dokonuje sumowania (algorytm sumowania parami „psuje" kopię, oryginalne składniki muszą pozostać niezmienione), a w etapie 3 następuje dzielenie przez sumę pozostawioną przez etap 2 w tablicy na miejscu N.

Procedura `ConstructFFPMatrices` (listing 29.21) konstruuje 10 macierzy przejścia od układu obserwatora do układu kostki standardowej, z których każda będzie używana do wykonania obrazów w jednej z klatek pokazanych na rysunku 29.6. Przez te macierze będą mnożone wektory współrzędnych jednorodnych wierzchołków podane w układzie, którego początkiem jest punkt kolokacji, a wersor osi z ma kierunek wektora normalnego bieżącego elementu i przeciwny zwrot. Macierze te zostają zapamiętane w tablicy `pm` będącej polem struktury typu `FFTransBl` otrzymanej jako parametr. Parametry `near` i `far` wyznaczają potrzebny zakres głębokości (aplikacja podała go, wywołując procedurę `BeginEnteringBalanceElements`). Teraz (w linii 14) jest obliczana ich średnia geometryczna, która zostanie punktem podziału tego zakresu na dwa podprzedziały.

Listing 29.21. Procedura `ConstructFFPMatrices`

C

```
1: typedef struct FFTransBl {
2:     GLuint  fftrbuf;
3:     GLfloat mm[16], pm[10][16], vpm[10][16];
4:   } FFTransBl;
5:
6: void ConstructFFPMatrices ( FFTransBl *fftrans, float near, float far )
7: {
8:   GLfloat pm[16];
9:   static const GLfloat rotm[2][16] =
10:     {{1.0,0.0,0.0,0.0, 0.0,0.0,1.0,0.0, 0.0,-1.0,0.0,0.0, 0.0,0.0,0.0,1.0},
11:      {0.0,1.0,0.0,0.0, -1.0,0.0,0.0,0.0, 0.0,0.0,1.0,0.0, 0.0,0.0,0.0,1.0}};
12:   float nf;
13:
14:   nf = sqrt ( near*far );
15:   M4x4Frustumf ( fftrans->pm[0], NULL,
16:                  -near/CF, near/CF, -near/CF, near/CF, near, nf );
17:   M4x4Frustumf ( pm, NULL, -near, near, 0.0, near*CF, near, nf );
18:   M4x4Multf ( fftrans->pm[1], pm, rotm[0] );
19:   M4x4Multf ( fftrans->pm[2], fftrans->pm[1], rotm[1] );
20:   M4x4Multf ( fftrans->pm[3], fftrans->pm[2], rotm[1] );
```

```
21:   M4x4Multf ( fftrans->pm[4], fftrans->pm[3], rotm[1] );
22:   M4x4Frustumf ( fftrans->pm[5], NULL,
23:                  -nf/CF, nf/CF, -nf/CF, nf/CF, nf, far );
24:   M4x4Frustumf ( pm, NULL, -nf, nf, 0.0, nf*CF, nf, far );
25:   M4x4Multf ( fftrans->pm[6], pm, rotm[0] );
26:   M4x4Multf ( fftrans->pm[7], fftrans->pm[6], rotm[1] );
27:   M4x4Multf ( fftrans->pm[8], fftrans->pm[7], rotm[1] );
28:   M4x4Multf ( fftrans->pm[9], fftrans->pm[8], rotm[1] );
29: } /*ConstructFFPMatrices*/
```

Procedura `M4x4Frustumf` wywołana w liniach 15–16 oraz 22–23 konstruuje macierze dla rzutowania na górną ścianę (kwadrat 2×2) kostki przedstawionej na rysunku 29.5. Macierze konstruowane w liniach 17 i 24 określają rzutowanie na prostokąt o wymiarach $2 \times C$ położony w płaszczyźnie $z = -1$.

W liniach 18 i 25 te macierze są mnożone przez macierz obrotu o kąt prosty wokół osi x, przez co powstają macierze rzutowania na ścianę kostki położoną w płaszczyźnie $y = -1$. Druga macierz 4×4, której współczynniki są podane w tablicy `rotm`, reprezentuje obrót o kąt prosty wokół osi z. Kolejne mnożenia przez tę macierz wytwarzają macierze rzutowania na pozostałe ściany boczne.

Procedura na listingu 29.22 przygotowuje pozaekranowy bufor ramki, przy użyciu którego będą tworzone obrazy sceny potrzebne do obliczania współczynników kształtu. Bufor ten ma dwa załączniki, bufor obrazu i bufor głębokości, które są teksturami o wymiarach 108×72 (rys. 29.6). Punkt dowiązania używany do określenia wszelkich parametrów tekstur przez wywoływane procedury jest ostatnim punktem określonym w specyfikacji; punkty, do których są dowiązane inne tekstury (nałożone na obiekty, reprezentujące obszary cienia, tekstura irradiancji itd.), mają inne numery. Tekstura używana jako bufor obrazu ma piksele o jednej składowej reprezentowanej jako 32-bitowa liczba całkowita bez znaku.

Listing 29.22. Procedura `PrepareFFFramebuffer`

C

```
1: static GLuint PrepareFFFramebuffer ( BalanceElements *belem )
2: {
3:   GLuint       fffbo;
4:   const GLenum buffers = GL_COLOR_ATTACHMENT0;
5:
6:   glGenTextures ( 2, belem->fftxt );
7:   glActiveTexture ( GL_TEXTURE0+47 );
8:   glBindTexture ( GL_TEXTURE_2D, belem->fftxt[0] );
9:   glTexStorage2D ( GL_TEXTURE_2D, 1, GL_R32UI, 3*FFTXTSIZE, 2*FFTXTSIZE );
10:  glBindTexture ( GL_TEXTURE_2D, belem->fftxt[1] );
11:  glTexStorage2D ( GL_TEXTURE_2D, 1, GL_DEPTH_COMPONENT32F,
12:                   3*FFTXTSIZE, 2*FFTXTSIZE );
13:  glGenFramebuffers ( 1, &fffbo );
14:  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, fffbo );
15:  glFramebufferTexture ( GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
16:                         belem->fftxt[0], 0 );
```

```
17:   glFramebufferTexture ( GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
18:                          belem->fftxt[1], 0 );
19:   glDrawBuffers ( 1, &buffers );
20:   glBindTexture ( GL_TEXTURE_2D, 0 );
21:   if ( glCheckFramebufferStatus ( GL_DRAW_FRAMEBUFFER ) !=
22:         GL_FRAMEBUFFER_COMPLETE )
23:     ExitOnError ( "PrepareFFFramebuffer" );
24:   ExitIfGLError ( "PrepareFFFramebuffer" );
25:   return fffbo;
26: } /*PrepareFFFramebuffer*/
```

Procedura `DrawViewFromElem` (listing 29.23) ma za zadanie utworzyć obrazy sceny widzianej z punktu kolokacji, którego współrzędne są podane w tablicy `elcp`, przy czym w tablicy `elnv` jest podany wektor normalny elementu. Program użyty do rysowania składa się z szaderów zamieszczonych na listingach 29.24, 29.25 i 29.26[25]. Procedura wywołana w liniach 10–11 nadaje wszystkim pikselom wartość `0xFFFFFFFF`, która reprezentuje (nieświecące) tło. W linii 13 jest wywoływana opisana dalej procedura, która oblicza macierz przejścia od układu świata do układu obserwatora dla podanego punktu kolokacji. W pętli są rysowane kolejno wszystkie obiekty; przed rysowaniem każdego z nich do bloku zmiennych jednolitych `FFTransBlock` jest przesyłana odpowiednia macierz przekształcenia modelu. Po zakończeniu rysowania do pracy przystępuje program z pokazanym na listingu 29.27 szaderem obliczeniowym, który wstępnie przetwarza otrzymane obrazy.

Listing 29.23. Procedura `DrawViewFromElem`

C

```
1: void DrawViewFromElem ( BalanceElements *belem, GLuint prog_id0,
2:                         GLuint prog_id1, GLuint bp, GLint *ofs,
3:                         int j, GLfloat elcp[3], GLfloat elnv[3] )
4: {
5:   BalanceObject *bobj;
6:   int           i;
7:   GLuint        inval = RESTART_IND_UINT;
8:
9:   glUseProgram ( prog_id0 );  /* listingi 29.24, 29.25, 29.26 */
10:  glClearTexImage ( belem->fftxt[0], 0, GL_RED_INTEGER, GL_UNSIGNED_INT,
11:                    &inval );
12:  glClear ( GL_DEPTH_BUFFER_BIT );
13:  LoadFFVPMatrices ( &belem->fftr, elcp, elnv );  /* listing 29.28 */
14:  for ( i = 0; i < belem->nobj; i++ ) {
15:    bobj = &belem->objtab[i];
16:    LoadFFMMatrix ( bp, ofs, &belem->fftr, bobj->obj->mm );
17:    glDrawElements ( GL_TRIANGLES, 3*bobj->cntr, GL_UNSIGNED_INT,
18:                     (GLvoid*)(bobj->ftrdesc*3*sizeof(GLuint)) );
19:  }
```

[25] Identyfikatory programów są parametrami procedury, co jest pozostałością po moich eksperymentach wykonywanych podczas uruchamiania implementacji.

```
20:   glFinish ();
21:   glUseProgram ( prog_id1 );  /* listing 29.27 */
22:   SetCTLUniformui ( belem, CTL_P0, (GLuint)(j*3*FFTXTSIZE*FFTXTSIZE) );
23:   COMPUTE ( 3*FFTXTSIZE, FFTXTSIZE, 1 );
24:   ExitIfGLError ( "DrawViewFromElem" );
25: } /*DrawViewFromElem*/
```

Zadanie szadera wierzchołków jest bardzo proste — ma on tylko obliczyć i przesłać na wyjście położenie wierzchołka w układzie świata oraz jego współrzędne w dziedzinie tekstury irradiancji.

