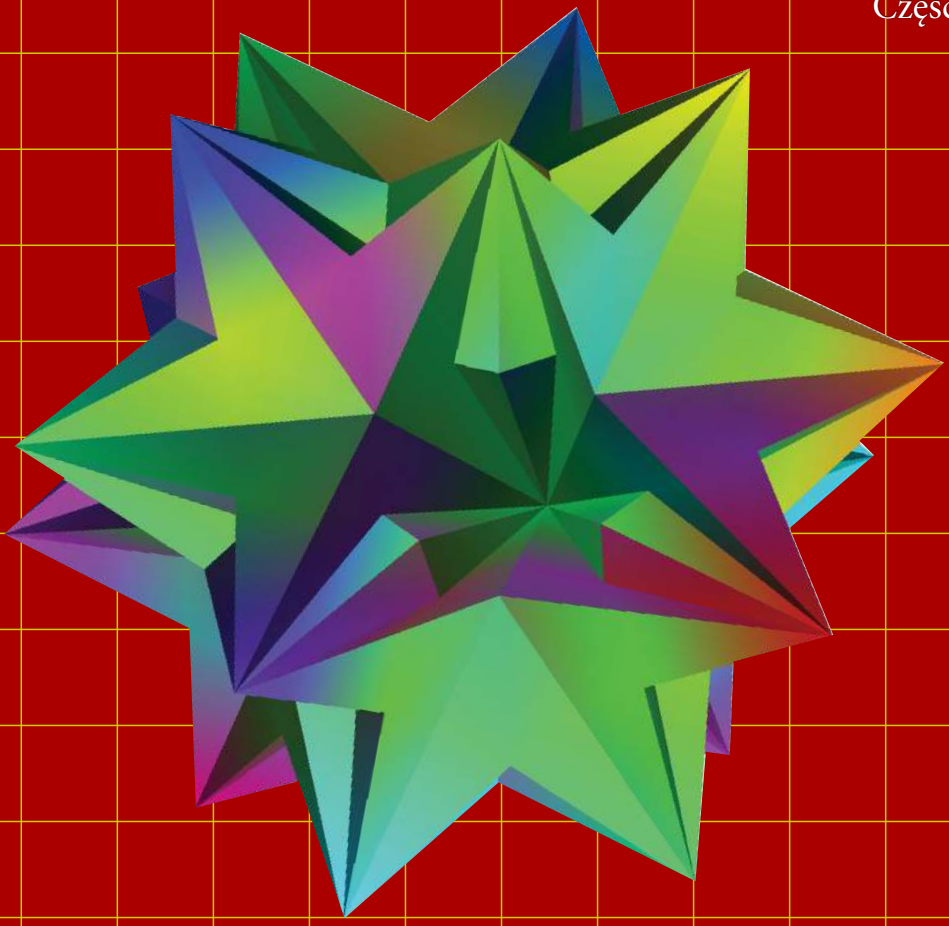
Przemysław Kiciak

Wydanie II
poprawione,
poszerzone
i pogłębione

OpenGL i GLSL

(nie taki krótki kurs)

Część I



WSA

OpenGL i GLSL

Pamięci Michała Jankowskiego

Przemysław Kiciak

Wydanie II
poprawione,
poszerzone
i pogłębione

OpenGL i GLSL

(nie taki krótki kurs)

Część I

WSA

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

ISBN 978-83-971793-1-8 część I
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: 4 grudnia 2024, 3754010 B.

Spis treści

Część I

Część II

Część III

Spis treści części I

Powiedziała, że musimy dziś robić to, na co mamy ochotę.

Tove Jansson: *Dolina Muminków w listopadzie*[1]

Przedmowa

Standard OpenGL opisuje sposób, w jaki programy mogą tworzyć obrazy obiektów trójwymiarowych z wykorzystaniem nowoczesnego sprzętu i jego mocy obliczeniowej. Jednocześnie standard ten umożliwia przenośność programów, które mogą działać na komputerach wyposażonych w procesory graficzne różnych typów (i pochodzące od różnych producentów).

Od swojego powstania, już ponad ćwierć wieku temu, OpenGL udostępnia dla programów pełne możliwości sprzętu. Ściślej biorąc, kolejne wersje OpenGL-a są rozwijane razem z procesorami graficznymi, aby umożliwiać korzystanie z ich najnowszych cech, pozostając przy tym możliwie łatwym w użyciu interfejsem programowania aplikacji. Do roku 2006 zestaw „gotowych do użycia" algorytmów realizowanych przez procedury OpenGL-a był stale rozszerzany. Stanowiąc wygodne oparcie dla programistów, procedury te umożliwiały otrzymywanie coraz większego repertuaru efektów na obrazach, choć repertuar ten pozostawał ograniczony. W 2006 roku standard uległ zasadniczemu przeobrażeniu; znikło wtedy wiele ograniczeń, kosztem jednoczesnej rezygnacji ze wspomnianego wygodnego oparcia, jakie dawały „gotowe" algorytmy. Wprowadzona nieco wcześniej *możliwość* programowania procesorów graficznych sprawiła, że jedynym ograniczeniem podczas pisania programów pozostała wyobraźnia (i budżet) programisty, ale aby to osiągnąć, trzeba było zastąpić większość tych algorytmów programami, które autor aplikacji *musi* dostarczyć. Do ich pisania został stworzony język GLSL — podobny do C język „wysokiego poziomu", umożliwiający skupienie się na rozwiązywanym zadaniu zamiast na zestawie rejestrów i rozkazów maszynowych procesora graficznego.

Pisania programów korzystających z OpenGL-a można się uczyć, korzystając z rozmaitych książek i (dostępnych w Internecie) specyfikacji. Jednak w moim przekonaniu obfitość materiału przedstawionego w istniejących podręcznikach jest ich wadą: autorzy starają się pokazać jak najwięcej elementów OpenGL-a, z konieczności ilustrując je fragmentarycznymi przykładami, co pozostawia u czytelników mieszane uczucia fascynacji, zagubienia i niedosytu (z dodatkiem frustracji podczas poszukiwania odpowiedzi na konkretne pytanie). Natomiast specyfikacje szczegółowo opisują procedury OpenGL-a, wyliczając *wszystkie* dopuszczalne wartości parametrów i *wszystkie* opcje, które można za ich pomocą włączyć albo wyłączyć, co jednak nie bardzo pomaga nowicjuszowi zgadnąć, jak poszczególne procedury współpracują ze sobą, w jakiej kolejności należy je wywoływać i jakie wartości parametrów

[1] Przekład Teresy Chłapowskiej, Nasza Księgarnia, 1980.

trzeba podać w konkretnej sytuacji, aby osiągnąć pożądany (albo w ogóle jakiś widoczny) efekt. W specyfikacjach nie ma też mowy o algorytmach i projektach, które warto by zrealizować za pomocą tych procedur.

Książka *OpenGL i GLSL* (*nie taki krótki kurs*) powstała ze skryptu *OpenGL i GLSL* (*krótki kurs*), który napisałem na potrzeby ćwiczeń z grafiki komputerowej prowadzonych na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego. Kurs, o którym mówi tytuł, polega na zapoznaniu uczestnika z wybranymi[2] elementami standardu OpenGL i języka GLSL za pomocą kompletnych, działających przykładów. Celem tego kursu *nie jest* zatem przedstawienie wszystkich możliwości, opcji, szczegółów i niuansów OpenGL-a, a raczej podanie opisu jak najkrótszej drogi do osiągnięcia wyniku — programu, który wyświetla obrazy w oknie na ekranie i który działa w trybie interakcji czynnej, na bieżąco reagując na działania użytkownika mającego do dyspozycji mysz i klawiaturę. W początkowych rozdziałach zawarłem niezbędne informacje podstawowe, a w kolejnych przedstawiłem krok po kroku aplikacje napisane z wykorzystaniem różnych dostępnych w OpenGL-u konstrukcji. Każda aplikacja ma kilka wariantów; każdy kolejny wariant powstał *po uruchomieniu* wariantu poprzedniego przez dodanie czegoś nowego i ilustruje sposób używania kolejnych elementów OpenGL-a i języka GLSL. Nauka w ramach kursu polega na eksperymentowaniu z aplikacjami i samodzielnym wprowadzaniu zmian, na przykład na dodawaniu do aplikacji nowych obiektów do narysowania lub nowych efektów wizualnych określonych przez Czytelnika — kursanta.

Wielu autorów, usiłując przekonać swoich czytelników o potrzebie stosowania matematyki w grafice komputerowej, przedstawia ją (matematykę) niemal jak wroga, co czasami wygląda jak groteskowa kokieteria; tak odbieram tytuł podrozdziału „Czy to ten okrutny rozdział o matematyce?" z jednej oraz słowa *"we spare you the gory detail"* z innej będącej w moim posiadaniu książki o OpenGL-u (*mimo to* obie te książki uważam za pożyteczne, wiele się z nich dowiedziałem i nauczyłem). Swoje zdanie w tej sprawie ujmę następująco: co najmniej tak samo ważne jak sposób przesyłania macierzy dowolnego przekształcenia do pamięci karty graficznej jest wcześniejsze obliczenie współczynników tej macierzy oparte na poprawnej matematycznie konstrukcji. Matematyka użyta do napisania aplikacji uruchomionych w ramach mojego kursu nie powinna sprawić najmniejszych kłopotów studentom, którzy zaliczyli na pierwszym roku Algebrę Liniową i Analizę, a dla Czytelników, którzy tych przedmiotów nie studiowali, właśnie grafika komputerowa jest najlepszą okazją do poznania tej odrobiny dosyć łatwej (i niewiele wykraczającej poza program szkoły średniej) matematyki, bez której nie ma nie tylko grafiki, ale i mnóstwa innych sposobów radzenia sobie z rozmaitymi problemami przy użyciu komputerów. W tej książce nie ma twierdzeń ani formalnych dowodów, ale potrzebne do napisania aplikacji wzory są porządnie wyprowadzone na podstawie dobrze znanych faktów, a przybliżenia lub nieścisłości w rozumowaniach wyraźnie zaznaczyłem, bo tak trzeba. Nieco bardziej zaawansowana (ale też nietrudna) matematyka opisana w dodatkach A, B i C[3] przyda się Czytelnikom w ich własnych projektach.

[2] przeze mnie, całkowicie arbitralnie

[3] W drugim wydaniu opis fizycznych podstaw oświetlenia wraz z potrzebnym aparatem matematycznym rozszerzyłem i przeniosłem z dodatku C do nowego rozdziału 28. Trochę matematyki jest też w nowym dodatku F.

Ćwiczenie. Przeczytaj głośno zdanie:

Poznam odrobinę matematyki i nie zawaham się jej użyć!

Czytelnik tej książki nie dowie się, gdzie w moim ulubionym zintegrowanym środowisku programistycznym należy kliknąć, aby utworzyć nowy pusty projekt, i to z dwóch powodów. Po pierwsze, nie odmawiając wartości zintegrowanym środowiskom programistycznym, nie używam żadnego z nich, dzięki czemu mogę wszytkich Czytelników traktować, cokolwiek to znaczy, sprawiedliwie. Po drugie, słowo „kliknąć" uważam w najlepszym razie za chwast, a w najgorszym za nowotwór na języku polskim i nie używam takich wyrazów. Zdając sobie sprawę, że walka z wywołującymi ból uszu i przygnębienie słowami — potworkami w rodzaju „mapowanie", „renderowanie", „werteksy" czy „rejkasty" (!) może być już nie do wygrania, podjąłem tutaj próbę ustalenia terminologii szanującej zarówno sens nazywanych pojęć, jak i język, którym mam przywilej i zaszczyt posługiwać się codziennie. Ale w zbiorze znaków dopuszczalnych w programach w C nie ma wszystkich liter polskiego alfabetu, a ponadto mało co wygląda tak źle jak tekst z wymieszanymi (i zniekształconymi) słowami z dwóch języków. Dlatego szanując także język angielski, z którego pochodzą słowa kluczowe języków C i GLSL oraz nazwy procedur OpenGL-a, i licząc na zrozumienie u Czytelników, w aplikacjach przedstawionych w tej książce znaczące identyfikatory (nazwy makrodefinicji, typów, zmiennych i procedur) utworzyłem ze słów angielskich. Ponadto, choć to zwyczaj anglosaski, liczby rzeczywiste w tekście zapisuję tak jak w programach, na przykład 0.5533, dla jednolitości notacji.

Każdy program komputerowy funkcjonuje w pewnym środowisku, tzn. w konkretnym systemie operacyjnym i podsystemie obsługującym urządzenia wejściowe i ekran. Aplikacje opisane w tej książce uruchomiłem w systemie Linux/X Window. Współpraca aplikacji ze środowiskiem może się odbywać za pośrednictwem biblioteki pomocniczej mającej własny interfejs programowania aplikacji, za którym biblioteka ta ukrywa środowisko. Korzystając z takiej biblioteki, można pisać aplikacje, których przeniesienie do innego systemu operacyjnego wymaga tylko ich skompilowania w tym systemie. W tej książce są przedstawione dwie takie biblioteki, FreeGLUT i GLFW. Związanie aplikacji z konkretnym systemem operacyjnym i podsystemem wejścia/wyjścia likwiduje jej przenośność, ale za to umożliwia pełniejsze wykorzystanie możliwości tego środowiska. Dlatego pokazałem także aplikacje, które współpracują bezpośrednio z systemem X Window, używając go do stworzenia (bardzo prostego, ale o nieograniczonych możliwościach rozbudowy) graficznego interfejsu użytkownika.

Podstawowy zamysł, z jakim zabrałem się do tworzenia aplikacji i ich opisów w skrypcie, a potem w książce, był taki, aby żadna linia napisanego przeze mnie kodu nie miała przed Czytelnikami tajemnic. Temu służy zarówno wybór języka programowania (ANSI C, a nie np. C++), jak i zamieszczenie na listingach i skomentowanie w tekście prawie całego kodu aplikacji (w tym również podprogramów pomocniczych, które nie wywołują procedur OpenGL-a). W dalszych rozdziałach opuściłem jednak długie na kilkaset linii ciągi liczb (np. współrzędnych punktów definiujących obiekty do narysowania lub bitowych wzorców znaków używanych do wyświetlania napisów), a ponadto pominąłem fragmenty kodu realizujące rutynowe zadania przedstawione w opisach wcześniejszych aplikacji (takie jak kom-

pilowanie szaderów i uzyskiwanie dostępu do ich komponentów), jeśli powtarzanie szczegółowych opisów rozwiązywania tych zadań byłoby łopatologią. Pliki źródłowe są w książce podzielone na wiele listingów, których kolejność jest podporządkowana objaśnieniom poszczególnych części składowych aplikacji. Ponieważ duże części kodu każdej aplikacji pozostały niezmienione w kolejnych jej wariantach i zostały przeniesione do dalszych aplikacji (i opisy tego kodu nie są powtórzone), utrzymanie zgodności numeracji linii na listingach z numerami linii w plikach źródłowych okazało się niewykonalne. Myślę, że ta niezgodność nie utrudni Czytelnikom lektury, ale aby odbyć kurs, oprócz zdobycia książki trzeba jeszcze zaopatrzyć się w te pliki. Można je znaleźć w sieci pod adresem[4]

`https://it.pwn.pl/Artykuly/OpenGL-i-GLSL-materialy-dodatkowe`

Trzeba też mieć odpowiedni sprzęt (tj. komputer z kartą graficzną realizującą standard OpenGL w wersji co najmniej 4.5) i niezbędne oprogramowanie. Potrzebny jest edytor ASCII (Czytelnik sam sobie wybierze najwygodniejszy) oraz kompilator języka C i narzędzia pomocnicze do uruchamiania programów (np. `gcc`, `make`, `gdb`). Należy mieć zainstalowany i działający sterownik karty graficznej (sposobu instalowania go tu nie przedstawiam) i opisane w rozdziałach 2 i 3 biblioteki. Warto się zaopatrzyć w dokumentację OpenGL-a i języka GLSL (zobacz spis źródeł), aby móc w miarę potrzeb doczytać informacje na temat używanych w aplikacjach procedur. Uprzedzam, że do skutecznej nauki trzeba znaleźć w sobie dużo cierpliwości i dociekliwości. Ale jest za to nagroda: jest nią końcowa satysfakcja.

Książkę tę dedykuję pamięci Michała Jankowskiego (1947–2015), pracownika Wydziału MIM Uniwersytetu Warszawskiego, znakomitego nauczyciela, który przez wiele lat prowadził zajęcia z metod numerycznych, grafiki komputerowej i modelowania geometrycznego. Michał był moim promotorem, a później kolegą z pracy. Zawdzięczam Mu m.in. poznanie teoretycznych podstaw i klasycznych algorytmów grafiki komputerowej. Bez nich nie mógłbym napisać tej książki.

Warszawa, 2019 *Przemysław Kiciak*

[4] Pakiet aplikacji dla drugiego wydania jest dostępny pod adresem podanym na s. 6.

— Ferszlus trzeba roztrajbować.

Julian Tuwim: *Ślusarz*

— Należy użyć odpowiedniego paternu w zależności od kejsu.

Usłyszane przez Annę na szkoleniu

Przedmowa do wydania drugiego

Nauka czegokolwiek nie kończy się na przeczytaniu podręcznika i wykonaniu wszystkich ćwiczeń; nie kończy się nawet na napisaniu podręcznika — zawsze można go poprawić. Pierwsze wydanie *nie takiego krótkiego kursu* OpenGL-a i GLSL-a gruntownie przejrzałem i poprawiłem. Jest w nim kilka informacji nieścisłych lub wręcz nieprawdziwych, choć podanych w dobrej wierze[1]. Ponadto wiele szczegółów w aplikacjach można było zaprojektować i zaprogramować lepiej. W obecnym wydaniu usunąłem wszelkie znalezione błędy i nieścisłości i dokonałem licznych poprawek w opisanych w książce programach. Miały one (i nadal mają) przede wszystkim ilustrować sposób używania OpenGL-a, ale w obecnej wersji wiele pracy włożyłem w takie poprawienie kodu, aby ułatwić Czytelnikom wprowadzanie modyfikacji i eksperymenty. Tam gdzie wpadłem na lepszy pomysł, zrealizowałem go. Ponadto, starając się zachować charakter książki, z jakiej sam chciałbym się uczyć grafiki (i jakiej w swoim czasie nie miałem), dodałem wiele wiadomości uzupełniających, od łatwych do dość zaawansowanych. Ufam, że nowe wydanie będzie służyć Czytelnikom nie tylko jako elementarz, ale też jako źródło, do którego będą wracać po ukończeniu kursu.

Nowe wiadomości dotyczą między innymi:

- większej ilości szczegółów opisu bibliotek FreeGLUT, GLFW i X11, używanych przez aplikacje wyświetlające grafikę,
- budowy i działania korzystających z OpenGL-a natywnych aplikacji systemu Windows,
- „powiększania danych", czyli powielania obiektów przez procesor graficzny,
- używania tekstur, w szczególności tekstur skompresowanych, tekstur bez dowiązania i tekstur kostkowych,
- rysowania obiektów rozciągających się „do nieskończoności",
- adaptacyjnego dobierania parametrów rozdrabniania płatów,
- zastosowania reprezentacji obrazów o szerokim zakresie dynamicznym,
- technik opóźnionego cieniowania,
- modeli oświetlenia opartych na prawach fizyki,
- metody bilansu energetycznego,

[1]Nie łudzę się, że drugie wydanie jest bezbłędne, choć bardzo się starałem, aby było. Błędne informacje można też znaleźć w innych książkach, choć to dla autora żadne pocieszenie, ani tym bardziej usprawiedliwienie.

- wykonywania obrazów w wielu klatkach jednocześnie,
- korekcji gamma,
- obrazowania obiektów fraktalowych,
- obliczeń wykonywanych przez procesor graficzny, niekoniecznie związanych z grafiką, określanych hasłem GPGPU,
- technik uruchamiania aplikacji, w tym „przechwytywania" wywołań procedur i korzystania z kontekstów uruchomieniowych OpenGL-a.

Nowe wersje aplikacji wykonują takie same obrazy jak w pierwszym wydaniu, ilustrując sposoby używania tych samych elementów OpenGL-a, ale wprowadziłem wiele zmian, aby „wyczyścić" kody źródłowe i poprawić ich czytelność. W szczególności w każdej aplikacji oddzieliłem tworzącą okna i obsługującą komunikaty wejściowe „część okienkową" od „części graficznej", której zadaniem jest tworzenie reprezentacji obiektów i wykonywanie obrazów. Części graficzne aplikacji nie mają żadnych instrukcji zależnych od środowiska, w których te aplikacje działają, a z kolei części okienkowe, poza przygotowaniem okien (z odpowiednimi kontekstami OpenGL-a), nie zawierają żadnych instrukcji przygotowujących obiekty do narysowania ani bezpośrednio wywołujących procedury OpenGL-a. Dzięki temu części graficzne wszystkich wersji pierwszej i drugiej aplikacji mogą być połączone z częściami okienkowymi otrzymanymi po wykonaniu bardzo niewielu przeróbek każdego z czterech szkieletów aplikacji opisanych w rozdziale 3 — można więc łatwo otrzymać korzystające z bibliotek FreeGLUT lub GLFW aplikacje przenośne, a także natywne aplikacje systemów X Window i Windows.

Graficzny interfejs użytkownika (GUI, zobacz podrozdz. H.1) użyty w części trzeciej kursu, korzystający z biblioteki Xl1, zmodyfikowałem i zaimplementowałem także w wersji dostosowanej do pracy w systemie Windows; uruchomienie trzeciej aplikacji w tym systemie wymagało napisania nowych części okienkowych dla jej kolejnych wersji (w tym zrealizowania GUI przy użyciu podsystemu GDI), ale części graficzne (wywołujące procedury OpenGL-a) są identyczne dla systemów Linux/X Window i Windows. Uznając, że wystarczy wydrukowany opis GUI dla X Window, nie zamieściłem w książce opisu wersji dla Windows, ale sporo wysiłku włożyłem w przygotowanie aplikacji tak, aby można je było łatwo skompilować i uruchomić w obu systemach. W szczególności zaopatrzyłem pakiet aplikacji w skrypty, na podstawie których program `cmake` jest w stanie zorganizować kompilację pakietu w systemie Linux przy użyciu kompilatora `gcc`, a w systemie Windows przy użyciu programu Visual Studio. Pakiet zawiera kody źródłowe aplikacji przedstawionych w tym kursie i plik `przeczytaj.pdf` z instrukcjami postępowania w obu systemach. Można go znaleźć pod adresem

`https://www.mimuw.edu.pl/~przemek/OpenGL-i-GLSL-II/`

Trzy nowe rozdziały na końcu części drugiej, poświęcone bardziej zaawansowanym technikom programowania grafiki, powstały jednocześnie z aplikacjami ilustrującymi sposoby realizacji tych technik. Po namyśle zdecydowałem się nie zamieszczać całego kodu tych aplikacji na listingach; zakładam, że Czytelnicy tych rozdziałów wcześniej poznali elementarz, tj.

sposoby tworzenia okien, kompilowania szaderów, a także przygotowania przekształceń geometrycznych, tekstur i opisów źródeł światła. Treść rozdziałów skupia się na sednie, czyli na opóźnionym cieniowaniu, zaawansowanych modelach oświetlenia i na bilansie energetycznym, a na listingach są przedstawione tylko najważniejsze procedury i szadery. Oczywiście obok książki jest dostępny komplet plików źródłowych tych aplikacji,

Opisy niektórych nowych tematów (np. modeli oświetlenia) wymagały użycia nieco bardziej zaawansowanej matematyki, bez której wprawdzie można zrozumieć, *jak* coś działa, ale nie da się zrozumieć, *dlaczego* to coś działa. No cóż, świetny programista, który chce być Programistą Wybitnym, bez względu na to, jakie projekty realizuje, nie może być na bakier z matematyką. A grafika komputerowa daje znakomite okazje, aby trochę matematyki poznać, zrozumieć, docenić i polubić.