Listing 29.24. Szader wierzchołków programu obliczania współczynników kształtu

GLSL

```
1: #version 450 core
2:
3: layout(location=0) in vec4 in_Position;
4: layout(location=2) in vec2 BTxtCoord;
5:
6: out vec2 BTexCoord;
7:
8: uniform FFTransBlock { mat4 mm;  mat4 vpm[10]; } fftr;
9:
10: void main ( void )
11: {
12:   BTexCoord = BTxtCoord;
13:   gl_Position = fftr.mm * in_Position;
14: } /*main*/
```

Zgodnie z kwalifikatorem w linii 3 szader geometrii (listing 29.25) jest dla każdego trójkąta wywoływany w dziesięciu instancjach. Numer instancji jest odczytywany w linii 15 i przypisywany zmiennej wyjściowej `gl_ViewportIndex`, która wyznacza klatkę, do której trafi wynik obcinania i rasteryzacji rysowanego trójkąta. Przypisywana zmiennej wyjściowej `gl_Position` wartość jest wektorem współrzędnych jednorodnych podanych w układzie kostki standardowej dla rzutowania perspektywicznego wybranego za pomocą numeru instancji. Macierze w tablicy `fftr.vpm` są iloczynami (jednej) macierzy przejścia do układu obserwatora z macierzami przekształceń perspektywicznych skonstruowanymi przez procedurę `ConstructFFPMatrices`.

Listing 29.25. Szader geometrii programu obliczania współczynników kształtu

GLSL

```
1: #version 450 core
2:
3: layout(triangles,invocations=10) in;
4: layout(triangle_strip,max_vertices=3) out;
5:
6: in vec2 BTexCoord[];
```

```
7: out vec2 BTxCoord;
8:
9: uniform FFTransBlock { mat4 mm;  mat4 vpm[10]; } fftr;
10:
11: void main ( void )
12: {
13:   int i, j;
14:
15:   j = gl_InvocationID;
16:   for ( i = 0; i < 3; i++ ) {
17:     gl_ViewportIndex = j;
18:     gl_Position = fftr.vpm[j] * gl_in[i].gl_Position;
19:     BTxCoord = BTexCoord[i];
20:     EmitVertex ();
21:   }
22:   EndPrimitive ();
23: } /*main*/
```

Wyjściem szadera fragmentów na listingu 29.26 jest numer makroelementu widocznego w danym pikselu. Współrzędne tekstury irradiancji fragmentu są w linii 21 zaokrąglane do liczb całkowitych i użyte do obliczenia indeksu do (indeksowanej współrzędnymi teksela) tablicy w buforze BUF_VMAP. W składowej x elementu tej tablicy jest podany numer elementu, lub jeśli teksel jest nieużywany, składowa ta ma wartość 0xFFFFFFFF. W pierwszym przypadku zmiennej wyjściowej jest przypisywany numer makroelementu, do którego element należy (jest on odczytywany ze składowej x odpowiedniego elementu tablicy w buforze BUF_VBUF), a w drugim przypadku (który może się zdarzyć wskutek niedokładności rasteryzacji) fragment jest odrzucany.

Listing 29.26. Szader fragmentów programu obliczania współczynników kształtu

GLSL

```
1: #version 450 core
2:
3: #define RESTART_IND 0xFFFFFFFF
4:
5: layout(early_fragment_tests) in;
6:
7: in vec2 BTxCoord;
8:
9: layout(location=0) out uint mel;
10:
11: layout(std430,binding=0) buffer VarBuf { uvec4 tvar[]; } vbuf;
12: layout(std430,binding=9) buffer VMapBuf { uvec2 px[]; } vmap;
13:
14: uniform CtlBlock { .... } ctl;  /* listing 29.5 */
15:
```

```
16: void main ( void )
17: {
18:   int  z;
19:   uint v;
20:
21:   z = int(BTxCoord.y)*ctl.width + int(BTxCoord.x);
22:   if ( (v = vmap.px[z].x) != RESTART_IND )
23:     mel = vbuf.tvar[v].x;
24:   else
25:     discard;
26: } /*main*/
```

Szader pokazany na listingu 29.27 jest wywoływany w grupie roboczej o wymiarach $108 \times 36 \times 1$. Każdy jego wątek odpowiada jednemu pikselowi w dolnej połowie tekstury, w której powstał obraz (rys. 29.6). Szader ten czyta liczbę przypisaną pikselowi i jeśli ta liczba jest różna od 0xFFFFFFFF, to jest ona numerem widocznego w pikselu makroelementu. W przeciwnym razie odczytywana jest liczba zapamiętana w odpowiednim pikselu w górnej połowie tekstury — wtedy ta liczba jest numerem widocznego makroelementu albo jest to liczba 0xFFFFFFFF reprezentująca tło.

Listing 29.27. Szader obliczeniowy drugiego programu obliczania współczynników kształtu

GLSL

```
1: #version 450 core
2:
3: #define RESTART_IND 0xFFFFFFFF
4:
5: layout(local_size_x=1) in;
6:
7: layout(std430,binding=7) buffer FFPixBuf { uint pix[]; } pxbuf;
8:
9: layout(binding=0,r32ui) uniform uimage2D fftxt;
10: uniform CtlBlock { .... } ctl;  /* listing 29.5 */
11:
12: void main ( void )
13: {
14:   ivec2 xy;
15:   uint z, m;
16:
17:   xy = ivec2 ( gl_GlobalInvocationID.xy );
18:   z = uint ( xy.y*gl_NumWorkGroups.x + xy.x );
19:   m = imageLoad ( fftxt, xy ).x;
20:   pxbuf.pix[ctl.p0+z] = m != RESTART_IND ?
21:           m : imageLoad ( fftxt, xy + ivec2(0,gl_NumWorkGroups.y) ).x;
22: } /*main*/
```

Odczytany numer jest zapamiętywany w tablicy w buforze utworzonym przez procedurę `ComputeFormFactors` w linii 13 na listingu 29.18. Indeks do tablicy jest sumą obliczonego przez szader w linii 19 numeru piksela i wartości zmiennej `ctl.p0`, która jest iloczynem numeru elementu dyskretyzacji w bloku i liczby $N = 3888$ (zobacz listing 29.23, linia 22).

Pokazana na listingu 29.28 procedura `LoadFFVPMatrices` w linii 27 wywołuje procedurę, która oblicza macierz przejścia od układu świata do układu obserwatora, a następnie mnoży tę macierz przez 10 macierzy przekształceń perspektywicznych skonstruowanych przez procedurę `ConstructFFPMatrices` i przesyła iloczyny do tablicy `vpm` w bloku zmiennych jednolitych `FFTransBlock` (listingi 29.24 i 29.25).

Listing 29.28. Procedury `M4x4ViewPVf` i `LoadFFVPMatrices`

C

```
1: void M4x4ViewPVf ( GLfloat vm[16], float p[3], float v[3] )
2: {
3:   float w1[3], w2[3], gamma, t;
4:
5:   memset ( vm, 0, 16*sizeof(GLfloat) );
6:   gamma = sqrt ( V3DotProductf ( v, v ) );
7:   memcpy ( w1, v, 3*sizeof(float) );
8:   w1[2] += v[2] > 0.0 ? gamma : -gamma;
9:   gamma = 2.0 / V3DotProductf ( w1, w1 );
10:  t = gamma * V3DotProductf ( w1, p );
11:  vm[12] = t*w1[0]-p[0];  vm[13] = t*w1[1]-p[1];  vm[14] = t*w1[2]-p[2];
12:  w2[0] = gamma*w1[0];  w2[1] = gamma*w1[1];  w2[2] = gamma*w1[2];
13:  vm[0] = 1.0-w1[0]*w2[0];  vm[5] = 1.0-w1[1]*w2[1];
14:  vm[10] = 1.0-w1[2]*w2[2];  vm[1] = vm[4] = -w1[1]*w2[0];
15:  vm[2] = vm[8] = -w1[2]*w2[0];  vm[6] = vm[9] = -w1[1]*w2[2];
16:  vm[15] = 1.0;
17:  if ( v[2] <= 0.0 ) {
18:    vm[2] = -vm[2];  vm[6] = -vm[6];  vm[10] = -vm[10];  vm[14] = -vm[14];
19:  }
20: } /*M4x4ViewPVf*/
21:
22: void LoadFFVPMatrices ( FFTransBl *fftrans, GLfloat cp[3], GLfloat nv[3] )
23: {
24:   GLfloat vm[16];
25:   int     i;
26:
27:   M4x4ViewPVf ( vm, cp, nv );
28:   for ( i = 0; i < 10; i++ )
29:     M4x4Multf ( fftrans->vpm[i], fftrans->pm[i], vm );
30:   glBindBufferBase ( GL_UNIFORM_BUFFER, fftrbbp, fftrans->fftrbuf );
31:   glBufferSubData ( GL_UNIFORM_BUFFER, fftrbofs[1],
32:                     10*16*sizeof(GLfloat), fftrans->vpm );
33:   ExitIfGLError ( "LoadFFVPMatrices" );
34: } /*LoadFFVPMatrices*/
```

Macierze przejścia do układu obserwatora mogłyby być konstruowane przez opisaną w podrozdziale A.2 procedurę `M4x4LookAtf`, ale za rozwiązanie wygodniejsze uznałem użycie procedury `M4x4ViewPVf` (niewymagającej podawania wektora $\boldsymbol{u}$ określającego kierunek „do góry"). Konstruowana przez nią macierz opisuje złożenie przesunięcia umieszczającego początek układu w punkcie określonym przez drugi parametr z odbiciem Householdera przeprowadzającym wersor osi z na kierunek wektora podanego jako trzeci parametr[26].

Przedstawiona na listingu 29.29 procedura, wywoływana po wykonaniu obrazów dla wszystkich elementów w bloku, za pomocą szaderów obliczeniowych z listingów 29.30 i 29.31 konstruuje blok macierzy G. Pierwszy z tych szaderów pracuje w lokalnych grupach roboczych o wymiarach $972 \times 1 \times 1$. Zadaniem każdej takiej grupy roboczej jest przetworzenie danych uzyskanych z obrazów wykonanych dla jednego elementu dyskretyzacji, z czego powstaje jeden wiersz macierzy G. Liczba lokalnych grup roboczych jest liczbą wierszy bloku, tj. 1024 dla wszystkich bloków z wyjątkiem ostatniego, który może mieć ich mniej.

Listing 29.29. Procedura `GPUComputeFF`

C

```
static void GPUComputeFF ( BalanceElements *belem, GLuint ffbuf[4],
                           int nseq, GPUSparseMatrix *bldesc )
{
  unsigned int k, m, d, nnz, N;

  N = 3*FFTXTSIZE*FFTXTSIZE;
  glUseProgram ( bprog_id[9] );   /* listing 29.30 */
  COMPUTE ( nseq, 1, 1 )
  glUseProgram ( bprog_id[10] );  /* listing 29.31 */
  SetCTLUniformi ( belem, CTL_STAGE, 0 );
  COMPUTE ( nseq+1, 1, 1 );
  if ( nseq > 1 ) {
    SetCTLUniformi ( belem, CTL_STAGE, 1 );
    SetCTLUniformi ( belem, CTL_H, (GLuint)nseq );
    d = nseq/2;
    for ( k = 0, m = nseq;  m > 0;  k++, m >>= 1 ) {
      SetCTLUniformi ( belem, CTL_STEP, (GLint)k );
      COMPUTE ( d, 1, 1 );
    }
  }
  glGetNamedBufferSubData ( ffbuf[3], nseq*sizeof(GLuint),
                            sizeof(GLuint), &nnz );
  bldesc->nnz = nnz;
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, bldesc->buf[0] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER, (nseq+1+nnz)*sizeof(GLuint),
                 NULL, GL_DYNAMIC_DRAW );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 2, bldesc->buf[1] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER, nnz*sizeof(GLfloat),
```

[26] W tym przypadku nie ma znaczenia, że odbicie symetryczne zmienia orientację, ważne, że jest izometrią.

```
29:                   NULL, GL_DYNAMIC_DRAW );
30:   glCopyNamedBufferSubData ( ffbuf[3], bldesc->buf[0], 0, 0,
31:                               (nseq+1)*sizeof(GLuint) );
32:   SetCTLUniformi ( belem, CTL_STAGE, 2 );
33:   COMPUTE ( nseq+1, N, 1 );
34:   SetCTLUniformi ( belem, CTL_STAGE, 3 );
35:   SetCTLUniformi ( belem, CTL_N, N*nseq );
36:   COMPUTE ( nnz, 1, 1 );
37:   ExitIfGLError ( "GPUComputeFF" );
38: } /*GPUComputeFF*/
```

Obliczenie wykonywane przez szader z listingu 29.30 jest wieloetapowe, ale poszczególne etapy są realizowane po *jednym* wywołaniu programu. Poszczególne wątki w lokalnej grupie roboczej współpracują ze sobą, korzystając z pamięci podręcznej GPU, w której znajdują się zadeklarowane w liniach 16 i 17 tablice `cache` i `fcache`. Tablice te zajmują razem $8N = 31104$ bajtów, a więc mieszczą się w limicie 32 kB określonym w specyfikacji OpenGL-a. Z kolei liczba wątków lokalnej grupy roboczej, 972, nie przekracza limitu podanego w specyfikacji i jest dzielnikiem połowy liczby pikseli przetwarzanych obrazów.

W pętli w liniach 37–39 do pamięci podręcznej są kopiowane dane: numery makroelementów zapisanych w pikselach i wagi pikseli. Procedura `barrier`, wywoływana po tym i każdym następnym etapie, służy do synchronizacji obliczeń — powrót z niej następuje, gdy wszystkie wątki lokalnej grupy roboczej zakończyły wykonywanie instrukcji poprzedzających. Zatem, wątki zakończyły czytanie danych do pamięci podręcznej i teraz każdy wątek ma dostęp do wszystkich tych danych.

Kolejnym etapem jest sortowanie danych (linie 41–58). Identyfikatory makroelementów mają być ustawione w kolejności niemalejącej, przy czym każde przestawienie liczb w tablicy `cache` wiąże się z przestawieniem wag pikseli w tablicy `fcache` (robi to procedura `CompSwap`, linie 19–28). Szader realizuje algorytm opisany w podrozdziale G.3. Sortując ciąg o długości N, można jednocześnie porównywać i przestawiać $N/2$ elementów, ale ponieważ grupa robocza liczy 972 wątki, każdy wątek wykonuje kolejno do trzech porównań[27]. Liczba etapów sortowania jest równa $\lceil \log_2 N \rceil = 12$; i-ty etap (licząc od 1) składa się z i kroków. Po każdym kroku następuje wywołanie procedury `barrier` (w liniach 49 i 56).

Po posortowaniu numery makroelementów i wagi pikseli są (w liniach 59–60) przepisywane z pamięci podręcznej do tablic, z których zostały odczytane. Pętla w liniach 62–69 do tablicy `cache` wpisuje ciąg zer i jedynek (znów, ponieważ wątków jest tylko 972, każdy z nich wpisuje dane dla czterech pikseli). Zera odpowiadają pikselom tła (w których zamiast numeru makroelementu widnieje liczba `0xFFFFFFFF`) oraz tym pikselom, których numer makroelementu jest taki sam jak numer w pikselu poprzednim. W szczególności pierwszemu elementowi tablicy jest (w linii 66) przypisywana wartość 1.

[27] Liczba 3 wzięła się stąd, że ciąg o długości N jest sortowany tak, jak ciąg o długości będącej najmniejszą całkowitą potęgą liczby 2 nie mniejszą niż N — w tym przypadku $2^{12} = 4096$ — z pominięciem przestawiania elementów za końcem danego ciągu. Zatem 972 wątki wykonują podczas sortowania zadania przewidziane dla 2048 wykonawców, z których wielu próżnuje; to nie szkodzi.

Listing 29.30. Szader sortowania wag

GLSL

```
#version 450 core

#define RESTART_IND 0xFFFFFFFF
#define N     3888  /* 3*36*36 */
#define STEPS   12  /* ceil log2 N */
#define PCS      4
#define SPCS     3
#define SIZEX  972  /* N/PCS */

layout(local_size_x=SIZEX) in;

layout(std430,binding=3) buffer FFWeightBuf { float w[]; } wbuf;
layout(std430,binding=4) buffer FFOnesBuf { uint k[]; } obuf;
layout(std430,binding=7) buffer FFPixBuf { uint pix[]; } pxbuf;

shared uint  cache[N];
shared float fcache[N];

void CompSwap ( uint i, uint j )
{
  uint  c;
  float f;

  if ( cache[i] > cache[j] ) {
    c = cache[i];   cache[i] = cache[j];    cache[j] = c;
    f = fcache[i];  fcache[i] = fcache[j];  fcache[j] = f;
  }
} /*CompSwap*/

void main ( void )
{
  uint x, y, iid[SPCS], id[SPCS], s, i, j, h, h2, h4, k, l,
       ii[PCS/2], m0, m1;

  x = gl_LocalInvocationID.x;
  y = N*gl_WorkGroupID.x + x;
  for ( k = 0, i = x, j = y;  k < PCS;  k++, i += SIZEX, j += SIZEX )
    fcache[i] = (cache[i] = pxbuf.pix[j]) != RESTART_IND ?
                wbuf.w[i] : 0.0;
  barrier ();
  for ( iid[0] = x, k = 1;  k < SPCS;  k++ )
    iid[k] = iid[k-1]+SIZEX;
  for ( s = 0, h = 2, h2 = 1;  s < STEPS;  s++, h2 = h, h += h ) {
    for ( k = 0; k < SPCS; k++ ) {
      l = iid[k] % h2;  id[k] = iid[k] / h2;  i = id[k]*h + l;
```

```
46:       if ( (j = (id[k]+1)*h-l-1) < N )
47:         CompSwap ( i, j );
48:     }
49:     barrier ();
50:     for ( h4 = h2 / 2;  h2 > 1;  h2 = h4, h4 /= 2 ) {
51:       for ( k = 0; k < SPCS; k++ ) {
52:         l = iid[k] % h4;  id[k] = iid[k] / h4;  i = id[k]*h2 + l;
53:         if ( (j = i+h4) < N )
54:           CompSwap ( i, j );
55:       }
56:       barrier ();
57:     }
58:   }
59:   for ( k = 0, i = x, j = y;  k < PCS;  k++, i += SIZEX, j += SIZEX )
60:     { pxbuf.pix[j] = cache[i];  wbuf.w[j+N] = fcache[i]; }
61:   barrier ();
62:   for ( k = 0, i = x, j = y;  k < PCS;  k++, i += SIZEX, j += SIZEX ) {
63:     if ( pxbuf.pix[j] == RESTART_IND )
64:       cache[i] = 0;
65:     else if ( i == 0 )
66:       cache[i] = 1;
67:     else
68:       cache[i] = int ( pxbuf.pix[j] > pxbuf.pix[j-1] );
69:   }
70:   barrier ();
71:   ii[0] = x+x;
72:   for ( k = 1; k < PCS/2; k++ )
73:     ii[k] = ii[k-1] + (SIZEX+SIZEX);
74:   for ( m0 = 0x01, m1 = 0;  m0 < N;  m1 = (m0 += m0)-1 ) {
75:     for ( k = 0; k < PCS/2; k++ ) {
76:       i = (ii[k] &  m0) | m1;
77:       if ( (j = i + (iid[k] & m1) + 1) < N )
78:         cache[j] += cache[i];
79:     }
80:     barrier ();
81:   }
82:   for ( k = 0, i = x, j = y;  k < PCS;  k++, i += SIZEX, j += SIZEX )
83:     obuf.k[j] = cache[i];
84: } /*main*/
```

Instrukcje w liniach 71–81 dla ciągu zer i jedynek w tablicy `cache` obliczają ciąg sum prefiksowych (podrozdz. G.2). Ostatni element tego ciągu (tj. liczba wpisanych wcześniej jedynek) jest liczbą niezerowych współczynników w wierszu macierzy G. Sumy prefiksowe są w liniach 82–83 przepisywane do tablicy w bloku `FFOnesBuf` (bufor z tym blokiem został utworzony przez procedurę `ComputeFormFactors`, listing 29.18, linia 14). Dane w tym bloku będą dalej przetwarzane przy użyciu szadera z listingu 29.31.