Czy po wprowadzeniu nowszego standardu programowania grafiki — Vulkana — nadal warto zajmować się OpenGL-em? Moim zdaniem warto. Twórcy Vulkana korzystali z doświadczeń zdobytych podczas rozwijania OpenGL-a i oba standardy mają wiele elementów podobnych. Vulkan jest bardziej zaawansowany technicznie i umożliwia wykorzystanie zdolności sprzętu w znacznie większym stopniu. Jednakże najprostsza aplikacja OpenGL-a, wyświetlająca tylko jeden trójkąt, ma 81 linii kodu w języku C, podczas gdy aby narysować jeden trójkąt za pomocą Vulkana trzeba napisać około 900 linii (w języku C++, zobacz [64]). Co prawda, rysowanie każdego następnego trójkąta kosztuje już trochę mniej pracy, ale zawsze któryś trójkąt jest pierwszy. Jeśli więc hipoteza, że każdy obraz wykonany przez aplikację OpenGL-a mógłby być utworzony także przez aplikację Vulkana, jest prawdziwa (w co nie wątpię[2]), to również prawdą jest, że napisanie takiej aplikacji Vulkana jest co najmniej kilkakrotnie bardziej pracochłonne.

Aplikacje OpenGL-a i Vulkana sterują procesorem graficznym za pomocą szaderów napisanych w tym samym języku GLSL[3]. Wszystkie podstawy matematyczne, w tym metody konstruowania przekształceń geometrycznych (w szczególności rzutowania przestrzeni trójwymiarowej na płaszczyznę) oraz sposoby reprezentowania danych opisujących rysowane obiekty i realizacji modeli oświetlenia są w obu standardach identyczne (nawet jeśli pewne szczegóły implementacji są różne). Poznanie OpenGL-a, który umożliwia znacznie łatwiejsze opanowanie podstaw programowania grafiki, jest zatem dobrą inwestycją także dla osób zainteresowanych docelowo Vulkanem.

Słownictwo używane w naukach technicznych nieustannie się zmienia, czego przykładem są zdania umieszczone powyżej tej przedmowy; zdania te jednocześnie pokazują, jak niewiele się zmieniło od czasów, gdy Tuwim napisał opowiadanie *Ślusarz*, a może nawet od tych czasów, gdy ludzie, aby sprawiać wrażenie wykształconych, z polską mową mieszali łacinę. Idąc wbrew tej modzie, starałem się także w drugim wydaniu książki dbać o język polski i ponawiam stały apel do Czytelników, aby go nie maltretowali.

Na zakończenie przedmowy przypomnę oczywistość: jedyny skuteczny sposób, aby się czegoś nauczyć, polega na robieniu tego czegoś. Nauka programowania polega na progra-

[2] choć czasami to bywa trudne

[3] Pewne konstrukcje GLSL-a nie mogą być użyte w szaderach przeznaczonych do współpracy z aplikacjami Vulkana. W opisach języka GLSL przedstawiam niektóre z tych ograniczeń, a pełną ich listę można znaleźć w specyfikacji [4].

mowaniu, czyli pisaniu programów, kompilowaniu, uruchamianiu, oglądaniu, jak to działa, i poprawianiu do skutku. Choć rzadko kiedy pierwszy, czy nawet drugi samodzielnie napisany program jest doskonały, bez nabrania doświadczenia z nimi nie da się tworzyć tych doskonałych. Czytanie dokumentacji i przykładów z tej książki i z innych źródeł (których wiele można znaleźć m.in. w sieci) pełni w tej nauce rolę ważną, ale dalece niewystarczającą. Dlatego kto chce się nauczyć programowania grafiki, powinien zacząć programować grafikę natychmiast (i nie poddawać się z powodu niepowodzeń), a nie tylko się do tego przymierzać, tłumacząc odkładanie pracy tym, że przecież ten trychter ma kajlę na iberlaufie.

Warszawa, 2024 *Przemysław Kiciak*

Podziękowania. Inspiracji podczas pisania tej książki dostarczali mi (nie zawsze zdając sobie z tego sprawę) studenci, po rozmowach z którymi wpadłem na wiele pomysłów — dotyczących zarówno programowania, jak i sposobów przedstawiania poszczególnych zagadnień w książce. W przygotowaniu pakietu do „wypuszczenia w świat" pomógł mi dr Marcin Wrochna, który podpowiedział mi, jak napisać skrypty dla programu `cmake`, aby aplikacje dały się kompilować w sposób ułatwiający ich modyfikowanie i eksperymenty. Na trasie wyprawy, na którą zapraszam Czytelników, pani redaktor Maria Kasperska znalazła wiele wybojów i kolców, które dzięki Jej spostrzegawczości mogłem wyrównać i sprzątnąć. Wszystkim im jestem ogromnie wdzięczny.

Źródła

Dokumentacja OpenGL-a i GLSL-a

[1] *The OpenGL Graphics System: A Specification* (*Version 4.5*), Khronos Group, https://registry.khronos.org/OpenGL/specs/gl/glspec45.core.pdf.

[2] *The OpenGL Graphics System: A Specification* (*Version 4.6*), Khronos Group, https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf.

[3] *The OpenGL Shading Language 4.50*, Khronos Group, https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.50.pdf.

[4] *The OpenGL Shading Language 4.60.8*, Khronos Group, https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf, https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.html.

[5] https://www.khronos.org/opengl/wiki.

[6] https://registry.khronos.org/OpenGL/index_gl.php.

[7] docs.gl.

[8] https://en.wikipedia.org/wiki/OpenGL.

[9] https://en.wikipedia.org/wiki/OpenGL_Shading_Language.

Programy i biblioteki pomocnicze

[10] https://www.khronos.org/opengles/sdk/tools/Reference-Compiler/.

[11] https://www.x.org/releases/X11R7.6/doc/.

[12] *OpenGL Graphics with the X Window System* (*Version 1.4*), https://www.khronos.org/registry/OpenGL/specs/gl/glx1.4.pdf.

[13] M.J. Kilgard: *The OpenGL Utility Toolkit* (*GLUT*) *Programming Interface. API Version 3*, Silicon Graphics, Inc., 1996, https://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf.

[14] freeglut.sourceforge.net/docs/api.php.

[15] http://www.glfw.org/docs/latest/.

[16] https://github.com/skaslev/gl3w.

[17] https://github.com/Dav1dde/glad.

[18] https://www.opengl.org/sdk/libs/GLM/.

[19] https://xion.org.pl/files/texts/mgt/html/3_4.html.

[20] B. Barney: *POSIX Threads Programming*, https://computing.llnl.gov/tutorials/pthreads/.

Podręczniki OpenGL-a

[21] D. Wolff: *OpenGL 4 Shading Language Cookbook*, second ed., Packt Publishing, 2013.

[22] J. Ganczarski: *OpenGL Podstawy programowania grafiki 3D*, Helion, 2015.

[23] G. Sellers, R.S. Wright Jr, N. Haemel: *OpenGL Księga eksperta*, Helion, 2016.

[24] J. Kessenich, G. Sellers, D. Shreiner: *OpenGL Programming Guide*, nineth ed., Addison–Wesley, 2017.

[25] J. de Vries: *Learn OpenGL — Graphics Programming*, Kendall & Welling, 2020, książka dostępna też w sieci pod adresem learnopengl.com.

[26] E. Luten: www.openglbook.com.

Podręczniki grafiki komputerowej

[27] A.S. Glassner (ed.): *An introduction to ray tracing*, Academic Press, London 1989.

[28] M. Jankowski: *Elementy grafiki komputerowej*, WNT, Warszawa, 1990.

[29] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes: *Computer Graphics — Principles and Practice*, Addison–Wesley, 1990.

[30] P. Prusinkiewicz, A. Lindenmayer: *The Algorithmic Beauty of Plants*, Springer-Verlag, 1990.

[31] *Grafika komputerowa. Metody i narzędzia*, praca zbiorowa pod redakcją Jana Zabrodzkiego, WNT, Warszawa, 1994.

[32] D. Hearn, M.P. Baker: *Computer Graphics*, Prentice Hall, Englewood Cliffs, 1994.

[33] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, R.L. Phillips: *Wprowadzenie do grafiki komputerowej*, WNT, Warszawa, 1995.

[34] R. Parent: *Animacja komputerowa. Algorytmy i techniki*, PWN, Warszawa, 2011.

[35] M. Pharr, W. Jakob, G. Humphreys: *Physically Based Rendering: From Theory To Implementation*, wyd. 3, Morgan Kauffmann, 2017.
Książka dostępna także w sieci pod adresem `https://pbr-book.org/`.

[36] S. Marschner, P. Shirley: *Fundamentals of Computer Graphics*, CRC Press, 2022.

Modelowanie geometryczne

[37] C. de Boor: *A practical guide to splines*, Springer, New York, 1978.

[38] G. Farin: *Curves and surfaces for Computer Aided Geometric Design*, Academic Press, 1993.

[39] G. Farin: *NURBS: from projective geometry to practical use*, A K Peters, Natick, MA, 1999.

[40] H. Prautzsch, W. Boehm, M. Paluszny: *Bézier and B-spline techniques*, Springer, 2002.

[41] P. Kiciak: *Podstawy modelowania krzywych i powierzchni. Zastosowania w grafice komputerowej*, wydanie III, PWN, Warszawa, 2019.

[42] P. Kiciak: Pakiet BSTools, `https://www.mimuw.edu.pl/~przemek/bstools.html`.

Publikacje naukowe

[43] W.R. Hamilton: $i^2 = j^2 = k^2 = ijk = -1$, Broom Bridge, Dublin, 1843.

[44] K.E. Torrance, E.M. Sparrow: Theory for off-specular reflection from roughened surfaces, *J. Opt. Soc. Am. 57* (Sept. 1967), s. 1105–1114.

[45] J.F. Blinn: Models of light reflection for computer synthesized pictures, *Computer Graphics*, 1977, s. 192–198.

[46] R. Cook, K. Torrance: A reflectance model for computer graphics, *Computer Graphics*, 1981, s. 301–316.

[47] J.T. Kajiya: The rendering equation, *Computer Graphics*, 1986, s. 143–150.

[48] Schlick C.: A Customizable Reflectance Model for Everyday Rendering.
Fourth EUROGRAPHICS Workshop on Rendering, Paris, 1993, s. 73–83.

[49] M. Oren, S.K. Nayar: Generalization of Lambert's Reflectance Model, *Computer Graphics*, 1994, s. 239–246.

[50] W. Heidrich, H.-P. Seidel, *Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware*, Computer Graphics Group, University of Erlangen, 1998.

[51] R. Ramamoorthi, P. Hanrahan: *An efficient Representation for Irradiance Environment Maps*, `https://cseweb.ucsd.edu/~ravir/papers/envmap/envmap.pdf`.

Lektury uzupełniające

[52] H.S. Coxeter: *Wstęp do geometrii dawnej i nowej*, PWN, Warszawa, 1967.

[53] B.W. Kerninghan, D.M. Ritchie: *Język ANSI C*, wydanie IX, WNT, Warszawa, 2004.

[54] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: *Wprowadzenie do algorytmów*, WNT, Warszawa, 2005.

[55] D. Kincaid, W. Cheney: *Analiza numeryczna*, WNT, Warszawa, 2006.

[56] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf: *Geometria obliczeniowa. Algorytmy i zastosowania*, WNT, Warszawa, 2007.

[57] `http://people.sc.fsu.edu/~jburkardt/data/smf/smf.txt`.

[58] `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/anim-2/models/smf/`.

[59] Wikipedia: *Utah teapot*, `https://en.wikipedia.org/wiki/Utah_teapot`.

[60] Wikipedia: *Cardioid*, `https://en.wikipedia.org/wiki/Cardioid`.

[61] A. Cheritat: *Mandelbrot set*, `https://www.math.univ-toulouse.fr/~cheritat/wiki-draw/index.php/Mandelbrot_set`.

[62] `https://refractiveindex.info`.

[63] G. Sellers: *Vulkan Programming Guide*, Addison–Wesley, 2017.

[64] A. Overvoorde: *Vulkan Tutorial*, `https://vulkan-tutorial.com`.

1

Wprowadzenie

Aplikacja OpenGL-a składa się z dwóch części: programu wykonywanego przez główny procesor komputera (**CPU**, *central processing unit*) i zbioru programów wykonywanych przez procesor graficzny (**GPU**, *graphics processing unit*). Pierwsza z tych części odpowiada za przygotowanie danych opisujących obiekty do narysowania oraz za współpracę ze środowiskiem (z systemem operacyjnym i podsystemem okien), a druga z nich realizuje poszczególne etapy tworzenia obrazu przez procesor graficzny. Części aplikacji OpenGL-a działające na CPU zazwyczaj pisze się w języku C lub którymś z jego potomków (np. C++). Części aplikacji działające na GPU można (choć nie jest to jedyna możliwość) napisać w języku GLSL.

1.1. Najprostsza aplikacja

Na początek zobaczmy najprostszą aplikację OpenGL-a, której zadaniem jest narysowanie jednego pokolorowanego trójkąta. Jej kompletny kod źródłowy jest pokazany na listingu 1.1. Kompilując ją, oprócz biblioteki GL, zawierającej procedury OpenGL-a, należy dołączyć bibliotekę FreeGLUT (podrozdz. 3.1), współpracującą z systemem okien, oraz biblioteki glad i dl (podrozdz. 2.2), których zadaniem jest dynamiczne połączenie aplikacji z biblioteką GL.

Czynności wstępne wykonywane przez procedurę `main` obejmują przygotowanie struktur danych biblioteki FreeGLUT (linie 54–58) i utworzenie okna aplikacji (linie 59–61). Wywołana w linii 62 procedura `gladLoadGL` dokonuje dynamicznego łączenia aplikacji z biblioteką GL. W liniach 64 i 65 aplikacja rejestruje procedury obsługi komunikatów skierowanych do okna; pierwsza z nich będzie wywołana w celu wykonania obrazu (po uruchomieniu aplikacji i po zdarzeniach takich jak zmiana wymiarów okna lub jego odsłonięcie na ekranie), a wywołanie drugiej nastąpi, gdy użytkownik naciśnie klawisz.

Część aplikacji działająca na GPU składa się z dwóch tzw. szaderów, których kody źródłowe w GLSL-u są zapisane w liniach 6–14 i 16–22. W liniach 66–71 szadery są kompilowane, a w liniach 72–75 następuje łączenie ich w program szaderów, który posłuży do rysowania. Po połączeniu w program skompilowane szadery przestają być potrzebne, więc w liniach 76–77 zajmowana przez nie pamięć jest zwalniana.

Listing 1.1. Aplikacja rysująca trójkąt

C

```
#include <stdlib.h>
#include "glad.h"
#include <GL/freeglut.h>

const GLchar *vertsh[] =
 {"#version 420 core\n"
  "out vec4 colour;"
  "const vec4 p[3] = {{0.9,-0.9,0,1},{0,0.9,0,1},{-0.9,-0.9,0,1}};"
  "const vec4 c[3] = {{1,0,0,1},{0,1,0,1},{0,0,1,1}};"
  "void main ( void )"
  "{"
    "gl_Position = p[gl_VertexID];"
    "colour = c[gl_VertexID];"
  "}"};
const GLchar *fragsh[] =
 {"#version 420 core\n"
  "in vec4 colour;"
  "out vec4 Colour;"
  "void main ( void )"
  "{"
    "Colour = colour;"
  "}"};

int    window;
GLuint prog_id, vao;

void DisplayFunc ( void )
{
  glClearColor ( 1.0, 1.0, 1.0, 1.0 );
  glClear ( GL_COLOR_BUFFER_BIT );
  glUseProgram ( prog_id );
  glBindVertexArray ( vao );
  glDrawArrays ( GL_TRIANGLES, 0, 3 );
  glBindVertexArray ( 0 );
  glUseProgram ( 0 );
  glFlush ();
  glutSwapBuffers ();
} /*DisplayFunc*/

void KeyboardFunc ( unsigned char key, int x, int y )
{
  if ( key == 0x1B ) {
    glDeleteVertexArrays ( 1, &vao );
    glDeleteProgram ( prog_id );
    glutDestroyWindow ( window );
```

```
46:     exit ( 0 );
47:   }
48: } /*KeyboardFunc*/
49:
50: int main ( int argc, char *argv[] )
51: {
52:   GLuint sh_id[2];
53:
54:   glutInit ( &argc, argv );
55:   glutInitContextVersion ( 2, 1 );
56:   glutInitContextFlags ( GLUT_FORWARD_COMPATIBLE );
57:   glutInitContextProfile ( GLUT_CORE_PROFILE );
58:   glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGBA );
59:   glutInitWindowSize ( 480, 360 );
60:   if ( !(window = glutCreateWindow ( "tr81" )) )
61:     exit ( 1 );
62:   if ( !gladLoadGL () )
63:     exit ( 1 );
64:   glutDisplayFunc ( DisplayFunc );
65:   glutKeyboardFunc ( KeyboardFunc );
66:   sh_id[0] = glCreateShader ( GL_VERTEX_SHADER );
67:   glShaderSource ( sh_id[0], 1, vertsh, NULL );
68:   glCompileShader ( sh_id[0] );
69:   sh_id[1] = glCreateShader ( GL_FRAGMENT_SHADER );
70:   glShaderSource ( sh_id[1], 1, fragsh, NULL );
71:   glCompileShader ( sh_id[1] );
72:   prog_id = glCreateProgram ();
73:   glAttachShader ( prog_id, sh_id[0] );
74:   glAttachShader ( prog_id, sh_id[1] );
75:   glLinkProgram ( prog_id );
76:   glDeleteShader ( sh_id[0] );
77:   glDeleteShader ( sh_id[1] );
78:   glGenVertexArrays ( 1, &vao );
79:   glutMainLoop ();
80:   exit ( 0 );
81: } /*main*/
```

W linii 78 aplikacja tworzy pomocniczą strukturę danych, zwaną obiektem tablicy wierzchołków. Po tych przygotowaniach następuje wywołanie procedury `glutMainLoop`, której zadaniem jest odbieranie od systemu okien i przekazywanie właściwym adresatom (tj. oknom, przez wywoływanie zarejestrowanych procedur) komunikatów o wszelkich zdarzeniach, na które aplikacja ma reagować. Procedura ta działa aż do zatrzymania aplikacji.

Procedura `DisplayFunc` (linie 27–38) wykonuje obraz. W linii 30 następuje kasowanie tła, tj. przypisanie wszystkim pikselom koloru białego, ustalonego w linii 29. W linii 31 program szaderów, utworzony w liniach 66–75, jest wskazywany jako ten, który ma być użyty do rysowania, a w linii 32 następuje uaktywnienie obiektu tablicy wierzchołków zawierającego

opis, skąd należy brać atrybuty (takie jak położenie, kolor itp.) wierzchołków rysowanych obiektów. W tej aplikacji opis jest pusty, bo wszelkie atrybuty wierzchołków są dostarczane przez szader zapisany w liniach 6–14.

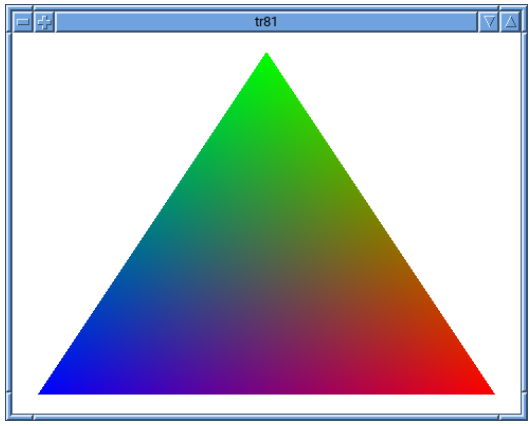
Procedura `glDrawArrays` wywołana w linii 33 powoduje narysowanie jednego trójkąta; dla każdego wierzchołka zostaje wywołany (równolegle) wątek szadera wierzchołków z programu wskazanego w linii 31. Każdy wątek otrzymuje w zmiennej wbudowanej `gl_VertexID` numer (`0`, `1` lub `2`) wierzchołka, który ma przetworzyć. Wektor współrzędnych położenia wierzchołka, brany z tablicy `p`, szader przypisuje zmiennej wbudowanej `gl_Position`. Dodatkowy atrybut, tj. wektor współrzędnych koloru wzięty z tablicy `c`, zostaje przypisany zmiennej `colour`, którą (w odróżnieniu od zmiennej `gl_Position`) trzeba zadeklarować.

Szader zapisany w liniach 16–22, wywołany dla każdego piksela należącego do obrazu trójkąta, ma za zadanie obliczyć jego kolor, co w tym przypadku oznacza skopiowanie koloru otrzymanego na wejściu (w zmiennej `colour`) do zmiennej `Colour` zadeklarowanej jako wyjście szadera.

Procedury wywołane w liniach 34 i 35 odłączają obiekt tablicy wierzchołków i wskazują (nieistniejący) program pusty. Wywołanie procedury `glFlush` wysyła sygnał, że nie będzie więcej obiektów do rysowania, które trzeba jak najszybciej dokończyć. Przedstawiona tu aplikacja (tak jak wszystkie opisane dalej) stosuje podwójne buforowanie obrazu, który nie powstaje bezpośrednio w oknie, tylko w niewidocznym miejscu w pamięci GPU. Procedura `glutSwapBuffers` przesyła gotowy obraz z tego miejsca do okna. Dzięki temu nigdy nie widać na ekranie obrazów niedokończonych.

Procedura `KeyboardFunc` zostaje wywołana po naciśnięciu klawisza; jeśli jest to klawisz `Esc`, to nastąpi zatrzymanie aplikacji. Przedtem aplikacja sprząta, tj. likwiduje obiekt tablicy wierzchołków, kasuje program szaderów i zamyka okno.

Przedstawiona tu aplikacja nie zawiera żadnej diagnostyki, tj. nie bada, czy w czasie działania wystąpiły błędy. Taka diagnostyka jest nieodzowną częścią programowania i uruchamiania aplikacji. Potrzebne do tego procedury poznamy dalej.



Rysunek 1.1. Okno najprostszej aplikacji

Rysowanie *w dawnych wersjach* OpenGL-a odbywało się w ten sposób, że wierzchołki obiektów (np. trójkątów) były podczas rysowania przesyłane przez CPU na bieżąco. Obecnie dane te muszą być umieszczone w pamięci GPU zawczasu — w utworzonych przez aplikację buforach lub, tak jak w aplikacji tu opisanej, w treści odpowiedniego szadera. Powody tej zmiany są przedstawione w następnym podrozdziale.

1.2. Odrobina historii i ideologii

Historia technologicznego rozwoju grafiki komputerowej ze szczególnym uwzględnieniem dziejów standardu OpenGL jest pięknie opisana w (niestety niedokończonym) samouczku [26], którego lekturę polecam. Jednak odrobina historii jest potrzebna do poznania zasady działania aplikacji OpenGL-a oraz do wyjaśnienia pewnych cech najnowszych wersji standardu. Zatem, bezpośrednim poprzednikiem OpenGL-a była biblioteka **IRIS GL** (*Integrated Raster Imaging System Graphical Library*) rozwijana przez firmę Silicon Graphics w latach 1982–1992. Biblioteka ta miała silne związki ze sprzętem i oprogramowaniem systemowym firmy Silicon Graphics.

Standard **OpenGL 1.0** powstał w roku 1992 przez usunięcie tych związków; powołany został wtedy komitet nazwany *OpenGL Architecture Review Board*, w skrócie **ARB**, odpowiedzialny za opracowanie kolejnych wersji standardu. Wersje te, do **OpenGL 2.1** włącznie (rok 2006), dawały coraz więcej możliwości, zachowując pełną zgodność wstecz; aplikacja napisana zgodnie z dowolną z tych specyfikacji mogła współpracować z *bibliotekami* OpenGL realizującymi wszystkie późniejsze wersje standardu.

Podstawowym sposobem działania aplikacji OpenGL-a w wersji 2.1 i wcześniejszych jest **tryb natychmiastowy** rysowania (*immediate mode*). Polega on na tym, że dane opisujące rysowane obiekty geometryczne (dokładniej: punkty lub wierzchołki łamanych lub wielokątów, z dodatkowymi atrybutami takimi jak kolor) są na bieżąco dostarczane przez aplikację działającą na CPU; procesor graficzny po otrzymaniu ostatniego wierzchołka łamanej natychmiast przystępuje do rysowania obiektu i nie zapamiętuje tych wierzchołków. W związku z tym, chcąc wykonać następny obraz (np. sceny, w której niewiele obiektów uległo zmianie), trzeba cały opis *wszystkich* obiektów ponownie przesłać do GPU. Oczywiście, to nie ma znaczenia, jeśli ma być wykonany tylko jeden obraz, i ma niewielkie znaczenie w świecie CAD, gdzie szybkość rysowania jest mało istotna. Ale ma to ogromne znaczenie w grach komputerowych, wyświetlających powstające w czasie rzeczywistym animacje skomplikowanych scen w tempie kilkudziesięciu obrazów na sekundę.