Listing 29.31. Szader konstrukcji bloku macierzy współczynników kształtu

GLSL

```
#version 450 core

#define FFTXTSIZE          36
#define NPIX             3888  /* 3*36*36 */

layout(local_size_x=1) in;

layout(std430,binding=1) buffer FFBlockRC { uint rc[]; } rc;
layout(std430,binding=2) buffer FFBlockA { float a[]; } a;
layout(std430,binding=3) buffer FFWeightBuf { float w[]; } wbuf;
layout(std430,binding=4) buffer FFOnesBuf { uint k[]; } obuf;
layout(std430,binding=5) buffer FFCSBuf { uint s[]; } csbuf;
layout(std430,binding=7) buffer FFPixBuf { uint pix[]; } pxbuf;

uniform CtlBlock { .... } ctl;  /* listing 29.5 */

void PrefixSum ( uint i )
{
  uint ii, m0, m1, ia, ib;

  ii = i+i;  m0 = 0x01 << ctl.step;  m1 = m0-1;
  ia = (ii &  m0) | m1;
  if ( (ib = ia + (i & m1) + 1) < ctl.H )
    csbuf.s[ib+1] += csbuf.s[ia+1];
} /*PrefixSum*/

uint BinSearch ( uint i )
{
  uint j, k, l;

  if ( obuf.k[0]-1 >= i )
    return 0;
  else {
    for ( j = 0, k = ctl.N;  k-j > 1; ) {
      l = j + (k-j)/2;
      if ( obuf.k[l]-1 >= i ) k = l; else j = l;
    }
    return k;
  }
} /*BinSearch*/

void main ( void )
{
  uvec3 inv;
  uint  i, n0;
```

```
46:   float s;
47:
48:   inv = gl_GlobalInvocationID;
49:   switch ( ctl.stage ) {
50: case 0:
51:     csbuf.s[inv.x] = inv.x > 0 ? obuf.k[inv.x*NPIX-1] : 0;
52:     return;
53: case 1:
54:     PrefixSum ( inv.x );
55:     return;
56: case 2:
57:     obuf.k[inv.x*NPIX+inv.y] += rc.rc[inv.x];
58:     return;
59: case 3:
60:     n0 = BinSearch ( inv.x );
61:     if ( obuf.k[n0]-1 == inv.x ) {
62:       s = wbuf.w[NPIX+n0];
63:       for ( i = n0+1;  i % NPIX > 0 && obuf.k[i]-1 == inv.x;  i++ )
64:         s += wbuf.w[NPIX+i];
65:       a.a[inv.x] = s;
66:       rc.rc[ctl.H+1+inv.x] = pxbuf.pix[n0];
67:     }
68:     return;
69: default:
70:     return;
71:   }
72: } /*main*/
```

Etap 0 obliczeń przy użyciu szadera z listingu 29.31, uruchamiany przez instrukcję procedury `GPUComputeFF` (listing 29.29, linia 11), do tablicy w bloku `FFCSBuf` wpisuje liczby niezerowych współczynników w kolejnych wierszach konstruowanego bloku macierzy G. Etap 1 (linie 13–19 na listingu 29.29 i 56 na listingu 29.31) oblicza ciąg sum prefiksowych ciągu tych liczb. Jest on potrzebny, aby utworzyć tablicę c reprezentacji bloku macierzy i w szczególności aby otrzymać całkowitą liczbę niezerowych współczynników w bloku. Liczbę tę procedura `GPUComputeFF` odczytuje z bufora w liniach 21–22. Następnie, w liniach 24–29 procedura ta dokonuje rezerwacji miejsca w pamięci GPU na bufory, w których będzie przechowywana reprezentacja bloku (wcześniej były zarezerwowane tylko identyfikatory tych buforów). W liniach 30–31 sumy prefiksowe są kopiowane z bloku `FFCSBuf` do tablicy `r` reprezentacji bloku.

W etapie 2 liczba wątków jest o 1 większa niż wielkość bloku (1024, lub mniej dla ostatniego bloku). Początkowa wartość elementu tablicy `obuf.k` jest elementem ciągu sum prefiksowych zer i jedynek obliczonego przez szader sortowania wag z listingu 29.31. Jest to numer niezerowego współczynnika *w wierszu* macierzy G, któremu odpowiada piksel o wadze znajdującej się na analogicznym miejscu w tablicy `wbuf.w`. Po dodaniu do niego elementu tablicy `r` powstaje numer niezerowego współczynnika *w bloku* macierzy G.

Etap 3 ma za zadanie zsumować wagi pikseli i wpisać sumy (współczynniki kształtu) do tablicy a stanowiącej część reprezentacji bloku macierzy G, a także wypełnić tablicę c, w której mają się znaleźć numery kolumn, w których występują niezerowe współczynniki. Procedura `BinSearch` metodą wyszukiwania binarnego znajduje początek podciągu wag do zsumowania, którego suma jest k-tym niezerowym współczynnikiem w bloku (liczba k jest numerem instancji szadera).

Procedura `AssembleFFMatrix` ma za zadanie utworzyć reprezentację *całej* macierzy G, po skonstruowaniu wszystkich bloków. Jej drugi parametr jest liczbą bloków, a trzeci tablicą opakowań macierzy rzadkich, które są tymi blokami. Procedura korzysta z szadera pokazanego na listingu 29.33. Po obliczeniu (w liniach 9–10) sumy liczb niezerowych współczynników w blokach następuje rezerwacja buforów, w których będą przechowywane tablice z reprezentacją macierzy G.

Listing 29.32. Procedura `AssembleFFMatrix`

C

```
1: static void AssembleFFMatrix ( BalanceElements *belem,
2:                                int nblocks, GPUSparseMatrix *ffbldesc )
3: {
4:   int    i;
5:   GLuint m, nnz;
6:
7:   glUseProgram ( bprog_id[11] );  /* listing 29.33 */
8:   for ( i = 0, m = nnz = 0;  i < nblocks;  i++ )
9:     { m += ffbldesc[i].m;  nnz += ffbldesc[i].nnz; }
10:  belem->ffmat.m = m;  belem->ffmat.n = belem->nmacroelem;
11:  belem->ffmat.nnz = nnz;  belem->ffmat.lmax = 0;
12:  glGenBuffers ( 2, belem->ffmat.buf );
13:  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 3, belem->ffmat.buf[0] );
14:  glBufferData ( GL_SHADER_STORAGE_BUFFER, (m+1+nnz)*sizeof(GLuint),
15:                NULL, GL_DYNAMIC_DRAW  );
16:  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 4, belem->ffmat.buf[1] );
17:  glBufferData ( GL_SHADER_STORAGE_BUFFER, nnz*sizeof(GLfloat),
18:                NULL, GL_DYNAMIC_DRAW );
19:  SetCTLUniformi ( belem, CTL_STAGE, 0 );
20:  for ( i = 0, m = nnz = 0;  i < nblocks;  i++ ) {
21:    glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, ffbldesc[i].buf[0] );
22:    SetCTLUniformi ( belem, CTL_H, m );
23:    SetCTLUniformui ( belem, CTL_NNZ, nnz );
24:    COMPUTE ( ffbldesc[i].m+1, 1, 1 );
25:    m += ffbldesc[i].m;  nnz += ffbldesc[i].nnz;
26:  }
27:  SetCTLUniformi ( belem, CTL_STAGE, 1 );
28:  SetCTLUniformui ( belem, CTL_H, m );
29:  for ( i = 0, nnz = 0;  i < nblocks;  i++ ) {
30:    glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, ffbldesc[i].buf[0] );
31:    glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 2, ffbldesc[i].buf[1] );
```

```
32:     SetCTLUniformui ( belem, CTL_NNZ, nnz );
33:     SetCTLUniformui ( belem, CTL_MI, ffbldesc[i].m );
34:     COMPUTE ( ffbldesc[i].nnz, 1, 1 );
35:     nnz += ffbldesc[i].nnz;
36:   }
37:   ExitIfGLError ( "AssembleFFMatrix" );
38: } /*AssembleFFMatrix*/
```

Etap 0 obliczeń, wykonywany w pętli dla kolejnych bloków, przepisuje informacje z tablic r reprezentacji bloku, dodając do każdej liczby sumę liczb niezerowych współczynników we wcześniej przetworzonych blokach. W etapie 1 następuje przepisywanie niezerowych współczynników do tablicy a i przepisywanie numerów kolumn z tymi współczynnikami do tablicy c.

Listing 29.33. Szader łączenia bloków macierzy G

GLSL

```
1: #version 450 core
2:
3: layout(local_size_x=1) in;
4:
5: layout(std430,binding=1) buffer FFBlockRCi{ uint rc[]; } rci;
6: layout(std430,binding=2) buffer FFBlockAi { float a[]; } ai;
7: layout(std430,binding=3) buffer FFBlockRC { uint rc[]; } rc;
8: layout(std430,binding=4) buffer FFBlockA  { float a[]; } a;
9:
10: uniform CtlBlock { .... } ctl;  /* listing 29.5 */
11:
12: void main ( void )
13: {
14:   uint i;
15:
16:   i = gl_GlobalInvocationID.x;
17:   switch ( ctl.stage ) {
18: case 0:
19:     rc.rc[ctl.H+i] = rci.rc[i] + ctl.nnz;
20:     return;
21: case 1:
22:     rc.rc[ctl.H+1+ctl.nnz+i] = rci.rc[ctl.mi+1+i];
23:     a.a[ctl.nnz+i] = ai.a[i];
24:     return;
25: default:
26:     return;
27:   }
28: } /*main*/
```

29.2.7. Obliczanie tekstury irradiancji

Listing 29.34 przedstawia procedurę obliczającą teksturę irradiancji. Jej parametrami są: wskaźnik opakowania danych z gotowymi wynikami preprocesingu opisanego wcześniej w tym rozdziale, opakowanie danych używanych do rzutowania obiektów i liczba T iteracji metody rozwiązywania układu równań (29.3).

Listing 29.34. Procedura `ComputeLightBalance`

C

```
1: char ComputeLightBalance ( BalanceElements *belem, TransBl *trans,
2:                            int niter )
3: {
4:   int i, nelem;
5:
6:   nelem = belem->nelem;
7:   ComputeElemEmission ( belem, trans );
8:   glCopyNamedBufferSubData ( BUF_LE, BUF_L0, 0, 0,
9:                              4*nelem*sizeof(GLfloat) );
10:  for ( i = 0; i < niter; i++ ) {
11:    GPUSMultSparseMatrixVectorf ( BUF_L1, &belem->avgmat, 4, BUF_L0 );
12:    GPUSMultSparseMatrixVectorf ( BUF_L0, &belem->ffmat, 4, BUF_L1 );
13:    FinishBalanceIteration ( belem, BUF_LE, BUF_L0 );
14:  }
15:  GPUSMultSparseMatrixVectorf ( BUF_L1, &belem->avgmat, 4, BUF_L0 );
16:  GPUSMultSparseMatrixVectorf ( BUF_L0, &belem->ffmat, 4, BUF_L1 );
17:  SetupEnvIrradianceTexture ( belem, BUF_L0 );
18:  glUseProgram ( 0 );
19:  ExitIfGLError ( "ComputeLightBalance" );
20:  return true;
21: } /*ComputeLightBalance*/
```

Wywołana w linii 7 procedura oblicza radiancję światła emitowanego przez poszczególne elementy dyskretyzacji, czyli oblicza wektor $\boldsymbol{L}_e$. Jest on zapisany w buforze, który nazwałem `BUF_LE` (bufor ten utworzyła procedura `EndEnteringBalanceElements` razem z potrzebnymi teraz buforami `BUF_L0` i `BUF_L1`), zawierającym tablicę elementów typu `vec4`; pola `r`, `g`, `b` tych elementów reprezentują składowe kolorowe emitowanego światła.