Zmiany standardu OpenGL po opublikowaniu wersji 2.1 zostały wymuszone przez wzrost mocy obliczeniowej i wprowadzenie możliwości programowania GPU; okazało się, że magistrala danych łącząca CPU z GPU stała się wąskim gardłem w przepływie danych. Rolą CPU, zamiast pracowitego dostarczania na bieżąco wszystkich danych do narysowania, stało się umieszczenie w pamięci GPU reprezentacji obiektów, z których składa się scena, „dyrygowanie" GPU, która przeprowadza znaczną część obliczeń i dostarczanie na bieżąco tylko tych danych, które w kolejnej klatce animacji muszą być zmienione. Przy tym, jak sami to zobaczymy, GPU może generować obszerne dane na podstawie modelu opisanego przez nie-

wiele parametrów przesyłanych magistralą przez CPU. Nazywa się to **powiększaniem danych** (*data amplification*).

Obiekty geometryczne należy w celu narysowania odpowiednio **oświetlić** i dokonać **rzutowania**. Na opis obiektu oprócz kształtu składa się informacja o odbijaniu światła przez ten obiekt — w najprostszym przypadku jest to kolor całej powierzchni, ale może też być **tekstura**, czyli funkcja opisująca odbijanie światła przez poszczególne punkty tej powierzchni. Tekstury, a także opis przekształceń prowadzących do obliczenia rzutu obiektu na płaszczyznę obrazu, opisy źródeł światła, mgły itp., także przy pracy w trybie natychmiastowym, należało umieścić w pamięci GPU *przed* przystąpieniem do rysowania. Te dane (zwłaszcza opisy tekstur) są zazwyczaj dość duże i niekoniecznie zmieniają się w kolejnych klatkach animacji. Także wierzchołki figur geometrycznych można umieścić w tablicach w pamięci GPU, dzięki czemu zamiast przesyłać je dla każdej kolejnej klatki animacji, CPU może wydawać tylko polecenia rysowania obiektów opisanych przez zawartość tych tablic (np. wywołując procedurę `glDrawArrays`). W **nowym OpenGL-u** (w wersji 3.0 i późniejszych) to jest w zasadzie jedyny sposób rysowania; wprawdzie tryb natychmiastowy jest dostępny na żądanie (aplikacja, tworząc tzw. **kontekst OpenGL-a**, określa, czy będzie używać tego trybu), ale nie jest zalecany. Żegnaj trybie natychmiastowy, było miło.

W standardzie OpenGL 2.0[1] pojawiła się *możliwość* (jeszcze nie konieczność) pisania programów wykonywanych przez GPU — w stworzonym w tym celu języku GLSL. Programy te, nazwane staropolskim słowem **szadery**[2], są kompilowane i łączone w przeznaczone do działania na GPU programy realizujące **potok przetwarzania grafiki**. Początkowo istniały dwa rodzaje szaderów: **szadery wierzchołków** i **szadery fragmentów**, wywoływane odpowiednio dla każdego punktu przesłanego przez aplikację i dla każdego piksela zrasteryzowanego odcinka lub wielokąta. W szczególności szader wierzchołków mógł realizować rzutowanie punktu na płaszczyznę obrazu dowolną metodą, niekoniecznie jednym ze standardowych w OpenGL-u sposobów rzutowania. Szader fragmentów mógł realizować dowolny efekt końcowy wyglądu obiektu, w szczególności nieosiągalny przy użyciu standardowego zestawu operacji OpenGL-a (np. za pomocą tekstury proceduralnej lub jakiegoś modelu oświetlenia, o którym nie śniło się twórcom standardu).

Ten standardowy zestaw i tak jest dość duży; OpenGL udostępnia mnóstwo przełączników umożliwiających włączanie i wyłączanie poszczególnych efektów, a także wiele zmiennych, których wartości są parametrami we wzorach stosowanych do obliczania oświetlenia itp. Takie bogactwo ma tę wadę, że GPU w trakcie obliczeń musi wykonać dla każdego piksela wiele sprawdzeń, które możliwości są w danej chwili włączone, aby wykonać właściwe obliczenia. Dalsze rozszerzanie standardu stało się zbyt dużym ciężarem; przy tym chyba nie istnieje aplikacja korzystająca ze *wszystkich* możliwości OpenGL-a 2.1, a ich obecność zajmuje pamięć i spowalnia proces wykonywania obrazu.

[1] a wcześniej w rozszerzeniach standardu OpenGL 1.4

[2] Będąc przeciwnikiem zaśmiecania polszczyzny przez robienie kalek językowych z obcych słów, nie mogę jednak uznać dosłownego przekładu słowa *shaders* na „cieniarze" za dobry pomysł. Określenie „programy cieniowania" też nie jest zadowalające, bo cieniowanie jest tylko jednym z wielu obliczeń, które mogą być wykonywane przez programy działające na GPU. Niech więc już będą szadery. *Nie* „szejdery".

Z tego powodu w nowym OpenGL-u nastąpiła radykalna zmiana: wszystkie procedury związane z natychmiastowym trybem rysowania zostały ze standardu usunięte, a dokładniej, **zdeprecjonowane** (*deprecated*), tj. oficjalnie uznane za pozbawione wartości i przestarzałe. Zdeprecjonowane procedury zostały usunięte z tzw. **profilu podstawowego** OpenGL-a (*core profile*). Aplikacja może z nich korzystać po zadeklarowaniu, że zamierza[3]. Zdeprecjonowane zostały też wszystkie standardowe procedury wprowadzające macierze przekształceń wykonywanych w celu rzutowania obiektów i wszystkie procedury wprowadzające parametry oświetlenia; zadania przekształcania i oświetlania obiektów w nowym OpenGL-u mają wykonywać szadery, których dostarczenie przez aplikację stało się odtąd obowiązkowe.

W roku 2006 opiekę nad standardem OpenGL i jego rozwojem objęło konsorcjum **Khronos Group**, zrzeszające głównych producentów sprzętu i oprogramowania. Opublikowane wersje od 3.0 do 3.3 miały na celu umożliwienie wykorzystania starszego sprzętu, natomiast wersje 4.0 i dalsze[4] wymagają sprzętu nowszego, który w chwili opublikowania specyfikacji 4.0 był pieśnią przyszłości, obecnie zaś jest powszechnie dostępny, choć nie w każdym komputerze — nie ma go na przykład w większości produkowanych obecnie laptopów nieprzeznaczonych do grania[5]. Nowe wersje zapewniają zgodność wstecz, do wersji 3.0 włącznie. Nowością w OpenGL-u 3.2 były **szadery geometrii**, a w wersji 4.0 pojawiły się **szadery rozdrabniania**; jedne i drugie znacznie poszerzają możliwości programowania GPU, przy czym nie ma obowiązku ich używania. W wersji 4.3 doszły **szadery obliczeniowe**, które służą do wykonywania *dowolnych* obliczeń przez GPU w sposób masywnie równoległy[6]. Obliczenia te mogą, ale nie muszą, być związane z wykonywaniem obrazu i szadery obliczeniowe nie działają w potoku przetwarzania grafiki.

Nazwa standardu, OpenGL, od samego początku oznaczała możliwość jego rozszerzania. Z tej możliwości korzystają producenci sprzętu. Poszczególne **rozszerzenia** (*extensions*) mają nazwy, będące (dosyć długimi) napisami. Aplikacja może zbadać, czy współpracująca z nią w danej chwili biblioteka OpenGL-a obsługuje dane rozszerzenie i jeśli tak, to może uzyskać dostęp do odpowiedniej procedury, a jeśli nie, to musi albo zrezygnować z efektów wytwarzanych przez to rozszerzenie, albo korzystać ze swoich podprogramów wytwarzających te efekty w jakiś inny sposób, zabierający na przykład więcej czasu na obliczenia. Lista rozszerzeń dostępnych na danym komputerze może być też wyświetlona przez odpowiedni program stanowiący część instalacji OpenGL-a. Poszczególne rozszerzenia mogą być włączane do kolejnych wersji standardu OpenGL i w tejże chwili stają się jego integralną częścią.

Cechą wszystkich wersji OpenGL-a jest **jednowątkowość**; kontekst OpenGL-a może być związany tylko z jednym wątkiem obliczeniowym aplikacji działającej na CPU, co stanowi

[3] Procedury starego OpenGL-a zostały przeniesione do tzw. **profilu zgodności** (*compatibility profile*). Aplikacja, tworząc kontekst OpenGL-a, może podać parametry deklarujące używanie profilu zgodności.

[4] Najnowsza specyfikacja w chwili pisania tego tekstu ma numer 4.6 [2], [4], przy czym dla mnie podstawą była specyfikacja 4.5 [1], [3].

[5] Z tego powodu wersje 3.3 i 4.0 zostały opublikowane jednocześnie. Bywa też tak, że procesor grafiki zgodny w chwili zakupu z pewną specyfikacją, na przykład 4.3, może obsługiwać nowszą wersję po zainstalowaniu nowych sterowników.

[6] Obecnie nawet tanie GPU mają co najmniej kilkadziesiąt procesorów, a te najbardziej wypasione mają ich kilkanaście tysięcy. Kiedyś kupiłem procesory po niecałe 86 groszy za sztukę i jeszcze dostałem sporo pamięci i wiatraki gratis …

pewne ograniczenie. Dlatego w roku 2016 pojawił się znoszący to ograniczenie następca standardu OpenGL, który otrzymał nazwę **Vulkan**. Standard ten jest efektem gruntownej rewizji OpenGL-a, którego procedury zostały starannie przesiane[7] i do których doszły m.in. elementy potrzebne w programowaniu współbieżnym, takie jak semafory i procedury ich obsługi. Vulkan daje programiście znacznie większy stopień kontroli nad sprzętem i sposobem jego wykorzystania, ale wymaga żmudnego skonfigurowania przez aplikację *wszystkiego*, zanim będzie możliwe narysowanie *czegokolwiek*. Ponadto autorzy Vulkana, „czyszcząc" standard OpenGL z ćwierćwiekowych naleciałości (co trzeba było zrobić), usunęli też wiele rozwiązań wygodnych dla programistów. Dlatego, choć oba standardy mają analogiczne elementy takie jak obiekty w pamięci GPU, szadery, potok przetwarzania grafiki itd., programowanie aplikacji Vulkana jest znacznie bardziej skomplikowane niż pisanie aplikacji OpenGL-a, który dlatego (moim zdaniem) lepiej nadaje się do *nauki* programowania grafiki. Być może Vulkan przejmie kiedyś wiodącą rolę w grafice, choć intencją Khronos Group jest współistnienie i dalszy rozwój obu standardów. Ale Vulkanem tu się nie zajmiemy.

1.3. Kontekst — maszyna stanów OpenGL-a

OpenGL to w istocie *specyfikacja*, czyli pewien dokument (np. [1]), zgodnie z którym powinny działać implementacje standardu. Dalej jednak, pisząc „OpenGL", zwykle mam na myśli utworzony przez aplikację **kontekst OpenGL-a**, który jest pewną **maszyną stanów**. Kontekst OpenGL-a zawiera wszelkie dane o wartościach nadanych podczas jego inicjalizacji i później zmienianych przez aplikację, oraz tworzone przez nią obiekty, w tym programy szaderów, bufory, tekstury, obrazy itd. Wywołania procedur OpenGL-a mogą zmieniać stan maszyny, co w szczególności bywa widoczne na ekranie. Pewne elementy stanu są niezmienne, ponieważ wynikają z ograniczeń sprzętu (np. ilości pamięci GPU) i z konkretnych rozwiązań przyjętych przez autorów implementacji, w tym obecności (albo braku) poszczególnych rozszerzeń standardu.

W związku z powyższym oprócz zmieniania stanu maszyny bardzo ważna jest możliwość jego odczytywania przez aplikację. Służy do tego wiele procedur. Zobaczmy przykłady. W kontekście OpenGL-a jest wiele dwustanowych przełączników. Do ich włączania i wyłączania służą procedury `glEnable` i `glDisable`. Aby odczytać stan przełącznika, należy wywołać procedurę `glIsEnabled`. Wszystkie trzy podprogramy mają jeden parametr, który jest identyfikatorem przełącznika i ma odpowiednią nazwę symboliczną, na przykład `GL_DEPTH_TEST` lub `GL_CULL_FACE`[8].

Są też przełączniki wielostanowe; na przykład jest 8 różnych sposobów działania bufora głębokości, za pomocą którego jest realizowany algorytm widoczności. Wyboru sposobu dokonuje się, wywołując procedurę `glDepthFunc` z odpowiednim parametrem, przy czym najczęściej korzysta się ze sposobu domyślnego (`GL_LESS`), ustawionego podczas tworzenia kontekstu, i wtedy nie trzeba go zmieniać.

[7] Te, które przeszły przez sito, otrzymały nowe nazwy. Aplikacja OpenGL-a *nie jest* aplikacją Vulkana.

[8] Listę tych przełączników najłatwiej jest znaleźć w opisie procedury `glEnable` na stronie [7]. Warto tam zaglądać co jakiś czas, ale ten kurs OpenGL-a nie polega na wkuwaniu listy na pamięć.

Na żądanie aplikacji kontekst (czyli opisywana tu maszyna) tworzy obiekty, które (do chwili ich likwidacji) są jego częścią. Identyfikatory tych obiektów są liczbami, które aplikacja powinna pamiętać i podawać jako parametry wywoływanym procedurom. Likwidacja obiektu unieważnia identyfikator[9]. Aplikacja może „pytać", czy dany identyfikator jest wciąż „ważny", czyli czy faktycznie istnieje odpowiadający mu obiekt. Służą do tego funkcje takie jak `glIsBuffer`, `glIsProgram`, `glIsTexture`, odpowiadające na pytania, czy podany parametr jest identyfikatorem istniejącego bufora, programu szaderów lub tekstury. Pewne obiekty są wyposażone w przełączniki, które określają sposoby działania tych obiektów, na przykład metodę interpolacji tekseli przez ewaluator tekstury.

Pewne pytania wymagają odpowiedzi bardziej treściwych niż tylko TAK/NIE. Aby je zadać, zwykle trzeba wywołać jedną z procedur, których nazwy zaczynają się od `glGet`. Jeśli odpowiedź jest liczbą całkowitą, to zazwyczaj udzieli jej procedura `glGetIntegerv`, która ma *dwa* parametry; pierwszy identyfikuje pytanie, a drugi jest adresem zmiennej (typu całkowitego), której procedura przypisze odpowiedź. Pytanie może dotyczyć ograniczeń implementacji, na przykład `GL_MAX_VERTEX_ATTRIBS` (jaka jest największa dopuszczalna liczba atrybutów wierzchołka) albo stanu bieżącego, na przykład `GL_ACTIVE_TEXTURE` (jaki jest ostatnio wybrany numer punktu dowiązania tekstury)

W praktyce ważna jest odpowiedź na pytanie (które aplikacja powinna zadawać dosyć często, zwłaszcza podczas uruchamiania i testowania), czy zdarzył się błąd. Błędy w działaniu OpenGL-a mogą być skutkiem ewidentnych błędów w aplikacji (np. żądaniem dostępu do nieistniejącego bufora), próbą użycia nieobecnego rozszerzenia[10], lub skutkiem wyczerpania zasobów takich jak dostępna pamięć GPU. Do uzyskania odpowiedzi na pytanie, czy (i jeśli tak, to jaki) był błąd, służy procedura `glGetError`.

Jeszcze bardziej skomplikowane odpowiedzi to zawartość bufora, tekst komunikatu o błędach kompilacji szadera lub bieżący obraz. Procedury, które podają te informacje, mają listy parametrów dostosowane do konkretnego zastosowania, na przykład `glGetBufferSubData`, `glGetShaderInfoLog` lub `glGetTexImage2D`[11].

1.4. Potok przetwarzania grafiki

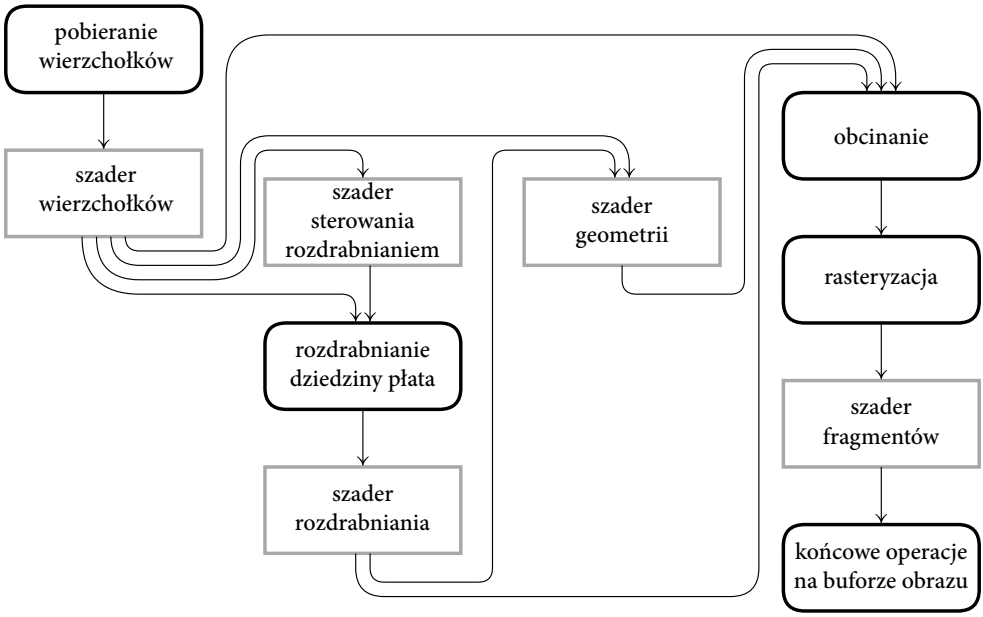
Każdy rysowany obiekt elementarny, tzw. **prymityw**, jest reprezentowany przez skończony ciąg **wierzchołków**. Prymityw może być ciągiem osobnych punktów, ciągiem osobnych odcinków, łamaną (otwartą lub zamkniętą), ciągiem osobnych trójkątów, taśmą trójkątową, wachlarzem trójkątów lub płatem. Wierzchołki są umieszczone w pamięci GPU, w tablicy na-

[9] Identyfikator zlikwidowanego obiektu może być nadany kolejnemu obiektowi, utworzonemu później. Wtedy znów staje się ważny, ale oznacza nowy obiekt.

[10] To może się zdarzyć po przeniesieniu aplikacji na inny komputer, a więc brak sprawdzenia obecności użytego rozszerzenia to też błąd w aplikacji.

[11] Rozdział 23 specyfikacji [1] zawiera pełną listę informacji, które można odczytać z kontekstu OpenGL-a przy użyciu wspomnianych tu procedur. Czytelników, którzy dopiero zaczynają poznawać OpenGL-a, nie namawiam do jego studiowania, ale na przykład podczas pracy nad implementacją metody bilansu enegretycznego (rozdz. 29) wielokrotnie zaglądałem tam po pomoc.

zywanej **buforem wierzchołków** (*vertex buffer*), skąd na polecenie aplikacji są wprowadzane do **potoku przetwarzania grafiki** (*rendering pipeline*), co uruchamia proces rysowania.



Rysunek 1.2. Uproszczony schemat potoku przetwarzania grafiki OpenGL-a

Rysunek 1.2 przedstawia schemat przetwarzania danych przez GPU podczas rysowania; strzałki pokazują możliwe drogi przepływu danych między kolejnymi etapami obliczeń. Szare prostokątne ramki oznaczają *programowalne* etapy tego przetwarzania, czyli miejsca, w których obliczenia są wykonywane przez szadery dostarczone przez aplikację. Pozostałe etapy, uwidocznione w czarnych ramkach, są realizowane przez stanowiące część implementacji OpenGL-a szadery, których aplikacja nie może zmieniać. Etapy te są jednak w znacznym stopniu *konfigurowalne* — ich przebiegiem sterują kwalifikatory układu (*layout qualifiers*) wejścia i wyjścia szaderów, a także dane dostarczone przez aplikację i ustawione przez nią parametry i przełączniki. Podany niżej opis poszczególnych etapów jest przeglądowy i bardzo uproszczony; miejsce na szczegóły jest w dalszych rozdziałach.

Etap **pobierania wierzchołków** (*vertex fetching*) polega na skompletowaniu dla każdego wierzchołka jego atrybutów w celu przekazania ich na wejście szadera wierzchołków; wśród atrybutów *prawie zawsze* jest wektor współrzędnych położenia wierzchołka i pewne dane identyfikacyjne dla wierzchołka. Ponadto aplikacja może określić **atrybuty dodatkowe**, takie jak wektor reprezentujący kolor, wektor normalny i wektor współrzędnych tekstury. Wierzchołek może mieć wiele atrybutów dodatkowych, może też nie mieć ani jednego. Interpretacja atrybutów wierzchołka należy do szaderów, które mogą same produkować wartości potrzebnych w dalszych etapach atrybutów, na przykład koloru, położenia, wielkości kropki rysowanej jako obraz punktu, danych wykorzystywanych do obcinania itd.

Szader wierzchołków (*vertex shader*) ma za zadanie wykonać pewne obliczenie dla pojedynczego wierzchołka. Jego wynikiem jest przekształcony wierzchołek, który będzie przetwarzany dalej, i jego atrybuty. W najprostszym przypadku wierzchołek przekształcony jest kopią wierzchołka dostarczonego na wejście; szader wierzchołków może też dokonać przekształcenia, które obszar, który ma być widoczny na obrazie (**bryłę widzenia**, *view volume*), przekształci na opisaną dalej kostkę standardową. Przekształcenie to wyznaczy rzutowanie obiektów (np. równoległe lub perspektywiczne), efektywnie przeprowadzane na etapie rasteryzacji.

Jeśli w potoku przetwarzania grafiki są obecne szadery rozdrabniania (*tesselation shaders*), to wierzchołki przetworzone przez szader wierzchołków są zbierane w **płaty** poddawane **rozdrabnianiu**, czyli podziałowi na kawałki. **Dziedzina płata** może być trójkątem lub kwadratem. Zadaniem **szadera sterowania rozdrabnianiem** (*tesselation control shader*) jest określenie, jak drobny ma to być podział. Umożliwia to dostosowanie podziału do wielkości obiektu na obrazie. Jeśli obraz obiektu jest bardzo mały (np. wielkości kilku pikseli), to podział może być zgrubny; jeśli zaś obraz obiektu jest duży, to podział na wiele drobnych kawałków może być konieczny do uzyskania wystarczająco dobrej dokładności obrazu. W etapie **rozdrabniania dziedziny płata**, na podstawie parametrów podanych przez szader sterowania rozdrabnianiem, są generowane odcinki lub trójkąty będące kawałkami dziedziny płata. **Szader rozdrabniania** (*tesselation evaluation shader*) dla każdego wierzchołka takiego odcinka lub trójkąta ma skonstruować punkt w przestrzeni (tj. wektor współrzędnych jednorodnych) będący wierzchołkiem odpowiedniego fragmentu płata. Szadery rozdrabniania mają dostęp do tablicy wierzchołków prymitywu (płata), przy czym liczba tych wierzchołków jest ograniczona (zależnie od implementacji, np. do 32). Jeśli wierzchołki płata mają dodatkowe atrybuty (np. kolor lub wektor współrzędnych tekstury), to szader rozdrabniania ma za zadanie dokonać odpowiedniej interpolacji (lub obliczenia w inny sposób) tych atrybutów dla wygenerowanych punktów w przestrzeni.

Kolejny opcjonalny etap obliczeń wykonuje **szader geometrii** (*geometry shader*), który wykonuje obliczenia dla całych punktów, odcinków lub trójkątów wprowadzonych bezpośrednio do potoku przetwarzania grafiki lub otrzymanych z podzielenia łamanej lub taśmy trójkątowej, a także w wyniku rozdrabniania płata. Jeśli na przykład przetwarzane dane reprezentują trójkąt (o wierzchołkach dostarczonych przez szader wierzchołków lub rozdrabniania), to szader geometrii (inaczej niż szader wierzchołków) ma dostęp do wszystkich wierzchołków tego trójkąta. Dzięki temu szader geometrii może wygenerować wektor normalny płaszczyzny trójkąta, który będzie następnie użyty do obliczenia koloru na podstawie przyjętego modelu oświetlenia (alternatywą jest dostarczanie wektorów normalnych jako atrybutów wierzchołków). Szader geometrii może dodatkowo rozdrobnić dany na wejściu odcinek lub trójkąt, produkując z niego łamane lub taśmy trójkątowe. Jeśli żaden szader opisany wcześniej nie dokonał rzutowania (a ściślej przekształcenia wspomnianego w opisie szadera wierzchołków), to powinien to zrobić szader geometrii.

Etapy (stałe i programowalne) opisane wyżej składają się na **część przednią** potoku przetwarzania grafiki. Dane wytworzone w ostatnim z tych etapów opisują punkty, odcinki lub trójkąty, które są następnie obcinane. Obszar przestrzeni $\mathbb{R}^3$, którego zawartość znajdzie się

na obrazie (po przekształceniu wykonanym przez szadery części przedniej), jest **standardową kostką** trójwymiarową, $[-1,1]^3$. **Obcinanie** polega na znalezieniu części wspólnej prymitywu ze standardową kostką, przy czym oprócz sześciu płaszczyzn ścian tej kostki aplikacja może wprowadzić dodatkowe płaszczyzny obcinające; odrzuca się wszystkie części prymitywu położone po „niewłaściwej" stronie każdej z tych płaszczyzn.

Każdy prymityw i to, co zostaje z niego po obcięciu, jest figurą wypukłą: punktem, odcinkiem albo wielokątem wypukłym. Współrzędne x, y wierzchołków obciętego prymitywu są przekształcane tak, aby odwzorować kwadrat $[-1,1]^2$ (ścianę standardowej kostki) na prostokąt o wymiarach **klatki** (*viewport*) podanych w pikselach, po czym następuje **rasteryzacja** — wyznaczenie pikseli, które odpowiadają rzutom punktów obiektów. Dla każdego piksela wyznaczana jest **głębokość** (tj. współrzędna z punktu obiektu w układzie współrzędnych kostki standardowej), na podstawie której jest później wykonywany test widoczności, oraz atrybuty punktu odpowiadającego pikselowi powstałe przez interpolację wszystkich[12] atrybutów wierzchołków prymitywu (odcinka lub trójkąta) dostarczonych przez ostatni szader części przedniej.

Dane te są podawane na wejście **szadera fragmentów** (*fragment shader*), który jest włączony do **części tylnej** potoku przetwarzania grafiki. Na ich podstawie szader ma obliczyć kolor, który będzie przypisany (albo nie) odpowiedniemu pikselowi (elementowi **bufora obrazu**, *image buffer*). Szader fragmentów może obliczać kolor na podstawie otrzymanych na wejściu atrybutów, a także **tekstur** i informacji o źródłach światła umieszczonych zawczasu w pamięci GPU.

Ostateczne przypisanie koloru piksela w etapie końcowych operacji na buforze obrazu zależy od **testu nożyczek** (*scissor test*), **testu maski** (wykorzystującego **bufor maski**, *stencil buffer*, w którym można określić obszar o dowolnym kształcie zabroniony dla rysowania), **testu widoczności** (wykonywanego przy użyciu **bufora głębokości**, *depth buffer*) oraz wybranej **funkcji mieszającej** (*blending function*), która określa ostateczny kolor na podstawie koloru podanego przez szader fragmentów i poprzedniej wartości piksela.

1.5. Programy szaderów

Skompilowane szadery dla poszczególnych etapów potoku przetwarzania grafiki łączy się w **programy szaderów**. Kompletny program pracujący w potoku przetwarzania grafiki *musi* zawierać szader wierzchołków i *powinien* zawierać szader fragmentów — jeśli ten ostatni jest nieobecny, to program może być poprawny, ale wykonany przy jego użyciu obraz jest nieokreślony. Wyniki obliczeń takiego programu mogą jednak zostać zapamiętane i użyte jako dane dla innego programu szaderów, lub ściągnięte do pamięci CPU i przetwarzane dalej przez aplikację. Szadery rozdrabniania i geometrii mogą być w programie nieobecne, a obecność szaderów obliczeniowych w programie zawierającym szadery innych rodzajów jest zabroniona.[13]

[12] z wyjątkiem tych opatrzonych kwalifikatorem `flat`, zob. p. 12.4.5

[13] Potok przetwarzania grafiki to w istocie kompletny działający na GPU program składający się z szaderów dostarczonych przez aplikację i z „gotowych" szaderów realizujących pozostałe etapy, tj. pobieranie wierzchoł-

Podstawowe dane wejściowe i wyniki obliczeń szaderów w potoku przetwarzania grafiki są przekazywane za pomocą tzw. **zmiennych interfejsu**, w tym **zmiennych wbudowanych** opisanych w specyfikacji języka GLSL i dodatkowych zmiennych (globalnych) o nazwach nadanych przez autora szaderów (te ostatnie są używane m.in. do przekazywania wartości dodatkowych atrybutów wierzchołków). Szadery mają też dostęp do opisanych dalej zmiennych jednolitych, a także do dowolnych danych przechowywanych w buforach magazynowych, obrazach (*images*) i teksturach zadeklarowanych w ich treści.

Aplikacja zazwyczaj tworzy wiele programów szaderów, przeznaczonych do rysowania różnych obiektów (np. jeden z nich ma wyświetlać krawędzie wielościanu, inny ma rysować jego ściany jako poteksturowane i oświetlone wielokąty, a jeszcze inny ma wykonywać obliczenia, których wyniki będą użyte do rysowania później). Przed przystąpieniem do rysowania należy wybrać odpowiedni program, wywołując procedurę `glUseProgram` z parametrem będącym identyfikatorem tego programu. Procedura ta powoduje ustawienie programu w blokach startowych i umożliwia przypisywanie wartości jego zmiennym jednolitym.

1.6. Źródła danych w potoku przetwarzania grafiki

Jak już wiemy, wierzchołki, których wprowadzenie do potoku przetwarzania grafiki uruchamia proces rysowania, muszą być zawczasu umieszczone w **buforze wierzchołków** (*vertex buffer*), utworzonym w pamięci GPU. Wierzchołek jest opisany przez współrzędne (kartezjańskie lub jednorodne) swojego położenia w przestrzeni oraz atrybuty dodatkowe, których liczba, w różnych zastosowaniach, może być od zera do kilkunastu. OpenGL umożliwia umieszczenie wszystkich atrybutów w jednym buforze (zawierającym tablicę struktur z polami przechowującymi poszczególne atrybuty) albo w osobnych buforach.

Bezpośrednio przed rysowaniem należy podać informacje umożliwiające etapowi pobierania wierzchołków odnalezienie wszystkich potrzebnych atrybutów, do czego służy procedura `glBindBuffer` przywiązująca bufor wierzchołków do odpowiedniego celu (nazwanego `GL_ARRAY_BUFFER`), procedura `glEnableVertexAttribArray` „uaktywniająca" atrybut i procedura `glVertexAttribPointer`, której parametry opisują rozmieszczenie danych w buforze. Jeśli wierzchołki mają wiele atrybutów, to liczba wywołań tych procedur może być dość duża.

Informacje przekazywane przez wspomniane wyżej procedury są zapisywane w **obiekcie tablicy wierzchołków** (*vertex array object*, *VAO*), co najlepiej jest zrobić jednocześnie z tworzeniem buforów wierzchołków i przesyłaniem do nich danych. Jedna instrukcja wykonana przed rysowaniem — wywołanie procedury `glBindVertexArray` — uaktywnia obiekt tablicy wierzchołków, co oznacza przywołanie wszystkich zapamiętanych w tym obiekcie informacji. Następnie wystarczy wywołać odpowiednią procedurę rysowania (np. `glDrawArrays`), która uruchomi potok przetwarzania grafiki. Każdy obiekt do narysowania powinien mieć swój obiekt tablicy wierzchołków, co umożliwia znaczne uproszczenie aplikacji i likwiduje wiele okazji do zrobienia błędów.

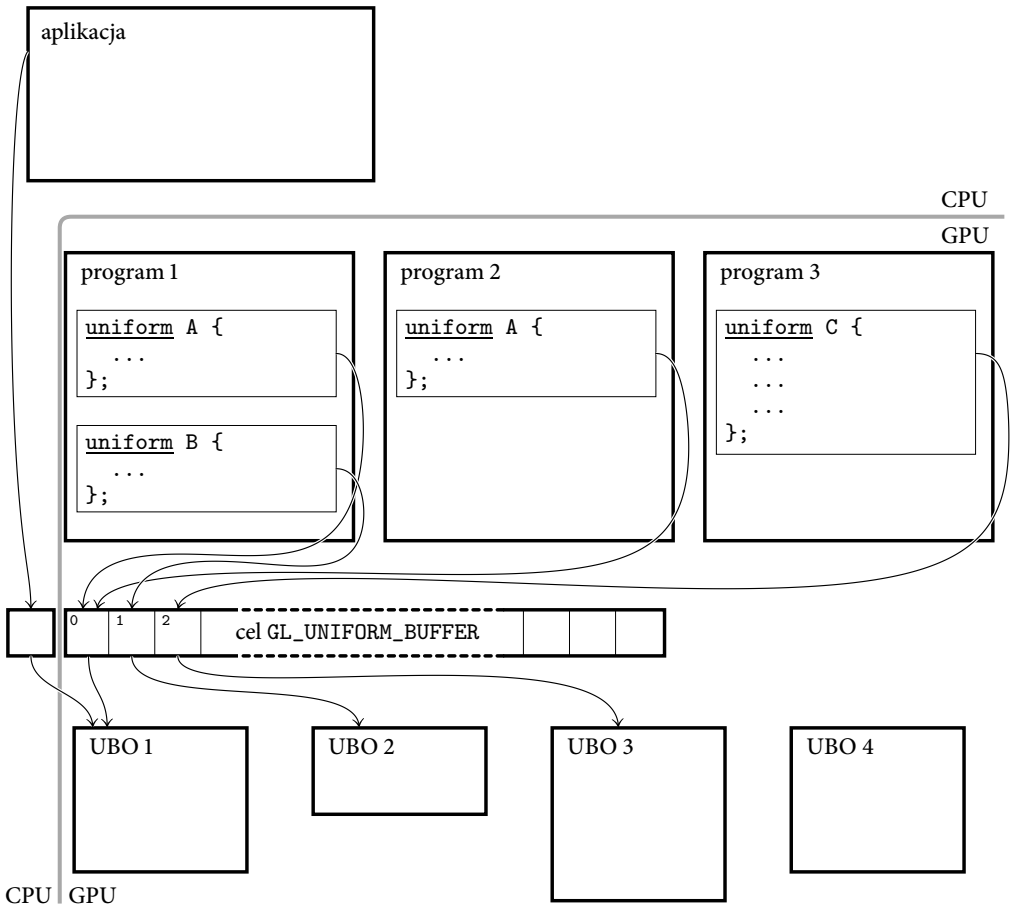
ków, rozdrabnianie dziedziny płata, obcinanie, rasteryzację i operacje końcowe. Program składający się tylko z szaderów obliczeniowych jest niezwiązany z potokiem przetwarzania grafiki.

Dane opisujące przekształcenia geometryczne wierzchołków, inne ich atrybuty (np. domyślny kolor wszystkich wierzchołków pewnego obiektu), oświetlenie, tekstury, fonty itp., aplikacja musi również umieścić w odpowiednich buforach w pamięci GPU. Szadery mogą z nich czytać dane, mogą też w nich zapisywać wyniki swoich obliczeń w celu przekazania dalej — do innych szaderów albo do aplikacji. Każdy bufor po utworzeniu i wypełnieniu po raz pierwszy danymi ma określony sposób używania (np. tylko do odczytu przez szadery lub do odczytu przez aplikację), ma też ustaloną wielkość, której nie można zmienić.

Zmienne jednolite (*uniform variables*[14]) są to zmienne globalne zadeklarowane w szaderach, którym wartości nadaje aplikacja. Szadery mogą tylko odczytywać ich wartości, *jednakowe dla wszystkich* szaderów (stąd nazwa); stwierdzenie to dotyczy zarówno składających się na program szaderów realizujących poszczególne etapy potoku przetwarzania grafiki, jak i wątków jednego szadera (np. wierzchołków lub fragmentów) działających równolegle na wielu procesorach GPU (oczywiście, aplikacja może nadać zmiennym jednolitym nowe wartości przed ponownym uruchomieniem potoku). Zmienne jednolite są stosowane do przechowywania macierzy przekształceń wierzchołków, opisów źródeł światła itd.

Obiekt bufora zmiennych jednolitych (*uniform buffer object*, *UBO*) jest buforem, który przechowuje zmienne jednolite opisane w **bloku zmiennych jednolitych** (*uniform block*) zadeklarowanym w programie szaderów. Aplikacja może utworzyć kilka takich obiektów, jednakowo zbudowanych, ale o różnych wartościach poszczególnych pól. W kontekście OpenGL-a występują tzw. **cele** (*targets*), do których przywiązywane są bufory, aby programy działające na GPU miały do nich dostęp. Obiekty buforów zmiennych jednolitych przywiązuje się do celu nazwanego `GL_UNIFORM_BUFFER`. Ten cel jest *indeksowany*; mianowicie zawiera on tablicę **punktów dowiązania**. Wywołanie odpowiedniej procedury wiąże wskazany UBO z punktem dowiązania o podanym numerze (zrywając poprzednie wiązanie). Bloki zmiennych jednolitych zadeklarowane w programach szaderów są skojarzone z konkretnymi punktami dowiązania w celu `GL_UNIFORM_BUFFER`. Dzięki temu różne programy szaderów mogą mieć dostęp do *tych samych* zmiennych jednolitych w UBO; na przykład wspólne dla różnych programów mogą być macierze przekształceń obiektów — wpisanie nowych współczynników macierzy do UBO (albo dowiązanie UBO z innymi współczynnikami do odpowiedniego punktu) spowoduje, że zmiana będzie widoczna dla wszystkich zainteresowanych programów. Inny przykład zastosowania to UBO zawierający wzorce czcionek (fonty) dla różnych krojów pisma; zmiana kroju pisma dla wyświetlanego tekstu polega wtedy na dowiązaniu UBO z innym fontem do odpowiedniego punktu. Zasadę połączeń bloków zmiennych jednolitych w programach szaderów z UBO za pośrednictwem tablicy punktów dowiązania ilustruje rysunek 1.3. Kwadrat z lewej strony narysowanej tablicy punktów dowiązania symbolizuje główny cel `GL_UNIFORM_BUFFER`; program wykonywany przez CPU ma za pośrednictwem tego celu dostęp do przywiązanego w tym momencie bufora (w tym przypadku

[14] Autorzy polskich książek lub przekładów jak dotąd nie radzili sobie z tym terminem na różne sposoby. Można na przykład przeczytać o „zmiennych uniform" (co jest kapitulacją) lub o „zmiennych jednorodnych" (co jest matematycznym nonsensem). Chyba tylko jeszcze nikt nie napisał o „zmiennych mundurowych". Dobre i to.



Rysunek 1.3. Wiązanie UBO z blokami zmiennych jednolitych

do UBO 1), którego zawartość (przez punkt dowiązania 0) jest też widoczna dla programów 1 i 2 jako blok zmiennych jednolitych `A`.

Zmienne jednolite mogą też być zadeklarowane w szaderach poza nazwanym blokiem. Taka zmienna jednolita jest widoczna dla wszystkich szaderów wchodzących w skład jednego programu i niewidoczna dla innych programów — znajduje się ona w tzw. **domyślnym bloku zmiennych jednolitych** programu.

Szadery mogą korzystać z **buforów magazynowych** (*storage buffers*, jest też skrót SSBO, czyli *shader storage buffer object*, obiekt bufora magazynowego); od zmiennych jednolitych różnią się one tym, że szadery *mogą* do nich wpisywać wyniki swoich obliczeń. Jeden z możliwych scenariuszy działania aplikacji jest taki, że pewien program szaderów (np. składający się z szadera obliczeniowego) wykonuje obliczenia pomocnicze przed rysowaniem, po czym inny program szaderów, realizujący potok przetwarzania grafiki, wykonuje obraz, korzystając

przy tym z danych wpisanych przez pierwszy program do bufora magazynowego. Mechanizm wiązania bloków magazynowych zadeklarowanych w programach szaderów z buforami magazynowymi jest podobny do opisanego wcześniej mechanizmu dla bloków zmiennych jednolitych, ale w tym celu wykorzystuje się tablicę punktów dowiązania w celu nazwanym `GL_SHADER_STORAGE_BUFFER`.

Bufory magazynowe podlegają mniejszym ograniczeniom niż zmienne jednolite; mogą one być znacznie większe i dopuszczają bardziej upakowane rozmieszczenie danych w pamięci. Jednak dostęp do danych w buforze magazynowym może zabierać nieco więcej czasu.

Ważnym elementem w tworzeniu obrazu są **tekstury**; są one zazwyczaj reprezentowane przez tablice wartości w pewnych punktach[15]. Element takiej tablicy, czyli **teksel**, może przechowywać kolor lub inny parametr realizowanego przez szader fragmentów modelu oddziaływania światła z powierzchnią. Przed rysowaniem obiektu z nałożoną teksturą aplikacja wiąże odpowiednie obiekty tekstury, czyniąc je bieżącymi w kontekście i dostępnymi dla szaderów. Dostęp do tekstury odbywa się za pomocą **ewaluatora tekstury** (*sampler*), czyli obiektu przechowującego dodatkowe informacje dla procedur obliczających wartości tekstury. Procedury te, zależnie od ustawień parametrów ewaluatora, podają „surowe" wartości tekseli lub dokonują interpolacji i filtrowania w celu poprawienia jakości obrazu.

Następnych pięć rozdziałów zawiera dość dużo informacji „technicznych". Czytelnik Niecierpliwy, który dopiero zaczyna swoje doświadczenia z OpenGL-em, może przejrzeć je pobieżnie, aby w trakcie studiowania aplikacji w rozdziale 7 i dalszych zaglądać do tych rozdziałów w celu uzupełnienia wiedzy. Jeśli jednak trzeba dopiero zainstalować OpenGL-a, to może warto na początek dokładniej przeczytać rozdział 2.

[15] Ale można też określić **teksturę proceduralną** — jest to funkcja opisana dowolnym wzorem, której wartości obliczy szader fragmentów, a raczej będący jego częścią podprogram.

2

Biblioteki i pliki nagłówkowe OpenGL-a

Aplikacja napisana w *standardzie* OpenGL musi być połączona z odpowiednią *biblioteką* OpenGL-a (w Linuksie `libGL.so`); biblioteki są w pakietach dystrybucji systemu operacyjnego (jeśli nie są zainstalowane domyślnie, to trzeba doinstalować odpowiedni pakiet) lub też są instalowane razem ze sterownikiem procesora graficznego dostarczanym przez jego producenta.

2.1. Pliki nagłówkowe

Oprócz biblioteki powinniśmy też mieć odpowiednie pliki nagłówkowe; podstawowy plik ma nazwę `GL/gl.h` i są w nim prototypy procedur podstawowych oraz procedur *starego* OpenGL-a, przeznaczonych do wykonywania obrazów w trybie natychmiastowym. Procedury nowego OpenGL-a są opisane w pliku `GL/glext.h`. Włączenie obu plików daje dostęp do procedur z *obu* rodzajów OpenGL-a — starego i nowego. Pisząc program, należy uważać, aby konsekwentnie używać tylko starego albo tylko nowego zestawu procedur, bo od mieszania głowa naprawdę boli. Dlatego, pisząc program *w starym stylu*, należy włączyć do aplikacji plik `GL/gl.h`.

Pisząc program *w nowym stylu*, zamiast pary plików `GL/gl.h` i `GL/glext.h` lepiej jest włączyć plik `GL/glcorearb.h`. Zawiera on tylko makrodefinicje, typy danych i prototypy procedur nowego OpenGL-a. Niezależnie od tego, czy używamy pary plików `GL/gl.h` i `GL/glext.h`, czy też pliku `GL/glcorearb.h`, napisanie samych dyrektyw `#include` *nie zadziała*: kompilator nie będzie „widział” prototypów procedur nowego OpenGL-a.

Aby uwidocznić prototypy procedur nowego OpenGL-a, możemy na początku programu (po innych potrzebnych dyrektywach, np. włączających pliki `stdlib.h`, `stdio.h`, `math.h` itd.) napisać

```
#define GL_GLEXT_PROTOTYPES
#include <GL/gl.h>
#include <GL/glext.h>
```

albo

```
#define GL_GLEXT_PROTOTYPES
#include <GL/glcorearb.h>
```

Jeśli zainstalowana biblioteka OpenGL zawiera wszystkie procedury wywoływane przez aplikację, to to wystarczy, aby można ją było skompilować i uruchomić. Ale biblioteka może nie zawierać procedur realizujących rozszerzenia, z których aplikacja korzystałaby, gdyby były dostępne, a których brak może zastąpić jakąś alternatywą. Ponadto podczas uruchamiania aplikacji, tj. znajdowania i poprawiania błędów, ogromną pomocą może być „przechwytywanie" wywołań procedur OpenGL-a; technika ta jest opisana w p. 4.5.3. Dlatego łączenie aplikacji z biblioteką zazwyczaj dokonuje się bardziej skomplikowanym sposobem.

Oto ten sposób: w katalogu z bibliotekami powinna być też biblioteka, która służy do zapewnienia współpracy aplikacji z systemem okien. Dla systemu X Window zazwyczaj dodatkowa biblioteka nie jest potrzebna, bo odpowiednie procedury są obecne w `libGL.so`[1]. Plik nagłówkowy zawierający prototypy procedur obsługujących współpracę z systemem okien ma nazwę `GL/glx.h`. Są w nim opisane procedury, które tworzą kontekst OpenGL-a[2] i przywiązują go do okna aplikacji. Oprócz tego jest tam procedura, która dla podanej *nazwy* dowolnej procedury OpenGL-a (łańcucha ASCIIZ) podaje *adres* tej procedury. Dla systemu X Window procedura podająca adresy ma nazwę `glXGetProcAddress`[3]. Podany adres może być pusty (tj. `NULL`), jeśli implementacja OpenGL-a nie zawiera procedury o takiej nazwie, bo na przykład taka procedura została określona w wersji OpenGL-a późniejszej niż wersja dostępna w danym komputerze, albo jest to procedura niestandardowa, należąca do rozszerzenia standardu dokonanego przez producenta innego sprzętu niż ten, który mamy[4].

Zatem, aplikacja napisana w sposób przenośny powinna zadeklarować odpowiednie zmienne będące wskaźnikami do procedur OpenGL-a i na początku działania „powyciągać" i przypisać tym zmiennym adresy potrzebnych procedur. Nazwy tych zmiennych wskaźnikowych powinny być identyczne z nazwami odpowiednich procedur OpenGL-a. W ten sposób na przykład instrukcja

```
glCompileShader ( shader_id );
```

ma identyczne skutki w obu przypadkach: zarówno wtedy, gdy identyfikator `glCompileShader` jest nazwą procedury dołączonej bezpośrednio do programu, jak i wtedy, gdy jest to

[1] Istnieje osobna biblioteka o nazwie `libGLX.so` z tymi procedurami, która może być potrzebna, jeśli podstawowa biblioteka OpenGL-a tych procedur nie zawiera, bo na przykład jest to nietypowa implementacja standardu.

[2] Informacje o kontekstach są w następnym rozdziale.

[3] Rozszerzeniem specyfikacji GLX 1.3 jest procedura `glXGetProcAddressARB`, w obecnej wersji 1.4 doszła `glXGetProcAddress` (w istocie *ta sama* procedura ma dwie nazwy). To jest ewidentny ślad po włączeniu rozszerzenia do następnej wersji standardu.

[4] Jeden z podstawowych mechanizmów rozwoju standardu OpenGL polega na tym, że producenci wprowadzają swoje rozszerzenia, które następnie, po podjęciu odpowiedniej decyzji przez Khronos Group, zostają włączone (albo nie) do kolejnej wersji standardu. Zresztą, rozszerzenia wprowadzone przez poszczególnych producentów, nawet jeśli nie są w standardzie, często są odtwarzane i udostępniane przez innych producentów, którzy nie chcą być gorsi.

nazwa zmiennej wskaźnikowej, której została nadana wartość otrzymana w odpowiedzi na pytanie o adres czegoś, co się nazywa `"glCompileShader"`.