W liniach 8–9 wektor $\boldsymbol{L}_e$ jest kopiowany do bufora `BUF_L0`, któremu w ten sposób jest przypisywany wektor $\boldsymbol{L}_0$. Następnie w pętli są wykonywane iteracje; procedura wywołana w linii 11 oblicza iloczyn $A\boldsymbol{L}_i$ (zapamiętując go w buforze `BUF_L1`), a wynikiem jej kolejnego wywołania jest wektor $GA\boldsymbol{L}_i$ w buforze `BUF_L0`. Procedura `FinishBalanceIteration` (listing 29.40) mnoży ten wektor przez macierz D i dodaje wektor $\boldsymbol{L}_e$, czego wynikiem jest wektor $\boldsymbol{L}_{i+1}$.

Po zakończeniu pętli wektor $\boldsymbol{L}_T$ jest jeszcze mnożony przez macierze A i G. Procedura `SetupEnvIrradianceTexture` (listing 29.42) na podstawie tego iloczynu oblicza wartości tekseli tekstury irradiancji, która odtąd jest gotowa do wykonywania obrazów.

Pierwszym krokiem obliczenia radiancji światła emitowanego przez elementy jest narysowanie trójkątów na pomocniczej teksturze (w pozaekranowym buforze ramki). Tekstura ta ma rozdzielczość większą od tekstury irradiancji o czynnik `VTXT_MAG` (listing 29.35) — jest to więc antyaliasing przez nadpróbkowanie. Procedura `ComputeElemEmission` w liniach 12–23 przygotowuje bufor ramki z załącznikiem, wybiera opisany dalej program szaderów, a następnie w pętli w liniach 29–39 rysuje trójkąty kolejnych obiektów. Następnie likwiduje bufor ramki i przy użyciu programu z szaderem z listingu 29.39 oblicza wektor $\boldsymbol{L}_e$.

Listing 29.35. Procedura `ComputeElemEmission`

C

```
1: #define VTXT_MAG 4
2:
3: static void ComputeElemEmission ( BalanceElements *belem, TransBl *trans )
4: {
5:   GLuint        efbo, etxt;
6:   GLsizei       w, h;
7:   BalanceObject *bobj;
8:   int           i;
9:
10:  w = VTXT_MAG*belem->irrtxt_width;
11:  h = VTXT_MAG*belem->irrtxt_height;
12:  glGenTextures ( 1, &etxt );
13:  glActiveTexture ( GL_TEXTURE0+49 );
14:  glBindTexture ( GL_TEXTURE_2D, etxt );
15:  glTexStorage2D ( GL_TEXTURE_2D, 1, GL_RGBA32F, w, h );
16:  glBindTexture ( GL_TEXTURE_2D, 0 );
17:  glGenFramebuffers ( 1, &efbo );
18:  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, efbo );
19:  glFramebufferTexture ( GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
20:                          etxt, 0 );
21:  if ( glCheckFramebufferStatus ( GL_DRAW_FRAMEBUFFER ) !=
22:       GL_FRAMEBUFFER_COMPLETE )
23:    ExitOnError ( "ComputeElemEmission" );
24:  glViewport ( 0, 0, w, h );
25:  glClearColor ( 0.0, 0.0, 0.0, 0.0 );
26:  glClear ( GL_COLOR_BUFFER_BIT );
27:  glUseProgram ( bprog_id[12] );  /* listingi 29.36, 29.37 i 29.38 */
28:  glBindVertexArray ( belem->tvao );
29:  for ( i = 0; i < belem->nobj; i++ ) {
30:    bobj = &belem->objtab[i];
31:    LoadMMatrix ( trans, bobj->obj->mm );
32:    glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
33:    SetCTLUniformi ( belem, CTL_FIRST, bobj->ftrdesc );
34:    glDrawElements ( GL_TRIANGLES, 3*bobj->cntr, GL_UNSIGNED_INT,
35:                     (GLvoid*)(bobj->ftrdesc*3*sizeof(GLuint)) );
36:    glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
37:    glDrawElements ( GL_TRIANGLES, 3*bobj->cntr, GL_UNSIGNED_INT,
```

```
                        (GLvoid*)(bobj->ftrdesc*3*sizeof(GLuint)) );
  }
  glFinish ();
  glBindVertexArray ( 0 );
  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
  glDeleteFramebuffers ( 1, &efbo );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, BUF_VARBUF );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, BUF_LE );
  glBindImageTexture ( 0, etxt, 0, GL_FALSE, 0, GL_READ_ONLY, GL_RGBA32F );
  glUseProgram ( bprog_id[13] );  /* listing 29.39 */
  COMPUTE ( belem->nelem, 1, 1 );
  glDeleteTextures ( 1, &etxt );
  ExitIfGLError ( "ComputeElemEmission" );
} /*ComputeElemEmission*/
```

Atrybuty wierzchołka na wejściu szadera z listingu 29.36 to wektor współrzędnych jednorodnych położenia w układzie modelu, wektor współrzędnych tekstury (nałożonej na obiekt) i wektor współrzędnych tekstury irradiancji. W zmiennych `Out.WPosition` i `Out.TexCoord` szader przekazuje na wyjście położenie wierzchołka w układzie świata i wektor współrzędnych tekstury. Wartość przypisywana zmiennej `gl_Position` jest wektorem współrzędnych tekstury irradiancji, przekształconych tak, aby dziedzinę tekstury irradiancji (prostokąt o wymiarach $w \times h$) odwzorować na ścianę kostki standardowej (tj. kwadrat $[-1,1] \times [-1,1]$).

Listing 29.36. Szader wierzchołków programu obliczania emisji

GLSL

```
#version 450 core

layout(location=0) in vec4 in_MPosition;
layout(location=1) in vec2 TxtCoord;
layout(location=2) in vec2 in_VPosition;

out GVertex { vec3 WPosition;  vec2 TexCoord; } Out;

uniform TransBlock { .... } trb;  /* listing 27.5 */
uniform CtlBlock { .... } ctl;  /* listing 29.5 */

void main ( void )
{
  Out.WPosition = (trb.mm * in_MPosition).xyz;
  Out.TexCoord = TxtCoord;
  gl_Position = vec4 ( 2.0*in_VPosition.x/ctl.width-1.0,
                       2.0*in_VPosition.y/ctl.height-1.0, 0.0, 1.0 );
} /*main*/
```

Zadaniem szadera geometrii (listing 29.37) jest obliczenie i dołożenie do atrybutów wierzchołków przekazywanych dalej wektorów normalnych wierzchołków (które tu są, ale

mogą nie być wektorem normalnym trójkąta, zobacz s. 286–287) i wektora normalnego trójkąta. Obraz powstanie w dziedzinie tekstury irradiancji, ale szader fragmentów musi obliczyć oświetlenie przez punktowe źródła światła w przestrzeni. Wartość przypisana zmiennej `gl_PrimitiveID` jest numerem trójkąta w rysowanym fragmencie tablic. Szader fragmentów, na jej podstawie, określi materiał rysowanego trójkąta.

Listing 29.37. Szader geometrii programu obliczania emisji

GLSL

```
1: #version 450 core
2:
3: layout(triangles) in;
4: layout(triangle_strip,max_vertices=3) out;
5:
6: in GVertex { vec3 WPosition;  vec2 TexCoord; } In[];
7:
8: out FVertex {
9:     vec3      WPosition, Normal;
10:    flat vec3 TNormal;
11:    vec2      TxtCoord;
12:  } Out;
13:
14: void main ( void )
15: {
16:   int  i;
17:   vec3 v1, v2, nv;
18:
19:   v1 = In[1].WPosition - In[0].WPosition;
20:   v2 = In[2].WPosition - In[0].WPosition;
21:   nv = normalize ( cross ( v1, v2 ) );
22:   for ( i = 0; i < 3; i++ ) {
23:     gl_Position = gl_in[i].gl_Position;
24:     gl_PrimitiveID = gl_PrimitiveIDIn;
25:     Out.WPosition = In[i].WPosition;
26:     Out.TxtCoord = In[i].TexCoord;
27:     Out.Normal = Out.TNormal = nv;
28:     EmitVertex ();
29:   }
30:   EndPrimitive ();
31: } /*main*/
```

Procedura `ComputeElemEmission` przed wydaniem polecenia narysowania trójkątów obiektu (w linii 31) przesyła do bloku `TransBlock` odpowiednią macierz przekształcenia modelu (aby szader wierzchołków użył jej w obliczeniu położenia wierzchołka w układzie świata) i nadaje (w linii 33) zmiennej jednolitej `ctl.first` numer pierwszego trójkąta obiektu. Szader fragmentów z listingu 29.38 w linii 40 dodaje ten numer do numeru trójkąta w rysowanym fragmencie tablic, aby obliczyć numer trójkąta w tablicach, po czym z bloku `MatBuf`

Listing 29.38. Szader fragmentów programu obliczania emisji

GLSL

```
#version 450 core

#define MAX_TEXTURES   4
#define MAX_MATERIALS 20
#define MAX_NLIGHTS    8

in FVertex {
    vec3      WPosition, Normal;
    flat vec3 TNormal;
    vec2      TxtCoord;
  } In;
out vec4 out_Colour;

layout(std430,binding=11) buffer MatBuf { int matnum[]; } mbuf;
layout(binding=0) uniform sampler2D tex[MAX_TEXTURES];
layout(binding=MAX_TEXTURES) uniform sampler2D shtex[MAX_NLIGHTS];
layout(binding=MAX_TEXTURES) uniform samplerCube cshtex[MAX_NLIGHTS];

uniform CtlBlock { .... } ctl;    /* listing 29.5 */
uniform TransBlock { .... } trb;  /* listing 27.5 */
struct LSPar { .... }             /* listing 22.1 */
uniform LSBlock { .... } light;   /* listing 22.3 */
struct Material { .... };         /* listing 19.12 */
uniform MatBlock { .... } mat;    /* listing 19.12 */

Material mm;

float IsEnlighted ( uint l ) { .... }               /* listing 26.23 */
float CubeDepth ( vec3 v ) { .... }                 /* listing 26.23 */
float IsEnlightedCube ( vec3 pos, uint l ) { .... } /* listing 26.23 */
vec3 posDifference ( vec4 p, vec3 pos, out float dist ) { .... }
float attFactor ( vec3 att, float dist ) { .... }   /* listing 10.4 */
vec3 MatEmission ( float cosnv ) { .... }           /* listing 18.3 */
vec4 LambertLighting ( void ) { .... }      /* listingi 22.5, 26.23 */

void main ( void )
{
  int n;

  n = mbuf.matnum[ctl.first+gl_PrimitiveID];
  mm = mat.mat[n];
  if ( mm.txtnum >= 0 )
    mm.diffref = texture ( tex[mm.txtnum], In.TxtCoord );
  out_Colour = LambertLighting ();
} /*main*/
```

(umieszczonego w buforze BUF_MATBUF) odczytuje numer materiału trójkąta. W linii 41 opis materiału jest przypisywany zmiennej mm. Jeśli opis ten zawiera informację, że własności materiału opisuje tekstura, to w linii 43 polu diffref zmiennej mm jest przypisywana (przefiltrowana przez ewaluator) zdolność odbijania światła wzięta z tekstury.