Jeśli *nie poprzedzimy* dyrektyw `#include` makrodefinicją `#define GL_GLEXT_PROTOTYPES`, to *zamiast* prototypów procedur nowego OpenGL-a kompilator do swoich tablic wprowadzi deklaracje typów wskaźnikowych do procedur o odpowiednich nagłówkach. Przykładowo, dla `glCompileShader` jest to (w uproszczeniu)

```
typedef void (*PFNGLCOMPILESHADERPROC) (GLuint shader);
```

W programie powinniśmy napisać deklarację zmiennej

```
PFNGLCOMPILESHADERPROC glCompileShader;
```

i tej zmiennej należy przypisać odpowiedni adres, co w aplikacji systemu X Window wykonuje instrukcja

```
glCompileShader = glXGetProcAddress ( "glCompileShader" );
```

Uwaga: W opisanych niżej bibliotekach pomocniczych jest to nieco bardziej skomplikowane: zmienne wskaźnikowe mają trochę inne nazwy (np. `glad_glCompileShader`), a w pliku nagłówkowym są makrodefinicje, które ukrywają nazwy tych zmiennych pod „oficjalnymi" nazwami procedur OpenGL-a. Zapobiega to błędom łączenia programów.

2.2. Biblioteki pomocnicze GLEW, gl3w i glad

Procedur OpenGL-a jest kilkaset, co sprawia, że pisanie deklaracji potrzebnych zmiennych i kodu, który pracowicie „wyciąga" adresy tych procedur, zabiera czas, który lepiej byłoby spędzić w przyjemniejszy sposób. Można użyć jednej z pomocniczych bibliotek, GLEW, gl3w lub glad, które wykonują tę pracę. Dodatkową zaletą tego sposobu jest ukrycie przed aplikacją zależności systemowych: bibliotek tych używamy tak samo w systemie Unix/Linux+X Window, jak i w systemie Windows.

2.2.1. Biblioteka GLEW

Aby użyć biblioteki GLEW (`libGLEW.so`), *zamiast* plików `GL/gl.h` i `GL/glext.h` albo `GL/glcorearb.h`, do programu włączamy plik `GL/glew.h`. Biblioteka GLEW zawiera (globalne) zmienne wskaźnikowe do procedur OpenGL-a, zgodne z opisem podanym wyżej.

Uwaga: Jeśli korzystamy z biblioteki GLEW, to dyrektywę `#include <GL/glew.h>` dajemy we *wszystkich* plikach źródłowych (w C) naszej aplikacji — w ten sposób wszystkie wywołania procedur OpenGL-a będą wykonywane za pośrednictwem zmiennych wskaźnikowych w bibliotece GLEW, które dzięki temu będą widoczne podczas kompilacji. To samo dotyczy plików nagłówkowych bibliotek gl3w i glad.

Na początku działania (*po* utworzeniu okna i kontekstu OpenGL-a, co jest opisane w następnym rozdziale) aplikacja wykonuje instrukcję

```
if ( (ec = glewInit ()) != GLEW_OK ) {
  fprintf ( stderr, "Error: %s\n", glewGetErrorString ( ec ) );
  exit ( 1 );
}
```

(zmienna `ec` ma być typu `int`). Procedura `glewInit` przypisuje odpowiednie adresy zmiennym i w razie niepowodzenia informuje o tym aplikację, przekazując kod określający, co poszło źle. Procedura `glewGetErrorString` tłumaczy ten kod na język angielski.

2.2.2. Biblioteka gl3w

Sposób użycia biblioteki gl3w jest podobny, z tym że w odróżnieniu od biblioteki GLEW, dostępnej w pakiecie[5], musiałem ją ręcznie skompilować i zainstalować, co okazało się dość proste; poza plikiem nagłówkowym kod źródłowy składa się z jednego pliku napisanego w C. Ze strony projektu [16] trzeba ściągnąć plik `gl3w-master.zip` i po jego rozpakowaniu wydać następujące polecenia:

```
cd gl3w-master
python gl3w_gen.py
cd src
```

Program w języku Python komunikuje się (przez sieć) z serwerem, na którym znajdują się informacje potrzebne do wygenerowania plików źródłowych w C: `gl3w.c`, `gl3w.h` oraz `glcorearb.h`. Dalej mamy dwie możliwości: wygenerowany przez powyższe polecenia plik źródłowy `gl3w.c` możemy skompilować w zwykły sposób, otrzymując pojedynczy plik obiektowy, `gl3w.o`, który dodajemy do aplikacji na etapie łączenia. Do listy bibliotek dołączanych do aplikacji należy dodać bibliotekę `libdl`, która zawiera procedury `dlopen` i `dlclose` wywoływane przez procedury w pliku `gl3w.o` (kompilując plik z procedurą `main` aplikacji, trzeba podać kompilatorowi opcję `-ldl`). Od nas zależy, gdzie na dysku umieścimy ten plik oraz plik nagłówkowy `gl3w.h`, który musi być włączony przez źródła aplikacji dyrektywą `#include`. Druga możliwość to utworzenie biblioteki łączonej dynamicznie (*shared object*). Wykonują to polecenia

```
gcc -c -Wall -ansi -pedantic -O2 -fPIC -I../include gl3w.c -o gl3w.o
gcc -shared -Wl,-soname,libgl3w.so -o libgl3w.so.1 gl3w.o
```

Potem trzeba umieścić pliki nagłówkowe i bibliotekę w katalogach, w których będą dostępne dla kompilatora (czyli w `/usr/include/GL` oraz `/usr/lib64`, albo gdzieś we własnym katalogu domowym, jeśli nie mamy lub nie chcemy skorzystać z uprawnień administratora). Korzystanie z biblioteki współdzielonej ma tę zaletę, że plik binarny aplikacji

[5] w używanej przeze mnie dystrybucji systemu Linux

jest mniejszy niż plik z procedurami dołączonymi statycznie. Ale uruchomienie tak przygotowanej aplikacji na innym komputerze wymaga, aby biblioteka współdzielona była tam zainstalowana, na co nie zawsze można liczyć.

W plikach źródłowych aplikacji należy *zamiast* `#include <GL/glew.h>` napisać dyrektywę `#include "gl3w.h"` albo `#include <GL/gl3w.h>` (zasady umieszczania tych dyrektyw w kodzie aplikacji są takie same jak dla biblioteki GLEW). Po utworzeniu okna i kontekstu OpenGL-a aplikacja powinna wykonać instrukcję

```
if ( gl3wInit () ) {
  fprintf ( stderr, "Error: gl3w initialisation failure\n" );
  exit ( 1 );
}
```

Uwaga: Jednocześnie z plikami źródłowymi `gl3w.c` i `gl3w.h` program `gl3w_gen.py` generuje również plik `glcorearb.h`. Warto ten plik skopiować do katalogu `/usr/include/GL`, nadpisując plik `glcorearb.h` zainstalowany razem ze sterownikiem GPU[6], ponieważ zawiera on makrodefinicje i nagłówki procedur związanych z większą liczbą rozszerzeń, w tym takich, które weszły do następnej wersji standardu OpenGL. Bywa tak, że sterownik obsługuje rozszerzenia nieopisane w instalowanym razem z nim pliku `glcorearb.h`.

2.2.3. Biblioteka glad

W podobny sposób można przygotować bibliotekę glad [17]. Polecenie

```
pip install --user glad
```

ściąga z Internetu i zapisuje w katalogu `.local/bin` program w Pythonie o nazwie `glad`. Polecenie `glad --help` wyświetla dostępne opcje, które umożliwiają wybór docelowego języka aplikacji (np. C, Pascal i parę innych), wersji OpenGL-a, listy rozszerzeń standardu, z których chcemy korzystać itd. Przykładowe polecenie wywołujące ten program ma postać

```
glad --profile core --out-path . --api gl=4.5 --generator c \
  --reproducible --local-files
```

W wyniku powstaną pliki źródłowe `glad.c`, `glad.h` i `khrplatform.h`, które trzeba przenieść do odpowiednich katalogów; może przy tym być konieczne zmodyfikowanie w pliku `glad.c` dyrektywy `#include "glad.h"`, aby kompilator mógł odnajdywać plik nagłówkowy. Plik `glad.h` *zastępuje* plik nagłówkowy `glcorearb.h` biblioteki `GL`, przy czym zestaw procedur opisanych w tym pliku odpowiada podanej w parametrach wywołania programu `glad` wersji standardu (np. 4.5 w przykładzie podanym wyżej). Przewidując, że aplikacje będą używane na sprzęcie realizującym starsze wersje, warto wygenerować pliki z odpowiednio mniejszym numerem wersji. Istnieje przy tym możliwość precyzyjnego określenia potrzebnych rozszerzeń wybranej wersji standardu.

[6] Do tego są potrzebne uprawnienia administratora.

Obecna w bibliotece glad procedura pobierająca adresy procedur OpenGL-a, którą trzeba wywołać na początku działania aplikacji, ma nazwę `gladLoadGL` i pustą listę parametrów. Jest też alternatywna procedura `gladLoadGLLoader`, w której wywołaniu należy podać jako parametr procedurę `glXGetProcAddress` lub inną procedurę pobierania adresów, dostępną w używanej przez aplikację bibliotece realizującej API do systemu okien (np. `wglGetProcAddress` w systemie Windows, `glutGetProcAddress` w bibliotece FreeGLUT lub `glfwGetProcAddress` w bibliotece GLFW, zobacz rozdz. 3).

Z pliku `glad.c` też można wygenerować bibliotekę łączoną dynamicznie, na przykład w pliku o nazwie `libglad.so.1` (w taki sam sposób jak w przypadku biblioteki gl3w), ale chyba lepiej jest dołączać skompilowany plik `glad.o` bezpośrednio do aplikacji.

Dla wygody warto plik nagłówkowy biblioteki GLEW, gl3w albo glad włączyć do źródeł aplikacji za pośrednictwem dodatkowego pliku takiego jak na listingu 2.1, który łatwo można dostosować do lokalnej instalacji OpenGL-a (także wtedy, gdy nie mamy prawa pisania w katalogu `/usr/include`). Zmiana biblioteki używanej do pobierania adresów procedur OpenGL-a wymaga tylko zmiany makrodefinicji w pierwszej linii i odpowiedniego zmodyfikowania plików `Makefile`.

Listing 2.1. Plik `openglheader.h`

C

```
#define USE_GL3W /* można zmienić na USE_GLAD lub wykomentować */

#ifdef USE_GL3W
  #include "GL/gl3w.h"
#else
  #ifdef USE_GLAD
    #include "GL/glad.h"
  #else           /* ale tego nie polecam */
    #include <GL/glew.h>
    #define USE_GLEW
  #endif
#endif
```

Listing 2.2 przedstawia procedurę (umieszczoną w pliku `utilities.c`, zawierającym różne procedury pomocnicze opisane w rozdziałach 4–6), która używa wybranej biblioteki do pobrania tych adresów. Procedura `ExitOnError` (listing 4.2), wywoływana razie niepowodzenia, wypisuje napis podany jako parametr i zatrzymuje program.

Po uzyskaniu dostępu do procedur następuje sprawdzenie, czy zainstalowana biblioteka OpenGL-a obsługuje wersję standardu opisaną przez parametry. Procedura `glGetIntegerv` (wywoływana za pomocą wskaźnika, któremu dopiero co nadała wartość procedura z używanej biblioteki pomocniczej) kolejno przypisuje zmiennym `ma` i `mi` części numeru wersji realizowanego standardu, po czym następuje ich porównanie z numerem wersji wymaganej przez aplikację i podanym w parametrach `major` i `minor`. Jeśli wersja potrzebna aplikacji

jest nowsza niż wersja obsługiwana, to następuje zatrzymanie aplikacji, poprzedzone poinformowaniem użytkownika o przyczynie tego przykrego incydentu.

Listing 2.2. Procedura pobierania adresów

C

```
1: void GetGLProcAddresses ( GLint major, GLint minor )
2: {
3:   GLint ma, mi;
4:   char  s[100];
5:
6: #ifdef USE_GL3W
7:   if ( gl3wInit () )
8:     ExitOnError ( "GetGLProcAddresses (gl3w)\n" );  /* listing 4.2 */
9: #else
10: #ifdef USE_GLAD
11:   if ( !gladLoadGL () )
12:     ExitOnError ( "GetGLProcAddresses (glad)\n" );
13: #else
14:   int ec;
15:
16:   if ( (ec = glewInit ()) != GLEW_OK )
17:     ExitOnError ( glewGetErrorString ( ec ) );
18: #endif
19: #endif
20:   glGetIntegerv ( GL_MAJOR_VERSION, &ma );
21:   glGetIntegerv ( GL_MINOR_VERSION, &mi );
22:   if ( ma < major || (ma == major && mi < minor) ) {
23:     sprintf ( s, "Requested OpenGL version %d.%d not supported,\n"
24:                  "only %d.%d available\n", major, minor, ma, mi );
25:     ExitOnError ( s );
26:   }
27: } /*GetGLProcAddresses*/
```

2.3. Nazwy i typy danych w OpenGL-u

Typy zmiennych liczbowych przetwarzanych przez OpenGL-a (w tym parametrów procedur OpenGL-a) mają nazwy zdefiniowane w opisanych wcześniej plikach nagłówkowych; nazwy te zaczynają się od **przedrostka** `GL` i są w zasadzie synonimami nazw typów standardowych w C. Filozofia przyświecająca ich wprowadzeniu jest taka, że nazwa ma podkreślać zastosowanie, a ponadto ma być zagwarantowana jednoznaczność reprezentacji: `GLint` ma zawsze oznaczać typ liczb całkowitych 32-bitowych ze znakiem w kodzie uzupełnieniowym do 2 niezależnie od tego, czy właśnie taki typ ukrywa się pod nazwą `int`. Nazwy najważniejszych typów prostych OpenGL-a są zebrane w tabeli 2.1.

Wszystkie stałe symboliczne określone za pomocą dyrektyw `#define` w plikach nagłówkowych OpenGL-a mają nazwy zaczynające się od przedrostka `GL_`.

Tabela 2.1. Typy OpenGL-a

typ OpenGL	typ C	zastosowanie
`GLvoid`	`void`	typ pusty,
`GLboolean`	`unsigned char`	zmienne boolowskie o wartościach `GL_FALSE` i `GL_TRUE`,
`GLbyte`	`signed char`	8-bitowe liczby całkowite ze znakiem,
`GLubyte`	`unsigned char`	8-bitowe liczby całkowite bez znaku,
`GLchar`	`char`	kody ASCII,
`GLshort`	`short`	16-bitowe liczby całkowite ze znakiem,
`GLushort`	`unsigned short`	16-bitowe liczby całkowite bez znaku,
`GLint`	`int`	32-bitowe liczby całkowite ze znakiem,
`GLuint`	`unsigned int`	32-bitowe liczby całkowite bez znaku, identyfikatory obiektów (buforów, szaderów itp.),
`GLenum`	`unsigned int`	stałe o nadanych nazwach,
`GLsizei`	`int`	wielkości buforów i tablic,
`GLbitfield`	`unsigned int`	32-bitowe pole bitowe,
`GLint64`	`long int`	64-bitowe liczby całkowite ze znakiem,
`GLuint64`	`unsigned long`[7]	64-bitowe liczby całkowite bez znaku, uchwyty (*handles*) tekstur,
`GLfloat`	`float`	32-bitowe liczby zmiennopozycyjne,
`GLdouble`	`double`	64-bitowe liczby zmiennopozycyjne.

2.4. Przedrostki i przyrostki nazw procedur

Wszystkie procedury podstawowego OpenGL-a mają nazwy zaczynające się od **przedrostka** `gl`, po którym następuje wielka litera — początek właściwej nazwy procedury. Zatem na przykład `glGetError` jest nazwą procedury w podstawowej bibliotece OpenGL-a. Ale procedur o nazwach zaczynających się od `glX` *może* nie być w `libGL.so` i wtedy należy ich szukać w bibliotece `libGLX.so`; tak czy inaczej, ich prototypy są w pliku `GL/glx.h`.

Przyrostek nazwy procedury OpenGL-a, *jeśli występuje*, określa typ jej parametrów i sposób ich podania — służy to do rozróżniania procedur spełniających tę samą rolę. Przyrostki składają się z jednej, dwóch lub trzech liter i ewentualnie poprzedzającej je jednej cyfry. Na przykład procedura `glUniform1f` nadaje (skalarnej zmiennopozycyjnej) zmiennej jednolitej wartość podaną przez jeden parametr typu `GLfloat`. Procedura `glUniform3f` nadaje zmiennej mającej trzy składowe zmiennopozycyjne wartość podaną w trzech parametrach typu `GLfloat`, a procedura `glUniform3fv` robi to samo, ale w tym przypadku trzy liczby typu `GLfloat` mają być podane w przekazanej jako parametr tablicy.

Zatem: cyfra `1`, `2`, `3` lub `4` w przyrostku określa liczbę podanych (jako osobne parametry lub w tablicy) liczb. Litery `b`, `ub`, `s`, `us`, `i`, `ui`, `f` albo `d` określają typ parametru lub parametrów, przy czym obecność litery `u` oznacza typ całkowity bez znaku, litery `b`, `s` i `i` oznaczają

[7] W komputerach z 32-bitowymi CPU typy `GLint64` i `GLuint64` są zdefiniowane jako `long long int` i `unsigned long long int`.

odpowiednio liczby całkowite 8-, 16- i 32-bitowe, a litery `f` i `d` — liczby zmiennopozycyjne, 32- lub 64-bitowe. Obecność litery `v` oznacza, że parametry są przekazywane w tablicy[8].

2.5. Zestawienie bibliotek

W nawiasach są podane nazwy plików bibliotek OpenGL-a, najpierw w systemie Linux, a po średniku w systemie Windows. Biblioteki mogą być **statyczne**, których potrzebne części są dołączane do aplikacji przez kompilator, lub **dynamiczne**, które muszą być zainstalowane w każdym systemie, w którym aplikacja ma działać. Niektóre biblioteki mają obie wersje. Mając wybór, trzeba go dokonać.

GL (`libGL.so`; `opengl32.lib`) — podstawowa biblioteka OpenGL-a, zawiera procedury rysowania i procedury niezbędne do tego, aby przygotować proces rysowania. Jej pliki nagłówkowe mają nazwy `gl.h`, `glext.h`, `glcorearb.h`, sposób ich używania w aplikacjach jest opisany wcześniej w tym rozdziale.

GLU — biblioteka pomocnicza, większość procedur w niej zawartych jest dostosowana do starego OpenGL-a. Nie będziemy jej używać, ale w tej książce jest o niej kilka wzmianek, tam gdzie ich umieszczenie uznałem za celowe.

GLX (`libGLX.so;`) — biblioteka z procedurami organizującymi współpracę OpenGL-a z systemem X Window; jej głównym zadaniem jest tworzenie kontekstów OpenGL-a, wiązanie ich z **kanwami** (*drawables*), tj. oknami (*windows*) i wewnętrznymi buforami obrazu[9] (*pixmaps*) systemu X Window oraz podawanie adresów procedur OpenGL-a. Jej plik nagłówkowy nazywa się `glx.h`.

Procedury z tej biblioteki mogą być obecne w bibliotece `libGL.so` (tak jest w używanej przeze mnie instalacji w Linuksie) i wtedy osobna biblioteka `libGLX.so` jest zbędna (nawet jeśli jest obecna na dysku).

WGL — zbiór procedur pełniących w systemie Windows rolę analogiczną do biblioteki GLX, zawarty w pliku `opengl32.lib`. Prototypy tych procedur i związane z nimi makrodefinicje znajdują się w pliku nagłówkowym `wglext.h`.

GLEW (`libGLEW.so`) — biblioteka z zadeklarowanymi zmiennymi wskaźnikowymi do procedur OpenGL-a, ma za zadanie nadać tym zmiennym odpowiednie wartości. Jej plik nagłówkowy ma nazwę `glew.h`, do aplikacji włączamy go *zamiast* plików nagłówkowych biblioteki GL. Osobiście nie polecam używania tej biblioteki, mimo że jest najłatwiejsza do zainstalowania. Zamiast niej lepiej jest używać biblioteki gl3w albo glad.

gl3w (`gl3w.o`, `libgl3w.so`; `gl3w.obj`) — alternatywa dla GLEW, plik obiektowy do bezpośredniego dołączenia do aplikacji lub biblioteka dołączana dynamicznie, z plikiem nagłówkowym `gl3w.h`.

[8] Dla celów mnemotechnicznych możemy przyjąć, że litera ta oznacza vskaźnik.

[9] W rozdziale 30 słowo kanwa oznacza tylko wewnętrzny bufor obrazu.

glad (`glad.o`, `libglad.so`; `glad.obj`) — jeszcze inna biblioteka dająca dostęp do procedur OpenGL-a za pomocą zadeklarowanych w niej zmiennych wskaźnikowych, bardziej rozbudowana niż biblioteka gl3w. Jej plik nagłówkowy ma nazwę `glad.h` i jeśli używamy tej biblioteki, to plik ten zastępuje plik nagłówkowy biblioteki GL[10].

FreeGLUT (`libglut.so`, `libglut.a`; `freeglut.dll`, `freeglut.lib`) — biblioteka opisana w następym rozdziale; jej zadaniem jest umożliwienie pisania aplikacji niezależnych od systemu operacyjnego i systemu okien. Dyrektywę włączającą jej plik nagłówkowy `freeglut.h` należy w aplikacjach nowego OpenGL-a poprzedzić włączeniem plików nagłówkowych biblioteki GL *albo* biblioteki GLEW *albo* gl3w *albo* glad.

GLFW (`libglfw.so`, `libglfw.a`; `glfw3.dll`, `glfw3.lib`) — inna (też opisana w następnym rozdziale) biblioteka ukrywająca kod zależny od systemu operacyjnego i systemu okien, nowsza i nowocześniejsza niż FreeGLUT. Jej plik nagłówkowy ma nazwę `glfw3.h`.

Do uruchomienia wersji D–K aplikacji opisanej w drugiej części kursu potrzebna jest też biblioteka TIFF (`libtiff.so`; `tiff.lib`), niezwiązana z OpenGL-em.

W tej książce *nie będzie* dokładnego opisu procedur OpenGL-a z wyszczególnieniem *wszystkich* informacji na temat parametrów, ich możliwych wartości itp., nie będzie też wszystkich szczegółów i niuansów języka GLSL. Przepisywanie pełnej dokumentacji, która liczy wiele setek stron, nie ma sensu. Szczegółowych informacji można (i w pewnym momencie warto) poszukać w tej dokumentacji, dostępnej w Internecie na stronach Khronos Group, na przykład [1], [3], [5] lub [7].

W książce będą natomiast podane przykłady *zastosowania* najważniejszych procedur OpenGL-a *w aplikacjach* — z opisem, co dokładnie dane procedury tam robią, dlaczego, i w szczególności jak te procedury współpracują ze sobą. Czytelnika, który zapoznał się z zastosowaniem procedury w którejś z opisanych dalej aplikacji zachęcam, *po zrozumieniu*, jaką rolę ta procedura spełnia w aplikacji, do zajrzenia na stronę [7], wyszukania tej procedury i przeczytania opisu, w tym opisu innych niż użyte w aplikacji wartości parametrów i możliwych skutków ich użycia. Zachęcam też do samodzielnych eksperymentów, w tym do modyfikowania aplikacji i patrzenia, co się ~~zepsu~~ zmieniło.

[10] Ale oczywiście nie zastępuje on pliku `glx.h`, nawet jeśli biblioteka GL zawiera także procedury z biblioteki GLX.

3

Otoczenie OpenGL-a

Aplikacja graficzna zazwyczaj nie tylko wyświetla obraz, ale także prowadzi dialog z użytkownikiem, przyjmując jego polecenia wydawane za pomocą urządzeń wejściowych (np. klawiatury, myszy, dżojstika). Ponadto aplikacja wyświetla obrazy na ekranie, na który jednocześnie inne programy wyprowadzają wyniki swoich działań; od tego jest system operacyjny i system okien, aby aplikacje nie przeszkadzały sobie nawzajem (lub nawet współpracowały, komunikując się między sobą) i aby otrzymywały komunikaty o działaniach użytkownika zgodnie z jego intencjami.

Niezbędnym elementem działającej aplikacji jest **kontekst OpenGL-a**. Jest to struktura danych zawierająca dane opisujące stan OpenGL-a (w tym wspominane wcześniej przełączniki) oraz dane aplikacji — bufory z opisami wierzchołków, zmienne jednolite, tekstury, programy szaderów itd. Kontekst składa się z dwóch części; jedna z nich znajduje się w pamięci CPU, a druga GPU. Sposób tworzenia kontekstu jest zależny od środowiska, w którym aplikacja ma działać. Inne instrukcje muszą być w tym celu wykonane przez aplikację systemu Unix (lub Linux), a inne w systemie Windows[1]. Najprostszym sposobem poradzenia sobie z tym fragmentem aplikacji jest użycie biblioteki **GLUT** (albo **FreeGLUT**) albo **GLFW**; biblioteki te, skompilowane dla różnych środowisk, ukrywają kod specyficzny dla środowiska.

Aby ułatwić utrzymanie porządku w kodzie podczas rozbudowywania aplikacji, warto procedury „okienkowe", tj. realizujące interfejs użytkownika (w tym tworzenie okien i obsługę komunikatów otrzymywanych od systemu okien) i procedury „graficzne", konstruujące obiekty do narysowania i wykonujące obrazy, zapisać w osobnych plikach źródłowych. Opisane w kolejnych czterech podrozdziałach szkielety aplikacji wywołują cztery procedury graficzne: `InitMyWorld`, która przygotowuje do pracy programy szaderów i reprezentacje obiektów, `ResizeMyWorld`, która zawiadamia część graficzną aplikacji o zmianie szerokości lub wysokości okna, `RedrawMyWorld`, która wykonuje obraz i `DeleteMyWorld`, która ma posprzątać, gdy użytkownik postanowił zatrzymać aplikację[2]. Procedury te są niezależne

[1] Po wprowadzeniu w 2014 r. własnościowego API o nazwie Metal firma Apple zaniechała implementowania OpenGL-a i Vulkana na swoim sprzęcie. Pozostaje tylko ubolewać nad tą decyzją.

[2] Ten interfejs między częściami graficzną a okienkową w miarę potrzeb będziemy rozszerzać.

od środowiska, tj. takie same w aplikacjach bibliotek FreeGLUT i GLFW oraz w natywnych aplikacjach systemów X Window i Windows, a ich prototypy są umieszczone w pliku `myheader.h`. Z kolei plik `utilities.h` zawiera prototypy procedur pomocniczych, identycznych dla wszystkich aplikacji opisanych w tej książce.

Wszystkie cztery aplikacje „szkieletowe" otwierają jedno okno gotowe do wyświetlania grafiki za pomocą OpenGL-a i realizują tę samą minimalną interakcję z użytkownikiem, reagując poprawnie na zmianę wielkości okna (która powoduje wykonanie nowego obrazu o odpowiedniej wielkości) oraz zatrzymując się po naciśnięciu klawisza `Esc`. Odpowiednio je rozszerzając, można dołączyć obsługę dżojstika i reagowanie na upływ czasu, można też utworzyć więcej okien.

Reagowaniem na komunikaty otrzymywane od systemu okien zajmują się procedury napisane przez autora aplikacji. Należy dbać o to, aby czas przetwarzania każdego komunikatu był jak najkrótszy. Jeśli dowolne obliczenie zabiera więcej niż 1/10 s lub nawet 1/30 s, to procedura przetwarzania komunikatu nie powinna wykonywać całego tego obliczenia, bo to utrudni korzystanie z aplikacji. Obliczenie trzeba albo podzielić na „krótkie etapy" (i wykonywać je podczas przetwarzania kolejnych komunikatów), albo zlecić je innemu wątkowi (który po zakończeniu obliczeń zawiadomi, że wynik jest gotowy). Ale ta książka nie jest podręcznikiem programowania aplikacji wielowątkowych.

3.1. Biblioteka FreeGLUT

Biblioteka GLUT (*The OpenGL Utility Toolkit*) była rozwijana w latach 1994–1998; jej autorem jest Mark Kilgard, pracujący wtedy w firmie Silicon Graphics. Biblioteka ta ma na celu uniezależnienie interakcyjnej aplikacji od środowiska. Projekt ten nie jest już rozwijany, a ponadto nie jest to wolne oprogramowanie. Ale istnieją jego ogólnodostępne i nadal rozwijane zamienniki. Jednym z nich jest biblioteka FreeGLUT (autor: Paweł W. Olszta, początek projektu w 1999 r.), która dokładnie odtwarza funkcjonalność oryginalnego GLUT-a i ma rozszerzenia dostosowujące projekt do nowego OpenGL-a. Opis procedur oryginalnego GLUT-a jest w dokumencie [13], a uzupełnienia FreeGLUT-a można znaleźć na stronie [14].

Makrodefinicje w pliku nagłówkowym biblioteki FreeGLUT mają nazwy zaczynające się od przedrostka `GLUT`, wszystkie zaś procedury mają na początku nazwy przedrostek `glut`. Jeśli dyrektywa `#include <GL/freeglut.h>` *nie jest* poprzedzona włączeniem pliku `glad.h`, `gl3w.h`, `GL/glew.h` albo oryginalnych plików nagłówkowych OpenGL-a, to kompilacja pliku `GL/freeglut.h` spowoduje włączenie plików `GL/gl.h` i `GL/glu.h` z prototypami procedur *starego* OpenGL-a.

Plik nagłówkowy `openglheader.h` i procedura `GetGLProcAddresses` są przedstawione w rozdziale 2. Jeśli nie chcemy używać biblioteki glad, gl3w ani GLEW i nie mamy możliwości dołączenia procedur nowego OpenGL-a bezpośrednio, to mamy do dyspozycji procedurę `glutGetProcAddress` (która wywołuje po kryjomu odpowiednią procedurę podającą adres potrzebnego podprogramu, taką jak `glXGetProcAddress`).

Na listingu 3.1 jest przedstawiony szkielet aplikacji biblioteki FreeGLUT. Procedura `main` wykonuje dwie instrukcje: wywołuje procedurę `Initialise`, której zadaniem jest przygoto-

Listing 3.1. Szkielet aplikacji FreeGLUT-a

C

```
#include <stdlib.h>
#include <stdio.h>
#include "openglheader.h"  /* najpierw ten, listing 2.1 */
#include <GL/freeglut.h>  /* potem ten */

#include "utilities.h"
#include "myheader.h"

int WindowHandle;

void ReshapeFunc ( int width, int height )
{
  ResizeMyWorld ( width, height );
} /*ReshapeFunc*/

void DisplayFunc ( void )
{
  RedrawMyWorld ();
  glutSwapBuffers ();
} /*DisplayFunc*/

void Cleanup ( void )
{
  DeleteMyWorld ();
  glutDestroyWindow ( WindowHandle );
} /*Cleanup*/

void KeyboardFunc ( unsigned char charcode, int x, int y )
{
  switch ( charcode ) {
case 0x1B:  /* <Esc> */
    Cleanup ();
    glutLeaveMainLoop ();
    break;
default:
    break;
  }
} /*KeyboardFunc*/

void SpecialKeyFunc ( int key, int x, int y ) { .... }
void MouseFunc ( int button, int state, int x, int y ) { .... }
void MotionFunc ( int x, int y ) { .... }
void JoystickFunc ( unsigned int buttonmask, int x, int y, int z ) { .... }
void TimerFunc ( int value ) { .... }
void IdleFunc ( void ) { .... }
```

```
46:
47: void Initialise ( int argc, char *argv[] )
48: {
49:   glutInit ( &argc, argv );
50:   glutInitContextVersion ( 2, 1 );
51:   glutInitContextFlags ( GLUT_FORWARD_COMPATIBLE );
52:   glutInitContextProfile ( GLUT_CORE_PROFILE );
53:   glutSetOption ( GLUT_ACTION_ON_WINDOW_CLOSE,
54:                   GLUT_ACTION_GLUTMAINLOOP_RETURNS );
55:   glutInitWindowSize ( 480, 360 );
56:   glutInitDisplayMode ( GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA );
57:   WindowHandle = glutCreateWindow ( "Aplikacja FreeGLUT-a" );
58:   if ( WindowHandle < 1 )
59:     ExitOnError ( "Could not create a window." );  /* listing 4.2 */
60:   GetGLProcAddresses ( 4, 2 );                     /* listing 2.2 */
61:   glutReshapeFunc ( ReshapeFunc );
62:   glutDisplayFunc ( DisplayFunc );
63:   glutKeyboardFunc ( KeyboardFunc );
64:   glutSpecialFunc ( SpecialKeyFunc );
65:   glutMouseFunc ( MouseFunc );
66:   glutMotionFunc ( MotionFunc );
67:   /*glutJoystickFunc ( JoystickFunc, 16 );*/
68:   /*glutTimerFunc ( 0, TimerFunc, 0 );*/
69:   /*glutIdleFunc ( IdleFunc );*/
70:   InitMyWorld ( argc, argv, 480, 360 );
71: } /*Initialise*/
72:
73: int main ( int argc, char *argv[] )
74: {
75:   Initialise ( argc, argv );
76:   glutMainLoop ();
77:   exit ( 0 );
78: } /*main*/
```

wanie programu do pracy, w tym utworzenie okna i zarejestrowanie odpowiednich procedur obsługi zdarzeń dla tego okna oraz przygotowanie szaderów i danych do rysowania, a następnie procedurę `glutMainLoop`, która działa aż do momentu zatrzymania programu; procedura ta wywołuje zarejestrowane procedury w odpowiedzi na zdarzenia wejściowe i zdarzenia spowodowane przez aplikację.

Instrukcje w liniach 49–54 przygotowują struktury danych FreeGLUT-a do pracy. Procedura `glutInit` inicjalizuje (z uwzględnieniem parametrów wywołania programu) wewnętrzne struktury danych FreeGLUT-a.

W zasadzie procedura `glutInitContextVersion` powinna być wywołana z parametrami na przykład 4, 2, aby przekazać FreeGLUT-owi informację, że ta aplikacja jest napisana zgodnie ze specyfikacją OpenGL 4.2. Ale z niejasnych dla mnie powodów coś się tam psuje i dlatego trzeba dać parametry 2 i 1, aby program dał się uruchomić i poprawnie działał. Ale

to nie szkodzi, procedura `glutInitContextFlags` przyjmuje w imieniu FreeGLUT-a deklarację, że chcemy, aby ta aplikacja działała także z implementacjami OpenGL-a zgodnymi z późniejszymi wersjami, w tym 4.2.

Procedura `glutInitContextProfile` przyjmuje od aplikacji zapewnienie, że nie będą używane procedury starego OpenGL-a (gdybyśmy chcieli, to trzeba podać parametr `GLUT_COMPATIBILITY_PROFILE`, ale nie róbmy tego).

Procedura `glutSetOption` wprowadza informacje o pożądanym zachowaniu procedur FreeGLUT-a — w rozpatrywanym tu przykładzie jest żądanie, aby aplikacja mogła spowodować powrót z procedury `glutMainLoop`[3], co umożliwi wykonanie dodatkowych instrukcji przez procedurę `main`.

W liniach 55–57 tworzymy okno. Procedura `glutInitWindowSize` określa jego wymiary początkowe (w pikselach, z pominięciem dołączanej przez menedżera okien ramki).

Parametr procedury `glutInitDisplayMode` określa potrzebne aplikacji zasoby OpenGL-a. W tym przykładzie aplikacja domaga się trybu RGBA, tzn. takiego, w którym składowe koloru piksela są przechowywane bezpośrednio w pikselu — w nowym OpenGL-u to jest jedyna dopuszczalna możliwość (w starym OpenGL-u była też możliwość używania trybu z paletą, wartość piksela była indeksem do palety, czyli tablicy przechowującej kolory do wyświetlenia na ekranie). Ponadto maska `GLUT_DEPTH` deklaruje zapotrzebowanie na bufor głębokości (do wyznaczania widoczności obiektów na obrazie), a maska `GLUT_DOUBLE` wybiera **podwójne buforowanie** — dla okna będą utworzone dwa bufory obrazu. W dowolnej chwili zawartość jednego z nich jest widoczna w oknie, a rysowanie odbywa się w drugim buforze; po zakończeniu rysowania bufory są zamieniane rolami, dzięki czemu obraz widoczny w oknie zawsze jest ukończony.

Po opisanych wyżej przygotowaniach można wywołać procedurę `glutCreateWindow`, z parametrem — napisem, który ma być wyświetlony na ramce okna. Wartość zwrócona przez tę procedurę jest identyfikatorem okna, tj. pewną liczbą dodatnią. Jeśli nie udało się utworzyć okna (bo np. zażądaliśmy niedostępnych zasobów), to procedura `glutCreateWindow` przekaże wartość `0`, co powinno spowodować zatrzymanie programu i próbę wyjaśnienia przez jego autora, co poszło nie tak. Bardziej skomplikowane aplikacje mogą tworzyć wiele okien i/lub podokien (za pomocą procedury `glutCreateWindow` albo `glutCreateSubwindow`), z których każde otrzyma swój unikatowy identyfikator.

Okno lub podokno zaraz po utworzeniu, a także w chwili wywołania przez FreeGLUT-a zarejestrowanej procedury obsługi komunikatu o zdarzeniu dla tego okna lub po wywołaniu (w trakcie obsługi takiego komunikatu dla innego okna) procedury `glutSetWindow` z parametrem — identyfikatorem okna — jest **oknem aktywnym**. W szczególności procedury rejestrujące przywiązują procedury obsługi zdarzeń do okna aktywnego. W naszym przykładzie procedury wywołane w liniach 61–66 przywiążą procedury obsługi komunikatów o zdarzeniach dla okna utworzonego przez instrukcję w linii 57, które jest jedynym oknem w tej aplikacji i pozostanie aktywne przez cały czas jej działania.

[3] Należy w tym celu wywołać procedurę `glutLeaveMainLoop`. Takiej możliwości nie było w oryginalnej bibliotece GLUT; aby zakończyć działanie programu, jedna z zarejestrowanych procedur aplikacji musiała wykonać `exit`.

Procedura `ReshapeFunc` rejestrowana w linii 61 przez `glutReshapeFunc` zostanie wywołana (przez `glutMainLoop`) po utworzeniu okna i po każdej zmianie jego szerokości lub wysokości. Jej parametry opisują nowe wymiary okna, a jej zadaniem jest obliczenie macierzy rzutowania dla okna w nowym kształcie.

Procedura `DisplayFunc` rejestrowana w linii 62 przez `glutDisplayFunc` zostanie wywołana po każdej zmianie wymiarów okna (ale wcześniej będzie wywołana procedura zarejestrowana przez `glutReshapeFunc`) oraz po jego odsłonięciu na ekranie (gdy użytkownik odsunął lub zamknął okno zasłaniające) i po zawiadomieniu FreeGLUT-a przez aplikację, że zawartość okna zmieniła się i należy wyświetlić nowy obraz. Do takiego zawiadamiania służą procedury `glutPostRedisplay` (wywołuje procedury rysowania dla wszystkich okien) lub `glutPostWindowRedisplay` (wywołuje procedurę rysowania jednego okna lub podokna, o identyfikatorze podanym jako parametr).

Uwaga: Jedyną procedurą wykonującą rysowanie w oknie ma prawo być procedura zarejestrowana dla tego okna przez `glutDisplayFunc` i tylko FreeGLUT ma prawo wywoływać tę procedurę. Jeśli komunikat o zdarzeniu (np. naciśnięciu klawisza) spowodował konieczność wykonania nowego obrazu, to procedura obsługi tego zdarzenia ma przygotować odpowiednio zmienioną reprezentację obiektów na obrazie i wywołać procedurę `glutPostRedisplay` albo `glutPostWindowRedisplay`.

Procedura `KeyboardFunc` rejestrowana w linii 63 przez `glutKeyboardFunc` jest wywoływana po naciśnięciu klawisza, gdy kursor znajduje się w oknie. Jej parametry to kod klawisza (kod ASCII skojarzonego z tym klawiszem znaku) oraz współrzędne położenia kursora w oknie. Zwróćmy uwagę, że układ współrzędnych używany przez FreeGLUT-a ma początek w *górnym lewym* narożniku okna; większą współrzędną y mają punkty położone niżej. To jest orientacja odwrotna do orientacji układu stosowanego przez OpenGL-a (którego początek jest w dolnym lewym narożniku okna). Jednostki długości osi x i y odpowiadają szerokości i wysokości jednego piksela.

W linii 64 jest rejestrowana procedura `SpecialKeyFunc`, która będzie wywołana po naciśnięciu klawisza specjalnego, takiego jak strzałka lub `F1`, ..., `F12`. Naciśnięty klawisz jest identyfikowany przez parametr `key`, którego wartość liczbowa jest w programie zastępowana przez czytelniejszą nazwę odpowiedniej makrodefinicji znajdującej się w pliku nagłówkowym `GL/freeglut_std.h`.

Procedura obsługi komunikatu o zdarzeniu spowodowanym przez klawiaturę lub mysz może wywołać procedurę `glutGetModifiers`, która podaje stan **modyfikatorów**, tj. klawiszy `Shift`, `Ctrl` oraz `Alt`, aby zareagować odpowiednio. Naciśnięcie każdego z tych klawiszy powoduje także wywołanie procedury zarejestrowanej przez `glutSpecialFunc`, ale system (lub menedżer) okien, z którym współpracuje FreeGLUT, może „nie przepuścić" tych komunikatów.

Procedury `glutMouseFunc` i `glutMotionFunc` wywoływane w liniach 65 i 66 rejestrują odpowiednio procedury, które będą wywołane po naciśnięciu lub zwolnieniu przycisku myszy oraz po przesunięciu myszy (czyli zmianie położenia kursora). Parametry `x` i `y` tych pro-

cedur opisują nowe położenie kursora w oknie. Parametry `button` i `state` identyfikują przycisk, którego dotyczy zdarzenie i jego stan po zmianie; ich możliwe wartości to odpowiednio `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` lub `GLUT_RIGHT_BUTTON` oraz `GLUT_DOWN` lub `GLUT_UP`.

Podprogram zarejestrowany przez `glutMotionFunc` jest wywoływany wtedy, gdy któryś z przycisków myszy jest naciśnięty. Jeśli aplikacja ma otrzymywać komunikaty o przesunięciach kursora także wtedy, gdy żaden przycisk nie jest naciśnięty, to powinna zarejestrować odpowiedni podprogram przy użyciu procedury `glutPassiveMotionFunc`.

Obecnie produkowane myszy są wyposażone w rolkę, w niektórych modelach pełniącą także obowiązki środkowego guzika. Informację o obróceniu rolki FreeGLUT przekazuje, wywołując procedurę zarejestrowaną przez `glutMouseFunc`, przy czym wywołuje ją dwukrotnie: za pierwszym razem parametr `state` ma wartość `GLUT_DOWN`, a za drugim `GLUT_UP`. Parametr `button` określa kierunek obrotu rolki, przy czym w pliku `GL/freeglut_std.h` nie ma odpowiednich nazw symbolicznych[4]. Obrót rolki „od użytkownika" jest sygnalizowany przez parametr `button` o wartości `3`, a jeśli nastąpił obrót w drugą stronę, to parametr `button` ma wartość `4`.

Jeśli aplikacja ma odbierać komunikaty wejściowe od dżojstika, to należy zarejestrować procedurę `JoystickFunc` (w linii 67). Drugi parametr procedury `glutJoystickFunc` określa częstotliwość, z jaką FreeGLUT „odpytuje" dżojstik i wywołuje zarejestrowaną procedurę, a dokładniej, odstęp (w milisekundach) między kolejnymi wywołaniami tej procedury. Jej pierwszy parametr jest maską bitową; poszczególne bity określają stan przycisków dżojstika (`1`, jeśli przycisk jest przyciśnięty, `0` w przeciwnym razie). Parametry `x`, `y`, `z` opisują położenie drążka dżojstika, tj. kąty jego obrotów wokół osi x, y, z. Miarą każdego kąta jest liczba całkowita z przedziału $[-1000, 1000]$, który odpowiada maksymalnemu zakresowi każdego z tych obrotów. FreeGLUT ma też procedury umożliwiające badanie, ile dżojstik ma osi i przycisków i konfigurowanie go do szczególnych potrzeb aplikacji[5].

Procedury `glutTimerFunc` używamy, jeśli chcemy, aby rejestrowana przez nią procedura została wywołana po upływie określonego czasu. Pierwszy parametr określa ilość czasu podaną w milisekundach, drugi to adres rejestrowanej procedury, a trzeci parametr może być użyty do przekazania tej procedurze (przez jej parametr) informacji o miejscu w programie, w którym została zarejestrowana. Jeśli chcemy, aby aplikacja wykonywała jakąś czynność co sekundę, to powinniśmy w procedurze `TimerFunc` wywołać procedurę `glutTimerFunc` z pierwszym parametrem równym `1000`, a drugim — równym adresowi procedury `TimerFunc`, a potem wykonać instrukcje realizujące tę czynność.

Wreszcie, procedura `glutIdleFunc` rejestruje podprogram, który ma być natychmiast wywołany za każdym razem, gdy FreeGLUT nie ma żadnych innych zadań do wykonania. Taki podprogram przydaje się w animacji; na podstawie zegara systemowego powinien

[4] Myszy z rolką zostały wynalezione po powstaniu biblioteki GLUT, a być może także FreeGLUT.

[5] Zwróćmy uwagę, że o ile procedury obsługi zdarzeń związanych z myszą i klawiaturą są wywoływane po wystąpieniu odpowiedniego zdarzenia (np. po naciśnięciu klawisza lub przesunięciu myszy), o tyle procedura związana z dżojstikiem jest wywoływana co chwila, niezależnie od tego, czy stan dżojstika się zmienił.

określić stan (np. położenie, kształt, kolor, oświetlenie) obiektów do narysowania w danej chwili i spowodować narysowanie tych obiektów (przez wywołanie `glutPostRedisplay` lub `glutPostWindowRedisplay`). Procedury `glutTimerFunc` i `glutIdleFunc` rejestrują odpowiednie podprogramy dla całej aplikacji, a nie dla jednego okna.

„Wyrejestrowanie" podprogramu obsługi komunikatów następuje przez wywołanie odpowiedniej procedury rejestrującej z parametrem `NULL`, można też w każdej chwili zarejestrować inny podprogram, który przejmie obsługę komunikatów w aplikacji.

Ostatnia instrukcja w procedurze inicjalizacji (linia 70) wywołuje procedurę, która ma umieścić w kontekście OpenGL-a potrzebne programy szaderów i utworzyć początkową reprezentację obiektów do narysowania, co obejmuje inicjalizację struktur danych aplikacji w pamięci CPU oraz utworzenie potrzebnych obiektów (np. VAO, UBO, zobacz rozdział 1) i tekstur w pamięci GPU.

Aby zakończyć działanie programu, jedna z procedur obsługi komunikatów powinna wywołać procedurę `glutLeaveMainLoop`, co spowoduje powrót z procedury `glutMainLoop`. Przedtem jednak wypada wywołać procedurę `Cleanup`, której zadaniem jest zwolnienie zasobów, które aplikacja zarezerwowała na swoje potrzeby; oprócz bloków pamięci zarezerwowanych bezpośrednio przez aplikację za pomocą procedury `malloc` są to zasoby OpenGL-a i systemu okien, w tym programy szaderów, bufory, tekstury, kontekst OpenGL-a i okna. Do zamknięcia okna służy procedura `glutDestroyWindow`, która oprócz okna likwiduje także związany z nim kontekst OpenGL-a.

3.1.1. FreeGLUT — procedury zarządzania oknami

Biblioteka FreeGLUT zawiera wiele procedur wykonujących podstawowe działania na oknach, takie jak zmiany wymiarów i położenia oraz kolejności w stosie okien (mającej wpływ na to, które okno zasłania inne okno, gdy oba zajmują ten sam obszar na ekranie). Odsyłając Czytelników po szczegóły do dokumentacji [13], wymienię tu najważniejsze z tych procedur. Działają one na oknie aktywnym aplikacji.

Procedury `glutPositionWindow` i `glutReshapeWindow` nadają oknu odpowiednio położenie (górnego lewego narożnika) i wymiary na ekranie — zatem aplikacja może je zmieniać z własnej inicjatywy, nie czekając na działania użytkownika. Jeśli okno aktywne jest podoknem, to skutkiem działania każdej z tych procedur jest zmiana położenia lub wymiarów podokna *w obrębie nadrzędnego okna* aplikacji.

Procedura `glutFullScreen` usuwa ramkę i nadaje oknu wymiary całego ekranu. Aby wycofać okno z trybu pełnoekranowego, należy wywołać procedurę `glutReshapeWindow`.

Procedury `glutPopWindow` i `glutPushWindow` zmieniają położenie okna w stosie okien na ekranie lub położenie podokna w stosie okien okna nadrzędnego, odpowiednio na samą górę lub na dół stosu. Zatem, po wykonaniu procedury `glutPopWindow` okno aktywne będzie zasłaniać wszystkie inne okna zajmujące ten sam obszar na ekranie.