Procedura LambertLighting oblicza, zgodnie z lambertowskim modelem oświetlenia, sumę radiancji światła emitowanego przez dany punkt i światła odbitego, które dochodzi do tego punktu bezpośrednio z punktowych źródeł światła, z uwzględnieniem cieni. Wartość przypisana zmiennej out_Colour, *bez* korekcji gamma, trafia do obrazu wynikowego.

Zadaniem uruchamianego przez instrukcję w linii 48 procedury ComputeElemEmission szadera pokazanego na listingu 29.39 jest obliczenie wektora $\boldsymbol{L}_\mathrm{e}$ na podstawie tego obrazu, którego rozdzielczość jest czterokrotnie większa niż wymiary w tekselach tekstury irradiancji. Wątek szadera oblicza jeden (wektorowy) współczynnik tego wektora. W linii 17 odczytuje jego indeks, w liniach 18 i 19 z tablicy w buforze BUF_VBUF odczytuje współrzędne elementu dyskretyzacji w dziedzinie tekstury irradiancji, a potem w pętli sumuje wektory emisji zapamiętane w pikselach. Niektóre piksele mają kolor tła, reprezentowany przez wektor zerowy. Kolor średni jest więc obliczany przez podzielenie (w linii 23) obliczonej sumy przez jej składową alfa; piksele „pomalowane" miały nadaną tej składowej wartość 1.

Listing 29.39. Szader obliczeniowy emisji

GLSL

```
1: #version 450 core
2:
3: #define VTXT_MAG 4
4:
5: layout(local_size_x=1) in;
6:
7: layout(std430,binding=0) buffer VarBuf { uvec4 tvar[]; } vbuf;
8: layout(std430,binding=1) buffer LeBuf { vec4 le[]; } lebuf;
9:
10: layout(rgba32f,binding=0) uniform image2D emission;
11:
12: void main ( void )
13: {
14:   uint v, x, y, i, j, k, l;
15:   vec4 s;
16:
17:   v = gl_GlobalInvocationID.x;
18:   x = VTXT_MAG*(vbuf.tvar[v].w & 0xFFFF);
19:   y = VTXT_MAG*(vbuf.tvar[v].w >> 16);
20:   for ( i = 0, k = x, s = vec4(0.0);  i < VTXT_MAG;  i++, k++ )
21:     for ( j = 0, l = y;  j < VTXT_MAG;  j++, l++ )
22:       s += imageLoad ( emission, ivec2 ( k, l ) );
23:   s = vec4 ( s.a > 0.0 ? s.rgb/s.a : vec3(0.0), 1.0 );
24:   lebuf.le[v] = s;
25: } /*main*/
```

Zadaniem procedury FinishBalanceIteration jest wywołanie programu z szaderem z listingu 29.41, który oblicza, dla każdego elementu dyskretyzacji, sumę radiancji światła emitowanego i iloczynu (bieżącego przybliżenia) irradiancji światła oświetlającego element i zdolności elementu do odbijania światła.

Listing 29.40. Procedura FinishBalanceIteration

C

```
static void FinishBalanceIteration ( BalanceElements *belem,
                                     GLuint LeBuf, GLuint LBuf )
{
  glUseProgram ( bprog_id[14] );  /* listing 29.41 */
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, BUF_ALBMAT );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, LeBuf );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 2, LBuf );
  COMPUTE ( belem->nelem, 1, 1 )
  ExitIfGLError ( "FinishBalanceIteration" );
} /*FinishBalanceIteration*/
```

Listing 29.41. Szader dodawania wektorów radiancji

GLSL

```
#version 450 core

layout(local_size_x=1) in;

layout(std430,binding=0) buffer AlbMat { vec4 a[];} am;
layout(std430,binding=1) buffer LeBuf { vec4 l[];} le;
layout(std430,binding=2) buffer Lbuf { vec4 l[];} l;

void main ( void )
{
  uint i;

  i = gl_GlobalInvocationID.x;
  l.l[i] = le.l[i] + am.a[i]*l.l[i];
} /*main*/
```

Procedura SetupEnvIrradianceTexture (listing 29.42) jest wywoływana po wykonaniu przepisanej liczby iteracji. Jej zadaniem jest końcowe przygotowanie tekstury irradiancji do wykonywania końcowych obrazów sceny. Służy do tego szader pokazany na listingu 29.43, realizujący obliczenia w dwóch etapach.

Grupa robocza uruchamiana w obu etapach ma wymiary dziedziny tekstury irradiancji; każdy wątek oblicza wartość jednego teksela. W etapie 0 każdy teksel używany otrzymuje wartość wziętą z bufora BUF_L0, a tekselom nieużywanym jest przypisywany wektor zerowy. Etap 1 dodatkowo przetwarza teksele nieużywane: podczas wykonywania końcowego obrazu ewaluator tekstury *może* sięgać do nich, czego skutkiem byłoby przyjęcie zerowej irradiancji

Listing 29.42. Procedura `SetupEnvIrradianceTexture`

C

```
void SetupEnvIrradianceTexture ( BalanceElements *belem, GLuint lb )
{
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 2, lb );
  glBindImageTexture ( 1, belem->irrtxt, 0, GL_FALSE, 0, GL_READ_WRITE,
                       GL_RGBA32F );
  glUseProgram ( bprog_id[15] );  /* listing 29.43 */
  SetCTLUniformi ( belem, CTL_STAGE, 0 );
  glDispatchCompute ( belem->irrtxt_width, belem->irrtxt_height, 1 );
  glMemoryBarrier ( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
  SetCTLUniformi ( belem, CTL_STAGE, 1 );
  glDispatchCompute ( belem->irrtxt_width, belem->irrtxt_height, 1 );
  glMemoryBarrier ( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
  ExitIfGLError ( "SetupEnvIrradianceTexture" );
} /*SetupEnvIrradianceTexture*/
```

pewnych punktów w pobliżu brzegów trójkątów. Dlatego każdy teksel nieużywany otrzymuje wartość średnią wartości sąsiadujących z nim tekseli używanych. Ponieważ podczas rozmieszczania trójkątów w dziedzinie tekstury irradiancji dookoła każdego płata został dołożony margines o szerokości jednego teksela, teksel nieużywany otrzyma wartość średnią wziętą tylko z tekseli jednego płata. Obliczenie to ma też na celu uzupełnienie „braków" pokrycia trójkątów elementami dyskretyzacji, będących skutkiem nieuniknionych błędów powstających podczas rasteryzacji.

Listing 29.43. Szader obliczania tekstury irradiancji

GLSL

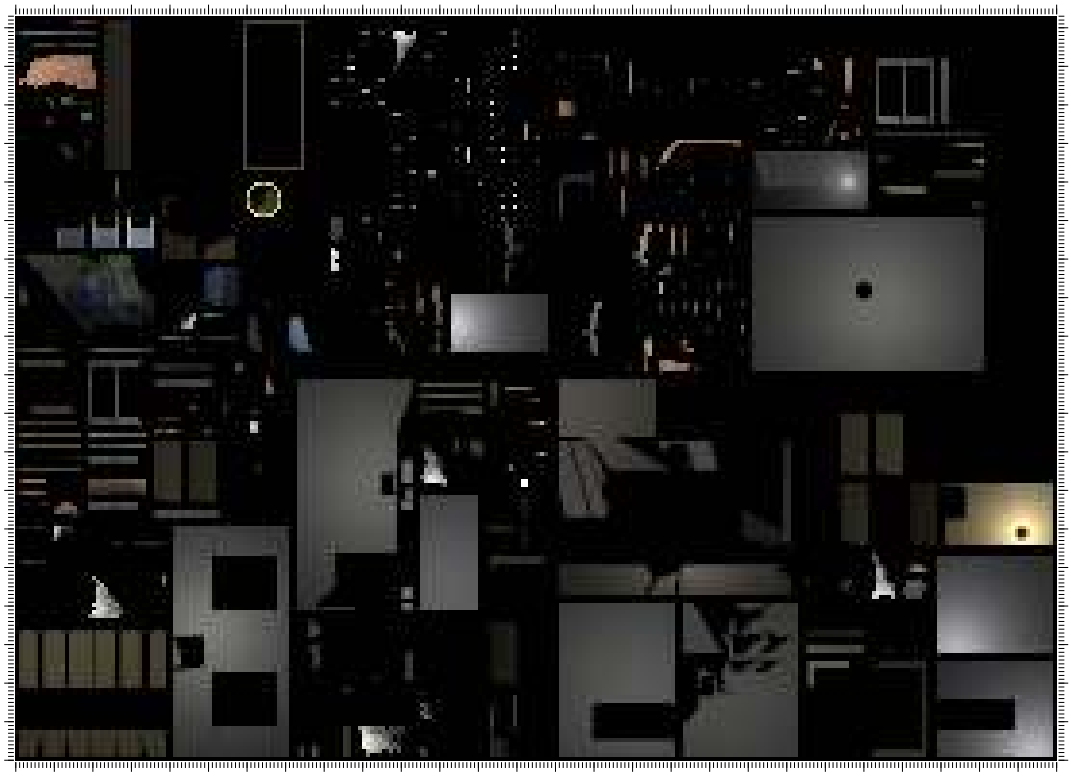
```
#version 450 core

#define RESTART_IND 0xFFFFFFFF

layout(local_size_x=1) in;

layout(std430,binding=2) buffer Lbuf { vec4 l[]; } l;
layout(std430,binding=9) buffer VMap { uvec2 px[]; } vmap;

layout(rgba32f,binding=1) uniform image2D irradtxt;

uniform CtlBlock { .... } ctl;  /* listing 29.5 */

const ivec2 d[8] =
   {ivec2( 1, 0), ivec2( 1, 1), ivec2( 0, 1), ivec2(-1, 1),
    ivec2(-1, 0), ivec2(-1,-1), ivec2( 0,-1), ivec2( 1,-1)};

void main ( void )
{
```

```
20:   ivec2 xy;
21:   uint  z, el;
22:   vec4  tx;
23:   int   i, j;
24:
25:   xy = ivec2 ( gl_GlobalInvocationID.xy );
26:   z = xy.y*ctl.width + xy.x;
27:   el = vmap.px[z].x;
28:   switch ( ctl.stage ) {
29: case 0:
30:     if ( el != RESTART_IND )
31:       imageStore ( irradtxt, xy, l.l[el] );
32:     else
33:       imageStore ( irradtxt, xy, vec4(0.0) );
34:     return;
35: case 1:
36:     if ( el == RESTART_IND ) {
37:       for ( i = j = 0, tx = vec4(0.0);  i < 8;  i++ )
38:         if ( vmap.px[(xy.y+d[i].y)*ctl.width + (xy.x+d[i].x)].x
39:              != RESTART_IND ) {
40:           tx += imageLoad ( irradtxt, xy+d[i] );
41:           j ++;
42:         }
43:       if ( j > 0 )
44:         imageStore ( irradtxt, xy, tx/float(j) );
45:     }
46:     return;
47:   }
48: } /*main*/
```