Procedura `glutSetWindowTitle` powoduje umieszczenie na ramce okna aktywnego napisu podanego jako parametr.

Procedury `glutHideWindow` i `glutIconifyWindow` „zdejmują" okno aktywne z ekranu, przy czym druga z nich umieszcza na ekranie ikonę, czyli obrazek, którego pstryknięcie powoduje ponowne umieszczenie okna na ekranie. Aplikacja może zrobić to samo, wywołując procedurę `glutShowWindow`.

Procedura `glutSetCursor` służy do wybierania kształtu kursora dla okna aktywnego; parametr (typu `int`) powinien być jedną ze stałych symbolicznych (np. `GLUT_CURSOR_RIGHT_ARROW`) określających ten kształt; stałe te są zdefiniowane w pliku nagłówkowym `GL/freeglut_std.h`.

3.1.2. FreeGLUT — uwagi dodatkowe

Domyślnie dla każdego okna i podokna FreeGLUT tworzy nowy kontekst OpenGL-a. Programy, bufory i inne obiekty obecne w kontekście są niewidoczne dla innych kontekstów, co sprawia, że jeśli mamy użyć takich samych obiektów (np. programów) do rysowania w różnych oknach, to musimy je utworzyć osobno dla każdego okna. Można chcieć, aby okna miały wspólny kontekst. W tym celu należy przed tworzeniem okien wykonać instrukcję

```
glutSetOption ( GLUT_RENDERING_CONTEXT, GLUT_USE_CURRENT_CONTEXT );
```

W czasie gdy okno jest aktywne, jest również aktywny związany z nim kontekst. Jeśli zatem aplikacja najpierw utworzy kilka okien z osobnymi kontekstami, a potem będzie instalować programy szaderów, tworzyć bufory itd., to *przed* utworzeniem tych obiektów dla każdego okna należy je uaktywnić (co uaktywnia związany z tym oknem kontekst) — wywołując procedurę `glutSetWindow` z parametrem identyfikującym to okno.

Animacja w czasie rzeczywistym wymaga informacji na temat czasu, który upłynął od ustalonej chwili, na przykład od uruchomienia aplikacji lub od pewnego zdarzenia w trakcie jej działania. Biblioteka FreeGLUT nie zawiera procedur dokonujących pomiarów czasu, z wyjątkiem procedury `glutTimerFunc`, która raczej nie wystarczy do przeprowadzenia animacji. W punkcie 3.5.1 jest opisany zestaw procedur realizujących użyteczny stoper, ukrywających wywołania specyficznych dla systemu operacyjnego procedur odczytujących zegar.

Biblioteka FreeGLUT zawiera komplet procedur umożliwiających tworzenie tzw. wyskakujących menu (*popup menus*), które można „podłączyć" do dowolnie wybranego przycisku myszy; menu, początkowo niewidoczne, po naciśnięciu tego przycisku pojawia się w miejscu, w którym znajduje się kursor. Takie menu zawiera prostokątne guziki z podanymi przez aplikację napisami (nazwami poleceń); pstryknięcie guzika powoduje wywołanie zarejestrowanej przez aplikację procedury obsługi poleceń wydawanych za pośrednictwem menu. Niestety, ten mechanizm nie nadaje się do użycia w aplikacji *nowego* OpenGL-a, bo wprawdzie procedury tworzenia i obsługi menu są i działają, ale po naciśnięciu guzika menu nie pojawia się na ekranie, co zmusza użytkownika do działania „na ślepo"[6].

[6] Dawno temu pisałem aplikacje starego OpenGL-a i FreeGLUT-a z działającym menu. Zamierzałem opisać tu procedury tworzenia menu bardziej szczegółowo i po wykonaniu eksperymentu przekonałem się, że już nie ma o czym pisać.

3.2. Biblioteka GLFW

Biblioteka GLFW występuje w dwóch istotnie różnych wersjach: 2.? (np. 2.7) i 3.? (np. 3.1, 3.2, 3.3); zajmiemy się tą drugą wersją. Choć można przy jej użyciu pisać aplikacje w stylu bardziej „niskopoziomowym" niż aplikacje FreeGLUT-a, na listingu 3.2 zamieściłem szkielet możliwie podobny do tego z listingu 3.1. Uwagi na temat „niskopoziomowego" stylu pisania aplikacji są podane na końcu tego podrozdziału, a pełną (znakomitą) dokumentację można znaleźć na stronie [15].

Przed włączeniem pliku nagłówkowego `GLFW/glfw3.h` biblioteki GLFW (linia 4) należy włączyć plik nagłówkowy odpowiedniej biblioteki „wprowadzającej" (tj. gl3w, glad lub GLEW[7]). Zamiast identyfikatorów okien, będących liczbami całkowitymi, biblioteka GLFW używa wskaźników do struktur typu `GLFWwindow`; razem z każdym oknem otwieranym przez aplikację biblioteka tworzy taką strukturę, której budowa jest dla aplikacji niewidoczna.

W bibliotece FreeGLUT działania takie jak rysowanie, zmiany wymiarów itp. są wykonywane na oknie aktywnym; chcąc je zmienić (np. po to, aby w trakcie reakcji na komunikat skierowany do okna móc zmienić obrazek w innym oknie), należy wywołać procedurę `glutSetWindow` z identyfikatorem okna, które ma być aktywne, podanym jako parametr. Dla odmiany, procedury w bibliotece GLFW i w jej aplikacjach mają wskaźnik do struktury `GLFWwindow`, pełniący obowiązki identyfikatora okna, podawany jako parametr.

W procedurze `main` (linie 90–96) mamy, jak w poprzednim szkielecie, wywołania kolejno procedury inicjalizacji `Initialise` i pętli komunikatów `MessageLoop`, a potem sprzątania `Cleanup` (która na końcu powinna wywołać procedurę `glfwTerminate`, sprzątającą po procedurach biblioteki GLFW). Przyjrzyjmy się inicjalizacji.

W linii 61 procedura `glfwSetErrorCallback` rejestruje procedurę obsługi błędów; zostanie ona wywołana, jeśli któraś z procedur biblioteki GLFW wykryje sytuację błędną, przekazując liczbowy kod błędu i tekst go opisujący. Procedura zapisana w liniach 11–14 wypisze ten tekst i przerwie działanie aplikacji.

Procedura wywołana w linii 62 inicjalizuje wewnętrzne struktury danych biblioteki GLFW. Jeśli to się nie powiedzie, to trudno, koniec, kropka i `exit`.

Podobnie, aplikacja zostanie zatrzymana, jeśli nie powiedzie się tworzenie nowego okna w liniach 64–65. Parametry procedury `glfwCreateWindow` określają odpowiednio wymiary (szerokość i wysokość) okna w pikselach, napis do wyświetlenia na ramce okna, wskaźnik monitora i wskaźnik okna współdzielącego kontekst. Wskaźnik monitora, jeśli jest pusty (`NULL`), zgłasza potrzebę utworzenia zwykłego okna przez system okien. Jeśli nie jest pusty, to rysowanie ma się odbywać w trybie pełnoekranowym na wskazanym monitorze (po szczegóły na ten temat odsyłam do dokumentacji). Wskaźnik okna współdzielącego, jeśli nie jest pusty (`NULL`), oznacza, że okno ma mieć wspólny kontekst OpenGL-a z innym, wcześniej utworzonym oknem.

Przed utworzeniem okna można podać rozmaite parametry wpływające na jego własności, wywołując procedurę `glfwWindowHint`. W podanym szkielecie używane są domyślne wartości tych parametrów, bez potrzeby nie warto ich zmieniać.

[7] Tej ostatniej nie polecam.

Listing 3.2. Szkielet aplikacji biblioteki GLFW

C

```
#include <stdlib.h>
#include <stdio.h>
#include "openglheader.h"
#include <GLFW/glfw3.h>
#include "utilities.h"
#include "myheader.h"

GLFWwindow *mywindow;
char       redraw;

void myGLFWErrorHandler ( int error, const char *description )
{
  ExitOnError ( description );
} /*myGLFWErrorHandler*/

void ReshapeFunc ( GLFWwindow *win, int width, int height )
{
  ResizeMyWorld ( width, height );
  redraw = true;
} /*ReshapeFunc*/

void Redraw ( GLFWwindow *win )
{
  RedrawMyWorld ();
  redraw = false;
  glfwSwapBuffers ( win );
} /*Redraw*/

void KeyboardFunc ( GLFWwindow *win, int key, int scancode,
                    int action, int mods )
{
  if ( action == GLFW_PRESS || action == GLFW_REPEAT )
    switch ( key ) {
  case GLFW_KEY_ESCAPE:
      glfwSetWindowShouldClose ( win, 1 );
      break;
  default:
      break;
    }
} /*KeyboardFunc*/

void DisplayFunc ( GLFWwindow *win )
{
  redraw = true;
} /*DisplayFunc*/
```

```
46:
47: void CharFunc ( GLFWwindow *win, unsigned int charcode ) { .... }
48: void MouseFunc ( GLFWwindow *win, int button, int action, int mods )
49: { .... }
50: void MotionFunc ( GLFWwindow *win, double x, double y ) { .... }
51:
52: void Cleanup ( void )
53: {
54:   DeleteMyWorld ();
55:   glfwDestroyWindow ( mywindow );
56:   glfwTerminate ();
57: } /*Cleanup*/
58:
59: void Initialise ( int argc, char **argv )
60: {
61:   glfwSetErrorCallback ( myGLFWErrorHandler );
62:   if ( !glfwInit () )
63:     ExitOnError ( "glfwInit failed." );
64:   if ( !(mywindow = glfwCreateWindow ( 480, 360,
65:                                        "Aplikacja GLFW", NULL, NULL )) ) {
66:     glfwTerminate ();
67:     ExitOnError ( "glfwCreateWindow failed." );
68:   }
69:   glfwMakeContextCurrent ( mywindow );
70:   GetGLProcAddresses ( 4, 2 );
71:   glfwSetFramebufferSizeCallback ( mywindow, ReshapeFunc );
72:   glfwSetWindowRefreshCallback ( mywindow, DisplayFunc );
73:   glfwSetKeyCallback ( mywindow, KeyboardFunc );
74:   glfwSetCharCallback ( mywindow, CharFunc );
75:   glfwSetMouseButtonCallback ( mywindow, MouseFunc );
76:   glfwSetCursorPosCallback ( mywindow, MotionFunc );
77:   InitMyWorld ( argc, argv, 480, 360 );
78:   ReshapeFunc ( mywindow, 480, 360 );
79: } /*Initialise*/
80:
81: void MessageLoop ( void )
82: {
83:   while ( !glfwWindowShouldClose ( mywindow ) ) {
84:     glfwWaitEvents ();
85:     if ( redraw )
86:       Redraw ();
87:   }
88: } /*MessageLoop*/
89:
90: int main ( int argc, char **argv )
91: {
92:   Initialise ( argc, argv );
```

```
93:   MessageLoop ();
94:   Cleanup ();
95:   exit ( 0 );
96: } /*main*/
```

Biblioteka GLFW wymaga, aby przed rysowaniem lub przesyłaniem dowolnych danych (w tym opisu obiektów do rysowania oraz szaderów) aplikacja jawnie wybierała aktywny kontekst OpenGL-a (w bibliotece FreeGLUT robi to procedura `glutSetWindow`). Dla aplikacji z jednym oknem, na przykład takiej o szkielecie z listingu 3.2, odbywa się to „raz na zawsze" w linii 69.

Po określeniu bieżącego kontekstu należy uzyskać dostęp do procedur OpenGL-a — następuje wywołanie procedury inicjalizacji biblioteki wprowadzającej, na przykład gl3w albo glad, za pośrednictwem procedury `GetGLProcAddresses`.

W liniach 71–76 rejestrowane są procedury obsługi komunikatów o zdarzeniach dla okna. Pozwoliłem sobie nadać im takie same lub podobne nazwy jak w szkielecie aplikacji FreeGLUT-a na listingu 3.1, ale trzeba zwrócić uwagę na inne parametry tych procedur. Pierwszy parametr każdej z nich (linie 16–50) to wskaźnik struktury `GLFWwindow`. Poza tym procedura `ReshapeFunc` jest taka jak w aplikacji FreeGLUT-a. Natomiast zadaniem procedury `DisplayFunc` jest tylko zapamiętanie informacji, że zawartość okna należy narysować — przez przypisanie niezerowej wartości zmiennej `redraw`[8]. Procedura rysująca `RedrawMyWorld` jest wywoływana w pętli komunikatów aplikacji; po wykonaniu obrazu wartość zmiennej `redraw` jest zmieniana na `false` (czyli `0`). Procedury obsługi komunikatów (np. o przesunięciu myszy lub naciśnięciu klawisza) powinny przypisać niezerową wartość zmiennej `redraw`, jeśli obraz w oknie przestał być aktualny i należy narysować nowy obraz.

Parametry wywoływanej po przesunięciu myszy procedury `MotionFunc`, opisujące położenie kursora, są typu `double`, a nie `int`, ale układ współrzędnych w oknie jest taki sam jak w bibliotece FreeGLUT, tj. z początkiem w górnym lewym narożniku okna i z osią *y* skierowaną do dołu. W bibliotece FreeGLUT procedury obsługi komunikatów o przesunięciach myszy, gdy któryś przycisk jest naciśnięty i gdy żaden nie jest, są rejestrowane osobno (można oczywiście zarejestrować tę samą procedurę dla obu przypadków). Biblioteka GLFW umożliwia zarejestrowanie tylko jednej takiej procedury, wywoływanej po każdym przesunięciu myszy niezależnie od stanu przycisków.

Procedura zarejestrowana za pomocą procedury `glfwSetMouseButtonCallback` jest wywoływana po naciśnięciu lub zwolnieniu przycisku myszy (którym może być rolka), ale komunikaty o obracaniu rolki są przekazywane za pomocą procedury rejestrowanej przez

[8] Identyfikatory `false` i `true` to słowa kluczowe, oznaczające wartości typu `bool` w językach C++ i GLSL, ale nie w języku C (w którym nie ma typu `bool`). Dlatego do reprezentowania wartości boolowskich w aplikacji w C będziemy używać zmiennych typu `char`, którym będą przypisywane wartości `0` lub `1`, nazwane dla wygody `false` i `true`. W plikach nagłówkowych OpenGL-a są też nazwy `GL_FALSE` i `GL_TRUE`, których będziemy używać w wywołaniach procedur OpenGL-a z parametrami boolowskimi. W wersji 3.3 biblioteki GLFW stałe `0` i `1` otrzymały jeszcze nazwy `GLFW_FALSE` i `GLFW_TRUE`, a w pliku nagłówkowym biblioteki X11 są identyfikatory `False` i `True`. Co za bałagan …

procedurę glfwSetScrollCallback. Parametr y procedury ScrollFunc ma wartość 1, jeśli użytkownik obrócił rolkę „od siebie" oraz -1, jeśli „do siebie". Parametr x w obu przypadkach ma wartość 0.

Procedura CharFunc, rejestrowana dla okna przez wywołanie glfwSetCharCallback, jest wywoływana po napisaniu dowolnego znaku na klawiaturze.

Procedura KeyFunc, rejestrowana przez wywołanie glfwSetKeyCallback, jest wywoływana po naciśnięciu lub zwolnieniu dowolnego klawisza, w tym klawisza specjalnego, na przykład strzałki lub F1, ..., F12. Po naciśnięciu klawisza „zwykłego" znaku (np. litery) procedura ta jest również wywoływana, ale jej parametry nie podają napisanego znaku, tylko identyfikują kod klawisza.

Uwaga: Choć pewne klawisze specjalne, na przykład Esc, są związane ze znakami, które mają kody ASCII, po ich naciśnięciu jest wywoływana tylko procedura zarejestrowana przez glfwSetKeyCallback. Nazwy makrodefinicji identyfikujących klawisze specjalne można znaleźć w pliku nagłówkowym GLFW/glfw3.h.

W linii 77 jest wywoływana procedura przygotowująca programy szaderów i obiekty do wyświetlania. W linii 78 jest wywołana procedura ReshapeFunc; wywołanie takiej procedury po utworzeniu okna FreeGLUT bierze na siebie, ale biblioteka GLFW nie, więc aplikacja musi to zrobić osobiście.

Pętla komunikatów musi być oprogramowana w aplikacji (nie ma odpowiednika procedury glutMainLoop). Najprostsza procedura pętli komunikatów jest podana w liniach 81–88. Aplikacja będzie czekać na zdarzenie i po jego wystąpieniu zostanie wywołana odpowiednia zarejestrowana procedura. Jeśli przypisze ona wartość niezerową zmiennej redraw, to procedura RedrawMyWorld narysuje, co trzeba. Warunek zakończenia pętli to niezerowa wartość powrotna procedury glfwWindowShouldClose; można spowodować podanie takiej wartości, wykonując instrukcję

```
glfwSetWindowShouldClose ( mywindow, true );
```

na przykład w odpowiedzi na naciśnięcie klawisza Esc.

Są dwie najważniejsze procedury, które można wywoływać w pętli komunikatów aplikacji biblioteki GLFW: glfwWaitEvents i glfwPollEvents. Pierwsza z nich czeka na zdarzenie (naciśnięcie przycisku, przesunięcie myszy itd.) i wykonuje powrót po wystąpieniu tego zdarzenia. Druga z tych procedur wraca natychmiast, także wtedy, gdy nic się nie wydarzyło. Dzięki tej procedurze można stworzyć mechanizm analogiczny do tego, który udostępnia procedura glutIdleFunc. Sposób, jak to zrobić, jest pokazany na listingu 3.3.

Jeśli zmienna idlefunc ma wartość NULL, to aplikacja będzie czekać na zdarzenia bez angażowania procesora (w tym czasie może on wykonywać instrukcje innych uruchomionych równolegle programów albo oszczędzać energię elektryczną). Przypisanie tej zmiennej adresu dowolnej procedury powoduje wywoływanie tej procedury, gdy tylko aplikacja nie ma nic innego do roboty — do odwołania, tj. do ponownego nadania tej zmiennej wartości NULL[9].

[9]Ten kod nie nadaje się do użycia w programie wielowątkowym, w którym przypisanie wartości zmiennej idlefunc może wykonać inny wątek niż ten, który wykonuje procedurę MessageLoop.

Listing 3.3. Alternatywna procedura pętli komunikatów

C

```
1: static void (*idlefunc)(void) = NULL;
2:
3: void SetIdleFunc ( void(*IdleFunc)(void) )
4: {
5:   idlefunc = IdleFunc;
6: } /*SetIdleFunc*/
7:
8: void MessageLoop ( void )
9: {
10:   do {
11:     if ( idlefunc ) {
12:       idlefunc ();
13:       glfwPollEvents ();
14:     }
15:     else
16:       glfwWaitEvents ();
17:     if ( redraw )
18:       Redraw ();
19:   } while ( !glfwWindowShouldClose ( mywindow ) );
20: } /*MessageLoop*/
```

Alternatywą dla użycia procedur obsługi komunikatów rejestrowanych dla okna, tak jak w szkielecie aplikacji na listingu 3.2, jest używanie procedur, które „odpytują" system okien (ukryty przed aplikacją) na temat zdarzeń, które nastąpiły. Jest cała seria procedur biblioteki GLFW, które podają bieżące wymiary okna, położenie kursora, ostatnio napisane na klawiaturze znaki itd. Zastosowanie tych procedur to właśnie jest ten „niskopoziomowy" styl programowania aplikacji biblioteki GLFW.

3.2.1. GLFW — obsługa dżojstika

Postaci informacji od dżojstika w API bibliotek GLFW i FreeGLUT są różne, inny jest też sposób ich przekazywania. Zamiast rejestrować procedurę, która będzie co chwila wywoływana z parametrami niosącymi te informację, aplikacja GLFW musi sama odpowiednio często „pytać" o stan i samopoczucie dżojstika. Do komputera może być podłączony więcej niż jeden; poszczególne urządzenia mają identyfikatory `GLFW_JOYSTICK_1`, ..., `GLFW_JOYSTICK_16`. Procedura `glfwJoystickPresent` bada, czy w chwili jej wywołania wskazany dżojstik jest podłączony (co może się zmieniać podczas działania aplikacji). Procedury `glfwGetJoystickAxes` i `glfwGetJoystickButtons` podają wskaźniki tablic, w których zapisany jest stan dżojstika. Drugi parametr każdej z tych procedur jest adresem zmiennej, w której procedura zapisuje liczbę osi lub przycisków dżojstika (czyli długość tablicy). Aplikacja nie ma prawa przypisywać wartości do tych tablic ani zwalniać zajmowanej przez nie pamięci. Jeśli dżojstik w międzyczasie został odłączony, procedury `glfwGetJoystickAxes` i `glfwGetJoystickButtons` przekazują wskaźnik pusty.

Listing 3.4. Czytanie stanu dżojstika w aplikacji GLFW

C

```
1: void PollJoystick ( int joy )
2: {
3:   const float         *axes;
4:   const unsigned char *buttons;
5:   int                 count, i;
6:
7:   if ( (axes = glfwGetJoystickAxes ( joy, &count )) )
8:     for ( i = 0; i < count; i++ )
9:       printf ( "%6.3f ", axes[i] );
10:  if ( (buttons = glfwGetJoystickButtons ( joy, &count )) )
11:    for ( i = 0; i < count; i++ )
12:      printf ( "%1d ", buttons[i] );
13:  printf ( "\n" );
14: } /*PollJoystick*/
```

Przykład procedury, która odczytuje (i wypisuje w terminalu) stan dżojstika, jest pokazany na listingu 3.4. Parametr `joy` ma być jednym z identyfikatorów, na przykład `GLFW_JOYSTICK_1`. Kąty wychylenia drążka są reprezentowane w tablicy `axes` przez liczby zmiennopozycyjne z przedziału $[-1, 1]$. Jeśli i-ty przycisk jest przyciśnięty, to i-ty element tablicy `buttons` ma wartość `1`, a w przeciwnym razie `0`.

3.2.2. GLFW — procedury zarządzania oknami i uwagi dodatkowe

Okno aplikacji może być przesunięte, powiększone lub zmniejszone albo zlikwidowane przez użytkownika (za pośrednictwem menedżera okien) lub z inicjatywy aplikacji. Biblioteka GLFW zawiera procedury, których wywołanie powoduje odpowiednie działania, a także procedury, które rejestrują podprogramy wywoływane w celu zawiadomienia aplikacji, że użytkownik wydał odpowiednie polecenie — na przykład procedurę `glfwSetWindowSizeCallback`. Do zmiany wymiarów i położenia okna służą procedury `glfwSetWindowSize` i `glfwSetWindowPos`. Aplikacja może w każdej chwili „spytać" o wymiary i położenie okna, wywołując procedury `glfwGetWindowSize` i `glfwGetWindowPos`.

Uwaga: Jednostkami osi układu współrzędnych używanego przez system okien zazwyczaj są, ale *nie muszą* być, szerokość i wysokość piksela. Procedura `glViewport` wymaga podawania wymiarów klatki (zobacz podrozdz. 6.1) w pikselach, dlatego należy je określać na podstawie parametrów podprogramu zarejestrowanego przez procedurę `glfwSetFramebufferSizeCallback`, tak jak na listingu 3.2, a nie `glfwSetWindowSizeCallback`, choć to na wielu komputerach może działać poprawnie[10].

Procedury `glfwHideWindow` i `glfwIconifyWindow` zdejmują okno z ekranu, przy czym ta ostatnia zostawia na ekranie ikonę umożliwiającą użytkownikowi przywołanie okna z po-

[10] W pierwszym wydaniu książki używałem `glfwSetWindowSizeCallback`. Na wielu komputerach to działało poprawnie. Później doczytałem tę informację w dokumentacji.

wrotem na ekran. Okno zdjęte z ekranu może być przywołane przez aplikację za pomocą procedury `glfwShowWindow`.

Procedura `glfwMaximizeWindow` pozbawia okno ramki i nadaje mu wielkość całego ekranu. Do wyjścia z trybu pełnoekranowego służy procedura `glfwRestoreWindow`.

Tytuł okna, umieszczony na jego ramce, jest początkowo nadawany przez procedurę `glfwCreateWindow`, ale może być w dowolnej chwili zmieniony przez procedurę `glfwSetWindowTitle` — jeśli na przykład aplikacja chce w ten sposób podawać nazwy obiektów widocznych w oknie w danej chwili.