Rysunek 29.7 przedstawia radiancję światła emitowanego przez poszczególne elementy dyskretyzacji, obliczoną dla przykładowej sceny przez procedurę `ComputeElemEmission` i szadery z listingów 29.36–29.38. Scena jest oświetlona przez trzy punktowe źródła światła: Księżyc (za oknami), lampę w przedpokoju i stojącą na stole świecę. Żaden element „sam" światła nie emituje.

Na rysunku 29.8 jest pokazana gotowa tekstura irradiancji, otrzymana po wykonaniu 10 iteracji metody rozwiązywania układu równań (29.3). Przypomnę, że tekstura ta reprezentuje irradiancję światłem co najmniej raz odbitym od powierzchni, a więc nie uwzględnia ona światła emitowanego przez elementy ani światła dochodzącego do nich bezpośrednio z punktowych źródeł. Światło ze źródeł punktowych zostało „zamienione" na światło emitowane przez elementy dyskretyzacji w celu obliczenia wektora $\boldsymbol{L}_e$ (rys. 29.7) i będzie ponownie obliczone przez szader fragmentów podczas wykonywania końcowego obrazu.

Jeden obiekt w scenie narusza warunki przyjęte na początku rozdziału, co jednak nie spowodowało problemów w obliczeniach: dywan nie jest bryłą, tylko pojedynczym prostokątem (unoszącym się 0.007 jednostek nad podłogą). Jego „niewidoczna" strona jest widoczna od strony podłogi, dlatego irradiancja elementów podłogi znajdujących się pod dywanem jest



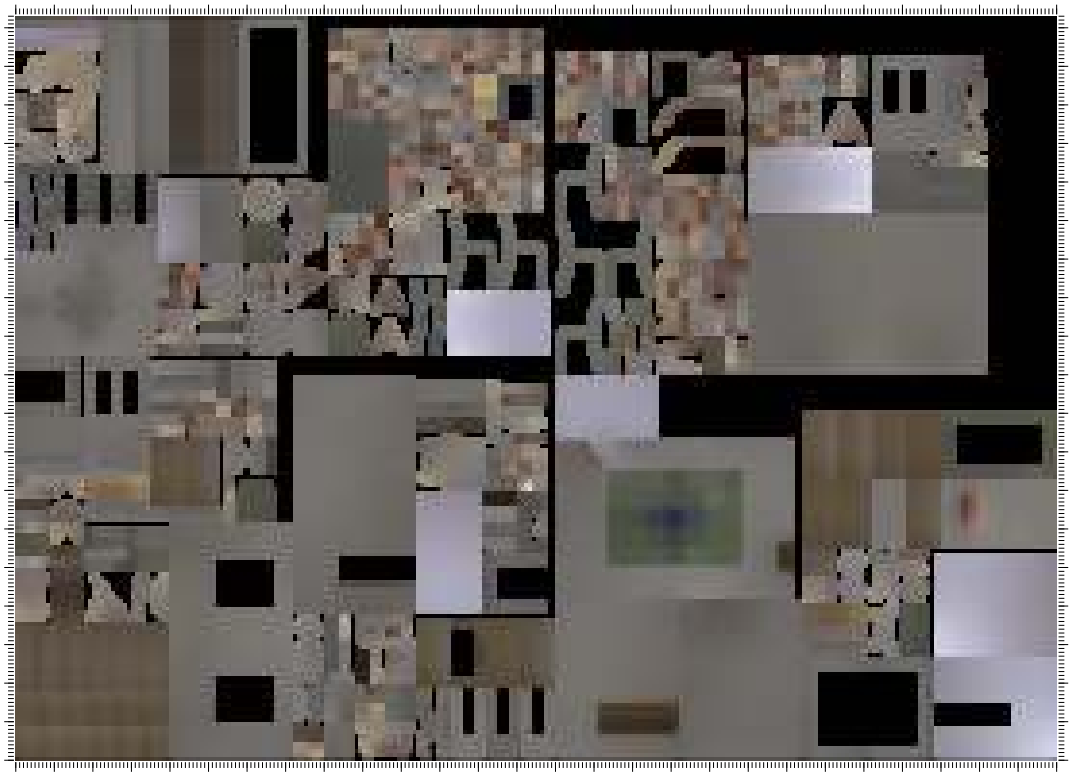
Rysunek 29.7. Obraz emisji światła przez elementy dyskretyzacji

niezerowa (można to zobaczyć na rys. 29.8). Ale elementy te nie są widoczne z innych elementów i nie są też widoczne na końcowych obrazach, dzięki czemu wynik obliczeń pozostaje poprawny.

Listing 29.44. Procedura wykonująca końcowy obraz

C

```
1: void DrawRadianceElements ( BalanceElements *belem, TransBl *trans )
2: {
3:   BalanceObject *bobj;
4:   int           i;
5:
6:   glUseProgram ( bprog_id[17] );
7:   glActiveTexture ( GL_TEXTURE0+MAX_NLIGHTS+MAX_TEXTURES );
8:   glBindTexture ( GL_TEXTURE_RECTANGLE, belem->irrtxt );
9:   glBindVertexArray ( belem->tvao );
10:  for ( i = 0; i < belem->nobj; i++ ) {
11:    bobj = &belem->objtab[i];
12:    LoadMMatrix ( trans, bobj->obj->mm );
13:    CTLUniformi ( belem, CTL_FIRST, bobj->ftrdesc );
```



Rysunek 29.8. Tekstura irradiancji

```
14:     CTLUniformi ( belem, CTL_TXTS, bobj->txts );
15:     glDrawElements ( GL_TRIANGLES, 3*bobj->cntr, GL_UNSIGNED_INT,
16:                      (GLvoid*)(bobj->ftrdesc*3*sizeof(GLuint)) );
17:   }
18:   glBindVertexArray ( 0 );
19:   ExitIfGLError ( "DrawRadianceElements" );
20: } /*DrawRadianceElements*/
```

Do wykonywania końcowych obrazów sceny oświetlonej z wykorzystaniem tekstury irradiancji nie są używane metody rysowania obiektów wskazywane przez pola `redraw` struktur typu `SceneObject` pokazanego na listingu 29.1. Procedura rysowania końcowych obrazów jest pokazana na listingu 29.44. W linii 6 wybiera ona odpowiedni program, w liniach 7–8 przywiązuje teksturę irradiancji[28], w linii 9 przywiązuje obiekt tablicy wierzchołków utworzony przez procedurę z listingu 29.4, a następnie w pętli przesyła do pamięci GPU macierze przekształcenia modelu i wydaje polecenia rysowania trójkątów, z których składają się kolejne obiekty.

[28] Tekstury nałożone na obiekty i tekstury reprezentujące obszary cienia dla punktowych źródeł światła powinny być przywiązane wcześniej.

29.3. Wyniki i wnioski

Scena użyta do uruchomienia implementacji i do zilustrowania tego rozdziału składa się z 11 obiektów opisanych łącznie przez 5729 trójkątów, z których powstało 586 płatów. Przeprowadziłem eksperymenty i pomiary dla dyskretyzacji zgrubnej, średniej, drobnej i bardzo drobnej, określając średnice elementów poszczególnych obiektów za pomocą ostatniego parametru procedury `BeginEnteringObjTriangles`. Wymiary otrzymanych zadań są pokazane w tabeli 29.1, przy czym dla ustalonego poziomu dyskretyzacji w kolejnych eksperymentach liczba n elementów dyskretyzacji i liczba N niezerowych współczynników macierzy G *mogą* się nieco zmieniać[29]. W pierwszych trzech przypadkach macierz G ma około 15% współczynników niezerowych, a w ostatnim niecałe 8%.[30]

Tabela 29.1. Wyniki dyskretyzacji sceny

dyskretyzacja	n	m	$w \times h$	N
zgrubna	12758	1246	200×158	2413743
średnia	26382	1858	270×193	7238061
drobna	97893	5215	482×340	73905549
b. drobna	171074	14694	653×441	199344607

W tabeli 29.2 są podane czasy obliczeń (w sekundach) zmierzone na komputerze stacjonarnym i na laptopie, wyposażonymi w GPU o różnych mocach obliczeniowych. Dla procesora NVIDIA GTX 940M rozwiązanie zadań z drobną i bardzo drobną dyskretyzacją okazało się niewykonalne. Podane są średnie czasy wykonania obliczeń w dziesięciu eksperymentach. Symbol t_{prep} oznacza czas preprocesingu obejmującego nadawanie wierzchołkom trójkątów położenia w dziedzinie tekstury irradiancji i obliczanie macierzy D, A i G. Najwięcej czasu w preprocesingu zajęło obliczanie macierzy G (ozn. t_G), w tym wykonywanie obrazów na ścianach kostki za pomocą procedury `DrawViewFromElem` (t_{draw}).

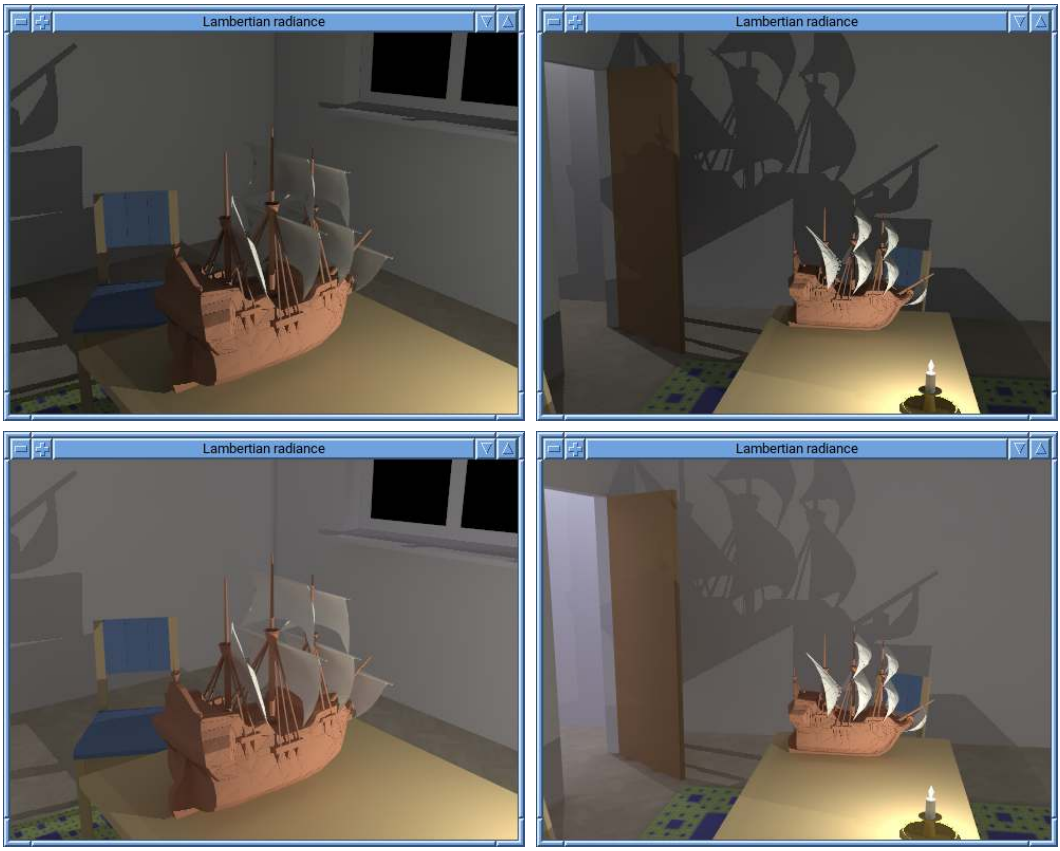
Symbol t_{solve} oznacza czas obliczania tekstury irradiancji. Jego największymi składnikami są czas t_1 obliczania emisji światła przez elementy dyskretyzacji, czas t_2 wykonywania 10 iteracji metody rozwiązywania układu równań i czas t_3 nadawania tekselom końcowych wartości. Oprócz tych obliczeń czas zabiera też znajdowanie obszaru cienia, które trzeba wykonać dla każdego punktowego źródła światła po zmianie jego położenia. Należy o tym pamiętać, planując animację oświetlenia przez poruszające się źródła.

[29] Jest to spowodowane zmianami (pozostającej poza kontrolą aplikacji) kolejności rysowania trójkątów przez procedurę `TexturePatchVertices` (listing 29.6, linie 39–43). Zależnie od tego, do którego trójkąta piksel został „zaliczony", szader z listingu 29.7 może go uznać za „używany", czyli związać z nim element dyskretyzacji, albo za „nieużywany".

[30] Spadek stopnia wypełnienia macierzy G pojawiający się ze wzrostem liczby elementów można wytłumaczyć ograniczoną rozdzielczością obrazów używanych do obliczania współczynników kształtu. Każdy zestaw pięciu obrazów utworzonych w celu obliczenia jednego wiersza macierzy G ma 3888 pikseli, a więc w żadnym wierszu nie może być więcej niezerowych współczynników, nawet jeśli z danego punktu kolokacji widocznych jest więcej makroelementów. Taki spadek pozwala przypuszczać, że dalsze zmniejszanie średnicy elementów nie musi prowadzić do poprawy dokładności wyników.

Tabela 29.2. Czasy obliczeń bilansu energetycznego

	dyskretyzacja	t_{prep}	t_G	t_{draw}	t_{solve}	t_1	t_2	t_3
RTX 3060	zgrubna	0.901	0.893	0.464	0.1018	0.0022	0.0926	0.0069
	średnia	1.848	1.836	0.959	0.1793	0.0032	0.1638	0.0123
	drobna	7.240	7.200	3.581	1.3560	0.0051	1.2317	0.1192
	b. drobna	13.307	13.228	6.288	3.6621	0.0079	3.3244	0.3298
GTX 940M	zgrubna	10.170	10.134	7.916	1.1170	0.0052	1.0099	0.1019
	średnia	21.240	21.176	16.358	3.3478	0.0078	3.0353	0.3046



Rysunek 29.9. Obrazy sceny z oświetleniem obliczonym metodą bilansu energetycznego

Na rysunku 29.9 można obejrzeć końcowe obrazy sceny wykonane za pomocą procedur i szaderów opisanych w tym rozdziale. Oświetlenie dla tych obrazów zostało obliczone przy użyciu średniej dyskretyzacji. Obrazy pokazane wyżej przedstawiają wynik jednej iteracji metody rozwiązywania układu równań (czyli skutki tylko dwóch odbić światła wysłanego przez źródła punktowe), a pod nimi są obrazy otrzymane po wykonaniu dziesięciu iteracji. Już po jednej iteracji na obrazach są widoczne półcienie i przeciekanie koloru.

Opisana tu implementacja prawdopodobnie nie jest gotowa do „wmontowania" do działającej w czasie rzeczywistym aplikacji (np. gry), ale może być podstawą do dalszych badań i rozbudowy. Poniżej przedstawiam kilka pomysłów, które być może zainspirują przynajmniej niektórych Czytelników.

Po pierwsze, warto jeszcze podzielić preprocesing na etapy realizowane przez osobne procedury, które można wywoływać po zmianie pewnych elementów sceny. Na przykład, jeśli (podczas animacji) zmianie uległ kolor lub nałożona na obiekt tekstura, to można powtórzyć tylko obliczenia prowadzące do otrzymania nowej macierzy D.

Jeśli pewne obiekty mogą pojawiać się i być usuwane, to warto przydzielić położenia w dziedzinie tekstury irradiancji wierzchołkom *wszystkich* obiektów. Wtedy dodanie, usunięcie lub zmiana położenia obiektu (przez zmianę macierzy przekształcenia modelu) wymaga tylko ponownego obliczenia macierzy G. Co więcej, jeśli zmianie uległo niewiele obiektów, to większość współczynników kształtu może pozostać niezmieniona, co daje okazję do ograniczenia ilości obliczeń. Wymaga to rozszerzenia repertuaru działań na macierzach rzadkich o kasowanie wierszy i kolumn i dołączanie niezerowych współczynników we wskazanych miejscach.

Można dołączyć do sceny (np. zawiesić na ścianie pomieszczenia) płaskie lustro. Polecam Czytelnikom zastanowienie się, jak zmodyfikować algorytm, a w szczególności, jak zmienić sposób wykonywania obrazów używanych do obliczania współczynników kształtu.

Wreszcie, łatwo jest oświetlić pomieszczenie wpadającym przez okna światłem dziennym, symulowanym przez oświetlenie hemisferyczne. W tym celu wystarczy wprowadzić dwa dodatkowe makroelementy zajmujące dwie półsfery wokół sceny. Najwygodniej jest nadać im numery m i $m + 1$. Wykonując obrazy na ścianach kostki, każdemu pikselowi trzeba zamiast numeru reprezentującego nieświecące tło (widziane przez okna) przypisać numer jednego z tych makroelementów. Macierz G będzie mieć dodatkowe dwie kolumny, a wektor $A\boldsymbol{L}_k$ trzeba rozszerzyć o dwie pozycje, których wartości to wektory radiancji światła dochodzącego z górnej i dolnej półsfery. Można też pokusić się o użycie do oświetlenia światłem dziennym tekstury kostkowej (*skybox*), której teksele spełnią role dodatkowych makroelementów. Jej rozdzielczość nie musi być duża.

Drugie wydanie książki *OpenGL i GLSL* (*nie taki krótki kurs*) jest *poprawione*, przez usunięcie błędów znalezionych w wydaniu pierwszym i ponowne zaimplementowanie aplikacji ilustrujących sposób korzystania ze standardu OpenGL, *poszerzone*, o nowe aplikacje realizujące różne algorytmy za pomocą karty graficznej, i *pogłębione*, przez dodanie bardziej szczegółowych opisów teoretycznych podstaw grafiki komputerowej. Dołączony do książki pakiet oprogramowania jest przygotowany do kompilowania i uruchamiania w systemach Linux/X Window i Windows.

Część II jest poświęcona grafice i przedstawia:

- rysowanie płatów powierzchni Béziera i wykonanych z nich przedmiotów,
- jednoczesne rysowanie wielu instancji tego samego obiektu,
- przykłady sposobów używania zmiennych interfejsu (w tym zmiennych wbudowanych),
- różne modele oświetlenia, od prostych wzorów empirycznych do modeli opartych na prawach fizyki,
- sposoby nakładania tekstur na przedmioty, w tym tekstur reprezentowanych przez tablice tekseli i tekstur proceduralnych,
- sposoby określania wektorów normalnych w celu otrzymania tekstury odkształceń i wizualizacji osobliwości na powierzchniach,
- algorytm cieni,
- metody antyaliasingu,
- sposoby i zastosowania tworzenia obrazów poza ekranem,
- szadery obliczeniowe,
- symulację układu cząsteczek,
- symulację głębi ostrości i rozmycia obiektów w ruchu,
- technikę opóźnionego cieniowania, umożliwiającą m.in. obrazowanie poświaty i zasłanianie otoczenia w przestrzeni obrazu,
- metody oświetlania obiektów przez otoczenie,
- podstawy teoretyczne i pełną implementację metody bilansu energetycznego — symulacji wielokrotnych odbić światła od powierzchni lambertowskich.

Cz. I–III
ISBN 978-83-971793-0-1
9 788397 179301

Cz. II
ISBN 978-83-971793-2-5
9 788397 179325