Procedura `glfwFocusWindow` powoduje przesunięcie okna na wierzch stosu okien na ekranie (tak, że staje się widoczne w całości, ewentualnie zasłaniając inne okna) i przechwytuje wejście z klawiatury i myszy (szczegółowe zachowanie okna zależy od menedżera okien). Nie ma natomiast procedury, która spycha okno „na samo dno", takiej jak `glutPushWindow` w bibliotece FreeGLUT.

Procedura `glfwSetWindowUserPointer` określa kształt, który kursor ma przybierać, gdy znajduje się w oknie. Kształt ten jest opisany przez dane binarne określone przez aplikację — nie ma tu wyboru jednej z wielu nazwanych (przez makrodefinicje w pliku nagłówkowym) możliwości, takiego jak w bibliotece FreeGLUT.

Procedura `glfwDestroyWindow` służy do likwidacji okien, które przestały być potrzebne.

Nie ma w bibliotece GLFW możliwości tworzenia podokien. Aplikacja, która potrzebuje podzielić okno na podobszary (np. widok sceny i menu z wihajstrami albo widok z okna samolotu i deska z przyrządami pokładowymi), musi dokonać odpowiedniego podziału sama. Efekt równoważny rysowaniu w podoknach może być osiągnięty przez odpowiednie ustawianie klatki (*viewport* — o tym w rozdziale 6) albo użycie bufora maski lub testu nożyczek (p. 18.4.1).

Procedura podająca adresy procedur OpenGL-a o nazwach wskazanych przez parametr ma nazwę `glfwGetProcAddress`; zależnie od systemu okien, dla którego została skompilowana, wywołuje odpowiednią procedurę w tym systemie, na przykład `glXGetProcAddress`.

Biblioteka GLFW może być wykorzystana w programach wielowątkowych, ale w zasadzie tylko jeden wątek obliczeniowy może wywoływać procedury OpenGL-a (nazwijmy go wątkiem graficznym). Inne wątki aplikacji mogą z nim współdziałać. Na przykład wywołanie procedury `glfwPostEmptyEvent` przez inny wątek spowoduje powrót z procedury `glfwWaitEvents` wywołanej przez wątek graficzny i czekającej na jakiekolwiek wydarzenie.

Biblioteka GLFW zawiera zestaw procedur do pomiaru upływu czasu, w tym `glfwGetTimerFrequency`, `glfwSetTime`, `glfwGetTime` i `glfwGetTimerValue`, o których można dowiedzieć się wszystkiego, studiując dokumentację [15] i ich kod źródłowy. Ale wszystkie opisane w tym kursie aplikacje korzystają z procedur opisanych w podrozdziale 3.5.1.

Biblioteka GLFW może też być użyta w aplikacjach standardu Vulkan. W czasie pisania tej książki trwają też prace nad przystosowaniem jej do systemów okien używających protokołu Wayland[11]. Może jest więc bardziej przyszłościowa niż FreeGLUT.

[11] Jak dotąd powstała możliwość pisania aplikacji Waylanda w standardzie OpenGL ES, ciąg dalszy zależy od producentów sprzętu i sterowników.

3.3. System X Window i biblioteka GLX

Aplikacja systemu X Window ma takie same elementy jak aplikacje FreeGLUT-a i biblioteki GLFW, tj. procedurę inicjalizacji, która wykonuje niezbędne przygotowania (tworzy co najmniej jedno okno, tworzy kontekst OpenGL-a, kompiluje szadery i przygotowuje struktury danych reprezentujące obiekty do narysowania), pętlę komunikatów (która obsługuje zdarzenia spowodowane działaniami użytkownika, w szczególności wykonuje obrazy w oknie, gdy jest to potrzebne) i procedurę sprzątającą na końcu. Opisany tu szkielet aplikacji korzysta tylko ze struktur i procedur opisanych w plikach nagłówkowych `X11/Xlib.h` i `X11/Xutil.h` biblioteki X11 oraz z biblioteki GLX[12] z plikiem nagłówkowym `GL/glx.h`. Ten interfejs jest znacznie bardziej niskopoziomowy i bardziej pracochłonny dla autora aplikacji w porównaniu z interfejsami udostępnionymi przez wcześniej opisane biblioteki, ale jest bardziej elastyczny i daje znacznie większą (bo pełną) kontrolę nad środowiskiem, w którym aplikacja działa. W szczególności umożliwia tworzenie okien o innym niż prostokątny kształcie (korzystając z rozszerzeń systemu X Window) i mieszanie grafiki tworzonej przez OpenGL-a z grafiką dwuwymiarową otrzymywaną za pomocą procedur rysujących systemu X Window — ale akurat od takiego mieszania głowa może bardzo rozboleć; jeśli trzeba w aplikacji korzystać zarówno z OpenGL-a, jak i procedur grafiki systemu X Window, które umożliwiają znacznie łatwiejsze stworzenie graficznego interfejsu użytkownika (*GUI — graphical user interface*), to najlepiej jest utworzyć w aplikacji okno z podoknami i w każdym podoknie tworzyć obrazy tylko za pomocą OpenGL-a albo tylko X Window. Szkielet głównej części aplikacji jest pokazany na listingu 3.5. W tym szkielecie jedynie dyrektywy `#include` w liniach `5` i `6` (oraz pominięte treści procedur) sprawiają, że jest to aplikacja OpenGL-a, a nie tylko systemu X Window.

W pliku `initglxctx.h` znajdują się prototypy procedur potrzebnych w aplikacji OpenGL-a działającej w systemie X Window, w tym procedury konstruującej kontekst OpenGL-a.

Procedura `Initialise`, wywołana natychmiast po uruchomieniu aplikacji, przygotowuje wszystko, czego trzeba do pracy. Pierwsze zadanie polega na nawiązaniu łączności z serwerem X Window; wykonuje to pokazana na listingu 3.6 procedura `InitXServerConnection`. Jeśli komunikacja nie zaiskrzyła, to aplikacja nie ma nic więcej do powiedzenia. W przeciwnym razie opis połączenia zostaje zapamiętany w zmiennej `xdisplay`, która będzie przekazywana jako parametr większości procedur z biblioteki X11. Procedura `InitXServerConnection` udostępnia także opis tzw. wizualu, czyli struktury danych opisującej odwzorowanie wartości pikseli na kolory wyświetlane na ekranie. Wskaźnik tego opisu jest przypisywany zmiennej `xvii` procedury tworzącej okno.

Procedura `InitMyGLXWindow` (listing 3.7) tworzy okno oraz kontekst OpenGL-a dla tego okna. Identyfikator okna i wskaźnik struktury kontekstu zostają przypisane zmiennym globalnym `xmywin` i `glxcontext`. Procedura `InitMyWorld` przygotowuje programy szaderów i tworzy obiekty składające się na scenę do narysowania.

Po powrocie z procedury `Initialise` następuje wywołanie procedury `MessageLoop`, która realizuje pętlę komunikatów. Procedura `XNextEvent` czeka na komunikat, a po jego nadejściu wpisuje informacje na temat zdarzenia, które spowodowało komunikat do zmien-

[12] być może stanowiącej część biblioteki `libGL.so`

Listing 3.5. Szkielet aplikacji OpenGL-a w systemie X Window

C

```
#include <stdlib.h>
#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include "openglheader.h"
#include <GL/glx.h>

#include "utilities.h"
#include "initglxctx.h"
#include "myheader.h"

Display    xdisplay;
Window     xmywin;
GLXContext glxcontext;
XEvent     xevent;
int        win_width, win_height;
char       terminate;

void InitMyGLXWindow ( int argc, char **argv, int major, int minor,
                       int width, int height ) { .... }  /* listing 3.7 */
void DeleteMyGLXWindow ( void ) { .... }                 /* listing 3.7 */

void Initialise ( int argc, char **argv )
{
  InitXServerConnection ( argc, argv, false );     /* listing 3.6 */
  InitMyGLXWindow ( argc, argv, 4, 2, 480, 360 );
  InitMyWorld ( argc, argv, 480, 360 );
} /*Initialise*/

void MyWinExpose ( void )
{
  RedrawMyWorld ();
  glXSwapBuffers ( xdisplay, xmywin );
} /*MyWinExpose*/

void MyWinConfigureNotify ( int width, int height )
{
  ResizeMyWorld ( win_width = width, win_height = height );
  PostExposeEvent ( xmywin, width, height );  /* listing 3.8 */
} /*MyWinConfigureNotify*/

void MyWinClientMessage ( XClientMessageEvent *xclient ) { .... }
void MyWinButtonPress ( int button, int x, int y ) { .... }
void MyWinButtonRelease ( int button, int x, int y ) { .... }
void MyWinMotionNotify ( int x, int y ) { .... }
```

```
46: void MyWinKeyPress ( unsigned int state, unsigned int key ) { .... }
47:
48: void MyWinChar ( char charcode )
49: {
50:   switch ( charcode ) {
51: case 0x1B:  /* <Esc> */
52:     terminate = true;
53:     break;
54: default:
55:     break;
56:   }
57: } /*MyWinChar*/
58:
59: void MyWinMessageProc ( void )
60: {
61:   char   charcode;
62:   KeySym ks;
63:
64:   switch ( xevent.xany.type ) {
65: case Expose:
66:     if ( xevent.xexpose.count == 0 )
67:       MyWinExpose ();
68:     break;
69: case ConfigureNotify:
70:     MyWinConfigureNotify ( xevent.xconfigure.width,
71:                            xevent.xconfigure.height );
72:     break;
73: case ButtonPress:
74:     MyWinButtonPress ( xevent.xbutton.button,
75:                        xevent.xbutton.x, xevent.xbutton.y );
76:     break;
77: case ButtonRelease:
78:     MyWinButtonRelease ( xevent.xbutton.button,
79:                          xevent.xbutton.x, xevent.xbutton.y );
80:     break;
81: case MotionNotify:
82:     MyWinMotionNotify ( xevent.xmotion.x, xevent.xmotion.y );
83:     break;
84: case KeyPress:
85:     MyWinKeyPress ( xevent.xkey.state, xevent.xkey.keycode );
86:     XLookupString ( &xevent.xkey, &charcode, 1, &ks, NULL );
87:     MyWinChar ( charcode );
88:     break;
89: case ClientMessage:
90:     MyWinClientMessage ( &xevent.xclient );
91:     break;
92: default:
```

```
93:     break;
94:   }
95: } /*MyWinMessageProc*/
96:
97: void MessageLoop ( void )
98: {
99:   terminate = 0;
100:  do {
101:    XNextEvent ( xdisplay, &xevent );
102:    if ( xevent.xany.window == xmywin )
103:      MyWinMessageProc ();
104:  } while ( !terminate );
105: } /*MessageLoop*/
106:
107: void Cleanup ( void )
108: {
109:  DeleteMyWorld ();
110:  DeleteMyGLXWindow ();
111:  XCloseDisplay ( xdisplay );
112: } /*Cleanup*/
113:
114: int main ( int argc, char **argv )
115: {
116:  Initialise ( argc, argv );
117:  MessageLoop ();
118:  Cleanup ();
119:  exit ( 0 );
120: } /*main*/
```

nej `xevent` i kończy swoje działanie. Zmienna ta jest unią 31 struktur opisujących zdarzenia; podstawowa biblioteka X11 obsługuje 35 rodzajów komunikatów[13] spowodowanych przez zdarzenia. Pola każdej struktury zawierają informacje odpowiednie dla danego zdarzenia. Dokładny opis tych struktur można znaleźć w dokumentacji [11] i na stronach podręcznika systemowego wyświetlanych przez polecenie `man`.

Po otrzymaniu komunikatu w linii 102 następuje sprawdzenie, czy komunikat jest związany z danym oknem; w aplikacji z jednym oknem jest ono niepotrzebne (choć nieszkodliwe), ale pisząc ten kod, chciałem pokazać sposób zidentyfikowania adresata w aplikacji, która otworzyła więcej niż jedno okno. Procedura `MyWinMessageProc` wywołana w linii 103 reaguje na komunikaty wysłane do jedynego okna naszej aplikacji, wywołując procedury obsługi poszczególnych komunikatów.

Treść procedury `MyWinMessageProc` jest pokazana w liniach 59–95; instrukcja przełącznika wybiera działanie odpowiednie do rodzaju zdarzenia. W tym przykładzie obsługiwane są zdarzenia **odsłonięcia okna** (komunikat `Expose`), którego zawartość trzeba

[13] Dodatkowe rodzaje komunikatów są wprowadzone w rozszerzeniach, m.in. umożliwiających współpracę z OpenGL-em, ale także tworzenie okien innych niż prostokątne lub związanych z wygaszaczem ekranu.

narysować, **zmiany wymiarów** (`ConfigureNotify`), po której trzeba dostosować reprezentację obrazu do nowych wymiarów okna, **naciśnięcie** i **zwolnienie przycisku myszy** (`ButtonPress`, `ButtonRelease`), **przesunięcie kursora** (`MotionNotify`), **naciśnięcie klawisza** (`KeyPress`) i **wiadomość z zewnątrz** (`ClientMessage`), którą aplikacja może wysłać sama do siebie[14], na przykład w celu wykonywania obliczeń wtedy, gdy działania użytkownika nie powodują wysyłania innych komunikatów (to może pełnić rolę podobną do mechanizmu udostępnionego przez procedurę `glutIdleFunc` biblioteki FreeGLUT). Dla każdego z tych komunikatów jest wywoływana procedura, której autor (mam nadzieję) wiedział, co ta procedura ma robić.

Informację o tym, który przycisk myszy został naciśnięty lub zwolniony, procedura `XNextEvent` zapisuje w polu `xbutton.button` zmiennej `event`; plik nagłówkowy systemu X Window zawiera makrodefinicje `Button1`, `Button2` i `Button3`, które identyfikują odpowiednio lewy, środkowy i prawy przycisk, a także `Button4` i `Button5`, które służą do sygnalizacji obracania rolki. Obrót rolki o jedną pozycję powoduje wysłanie pary komunikatów `ButtonPress` i `ButtonRelease`[15].

Odsłonięcie części okna może spowodować wysłanie serii komunikatów `Expose`, jeśli odsłonięty obszar nie jest jednym prostokątem (obszar taki jest dzielony na prostokąty, każdy komunikat zawiera informację o wymiarach i położeniu jednego z nich). Warunek w linii 66 powoduje rysowanie tylko dla ostatniego komunikatu w takiej serii — system X Window wyświetli całą widoczną część obrazu w oknie.

Po zakończeniu rysowania trzeba wywołać procedurę `glXSwapBuffers`, która realizuje podwójne buforowanie — rysowanie odbywało się w niewidocznym buforze, podczas gdy w oknie było widać poprzedni obraz; po zakończeniu rysowania gotowy obraz staje się widoczny.

Uwaga: System X Window zezwala na rysowanie w oknie w trakcie obsługi dowolnego komunikatu, nie tylko komunikatu `Expose`; nie ma więc konieczności używania procedur takich jak `glutPostWindowRedisplay`[16]. Ale można to robić, posługując się na przykład procedurą `PostExposeEvent` pokazaną na listingu 3.8.

Procedura `MyWinMessageLoop` kończy działanie, jeśli dowolna procedura obsługi komunikatu przypisze niezerową wartość zmiennej `terminate`. Procedura `main` wywołuje wtedy procedurę `Cleanup`, której zadaniem jest posprzątanie.

3.3.1. Tworzenie kontekstu OpenGL-a dla aplikacji X Window

Uwaga: Opisane w tym i w następnym podrozdziale procedury tworzenia kontekstu OpenGL-a dla natywnych aplikacji systemów X Window i Windows są maksymalnie uproszczone,

[14] Wiadomość do aplikacji X Window może też wysłać inny działający w tym samym czasie program, musi tylko znać identyfikator okna, do którego ta wiadomość ma trafić.

[15] Rozwiązanie zrealizowane w bibliotece FreeGLUT działa podobnie.

[16] Ściślej biorąc, polecenia rysowania wydawane przez aplikację systemu X Window są pakowane do kolejki. System wyjmuje z kolejki i wykonuje te polecenia w dogodnych dla siebie momentach. Ale implementacja OpenGL-a może zawierać tzw. DRI (*direct rendering interface*), który rysowanie przeprowadza z pominięciem kolejki X Window.

bo ich zadaniem jest tylko umożliwienie uruchomienia aplikacji przedstawianych w moim kursie OpenGL-a, GLSL-a i podstaw grafiki komputerowej i chciałem, by były jak najkrótsze. „Poważniejsze" projekty, przeznaczone do działania na wielu różnych komputerach, powinny poświęcić większą uwagę dostępnym w danym systemie zasobom i odpowiednio się do niego dostosować. Realizując taki projekt, można wzorować się na procedurach z bibliotek FreeGLUT lub GLFW.

Na listingu 3.6 są pokazane dwie procedury. Zadaniem pierwszej z nich jest nawiązanie łączności aplikacji z serwerem X Window, otrzymanie informacji o numerze ekranu i oknie nadrzędnym (np. pulpicie), na którym okna aplikacji mają być otwarte. Uzyskane przez tę procedurę informacje są zapamiętywane w zmiennych globalnych `xdisplay`, `xscreen` i `xrootwin`. Jeśli aplikacja ma być wielowątkowa, to zostaje najpierw wywołana procedura `XInitThreads`. Dodatkowo procedura zapamiętuje dwa atomy potrzebne do komunikacji z menedżerem okien[17]. **Atom** jest to liczba całkowita używana jako identyfikator typu danych opisujących tzw. **własność** (ang. *property*). Pewien zbiór „gotowych" atomów (zobacz plik `X11/Xatom.h`) służy do identyfikowania własności (na przykład okien) zdefiniowanych przez system X Window. Wspomniane dwa atomy umożliwiają zatrzymanie aplikacji za pomocą menedżera okien, co jest opisane dalej.

Listing 3.6. Procedury nawiązania komunikacji i tworzenia kontekstu OpenGL-a w X Window

C

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include "openglheader.h"
#include <GL/glx.h>

#include "initglxctx.h"
#include "utilities.h"

typedef GLXContext (*PFNGLXCREATECONTEXTATTRIBSARBPROC)
                      ( Display *dpy, GLXFBConfig config,
                        GLXContext share_context, Bool direct,
                        const int *attrib_list );

int    xscreen;
Window xrootwin;
Atom   WMProtocols, DeleteWindow;

void InitXServerConnection ( int argc, char **argv, char threads )
{
```

[17] W systemie X Window przedrostek `WM_` jest skrótem słów *window manager*, a w systemie Windows oznacza *window message*.

```
24:   if ( threads ) XInitThreads ();
25:   if ( !(xdisplay = XOpenDisplay ( "" )) )
26:     ExitOnError ( "InitXServerConnection 0" );
27:   xscreen = DefaultScreen ( xdisplay );
28:   xrootwin = RootWindow ( xdisplay, xscreen );
29:   WMProtocols = XInternAtom ( xdisplay, "WM_PROTOCOLS", False );
30:   DeleteWindow = XInternAtom ( xdisplay, "WM_DELETE_WINDOW", False );
31: } /*InitXServerConnection*/
32:
33: void InitGLXContext ( int major, int minor, int flags,
34:                       int *visattr, XVisualInfo **xvii, GLXContext *context )
35: {
36:   PFNGLXCREATECONTEXTATTRIBSARBPROC glXCreateContextAttribsARB;
37:   int nelements;
38:   int vattr[] =
39:     { GLX_RGBA,         True,
40:       GLX_DOUBLEBUFFER, True,
41:       GLX_RED_SIZE,     1,
42:       GLX_GREEN_SIZE,   1,
43:       GLX_BLUE_SIZE,    1,
44:       GLX_DEPTH_SIZE,  24,
45:       None };
46:   int ctxattr[] =
47:     { GLX_CONTEXT_MAJOR_VERSION_ARB, 0,
48:       GLX_CONTEXT_MINOR_VERSION_ARB, 0,
49:       GLX_CONTEXT_PROFILE_MASK_ARB, GLX_CONTEXT_CORE_PROFILE_BIT_ARB,
50:       GLX_CONTEXT_FLAGS_ARB, 0,
51:       None };
52:   GLXFBConfig *glxfbc;
53:
54:   if ( !visattr )
55:     visattr = vattr;
56:   if ( !(*xvii = glXChooseVisual ( xdisplay, 0, visattr )) )
57:     ExitOnError ( "InitGLXContext 0" );
58:   if ( (glXCreateContextAttribsARB =
59:         (PFNGLXCREATECONTEXTATTRIBSARBPROC)glXGetProcAddress
60:             ( (const GLubyte*)"glXCreateContextAttribsARB" )) ) {
61:     ctxattr[1] = major;  ctxattr[3] = minor;  ctxattr[7] = flags;
62:     glxfbc = glXChooseFBConfig ( xdisplay, xscreen, 0, &nelements );
63:     if ( !(*context = glXCreateContextAttribsARB ( xdisplay,
64:                                           *glxfbc, 0, 1, ctxattr )) )
65:       ExitOnError ( "InitGLXContext 1" );
66:     XFree ( glxfbc );
67:   }
68:   else
69:     ExitOnError ( "InitGLXContext 2" );
70: } /*InitGLXContext*/
```

Procedura `InitGLXContext` tworzy kontekst OpenGL-a przy użyciu procedur z biblioteki GLX[18]. Pierwszym krokiem jest wybór z zasobów lokalnie zainstalowanego systemu X Window odpowiedniego **wizualu** (*visual*). Jest to struktura danych opisująca odwzorowanie wartości pikseli w oknie na wyświetlane na ekranie kolory; zależnie od tzw. **klasy wizualu** odwzorowanie to może wykorzystywać paletę lub prowadzić do otrzymania obrazów czarno-białych[19], wizual może mieć też specjalne własności, na przykład umożliwiające osiągnięcie stereoskopii. Aplikacja, wywołując procedurę `InitGLXContext`, może podać jako trzeci parametr tablicę potrzebnych jej atrybutów wizualu, może też podać wskaźnik pusty, co spowoduje użycie tablicy `vattr` z domyślnym zestawem atrybutów (linie 39–45). Zestaw ten określa zapotrzebowanie na wizual, w którym wszystkie trzy składowe koloru mają co najmniej 1 bit, a ponadto jest do dyspozycji podwójny bufor obrazu i bufor głębokości o co najmniej 24 bitach na każdy piksel.

Wskaźnik opisu wybranego wizualu jest przypisywany zmiennej wskazywanej przez parametr `xvii`; wskaźnik ten jest później potrzebny w konstrukcji okien.

W linii 58 korzystamy z procedury `glXGetProcAddress` w celu otrzymania adresu procedury `glXCreateContextAttribsARB`, która konstruuje kontekst (nowego) OpenGL-a. Ta procedura realizuje rozszerzenie specyfikacji GLX 1.4[20], więc teoretycznie może jej nie być i wtedy aplikacja kapituluje.

W liniach 61–62 wybierana jest konfiguracja domyślnego tzw. **bufora ramki** (*framebuffer*), który będzie związany z oknem.

Procedura `glXCreateContextAttribsARB` konstruuje kontekst o postulowanych przez aplikację własnościach, w szczególności realizujący wersję standardu o podanym numerze, który w linii 61 został umieszczony na liście wymaganych przez aplikację atrybutów kontekstu. Parametr `context` jest wskaźnikiem zmiennej, której skonstruowany kontekst zostaje przypisany. Jeśli aplikacja ma otworzyć więcej niż jedno okno, w którym będzie wykonywać obrazy za pomocą OpenGL-a, to może przy użyciu procedury `InitGLXContext` utworzyć konteksty dla poszczególnych okien.

3.3.2. Tworzenie okna

Procedura `InitMyGLXWindow` pokazana na listingu 3.7 za pomocą procedury opisanej wcześniej tworzy kontekst OpenGL-a, a następnie okno, z którym ten kontekst będzie współpracował. Do utworzenia okna X Window oprócz identyfikatora okna nadrzędnego i opisu wizualu jest potrzebna **paleta**, tworzona w liniach 9–10.

W liniach 12–14 w (pewnych) polach zmiennej `swa` są zapamiętywane informacje wymagane przez procedurę tworzącą okno (informacje, które można podać w pozostałych polach,

[18] Jest to bardzo prosta procedura; nie gwarantuję, że na każdym komputerze zadziała poprawnie, ale zapewniam, że na co najmniej kilkunastu zadziałała.

[19] W aplikacjach nowego OpenGL-a wymagany jest wizual klasy `TrueColor`, w którym piksele mają składowe *r*, *g*, *b* wyświetlane bezpośrednio na ekranie.

[20] Specyfikacja GLX 1.4 [12], ostatnia dostępna, prawie w całości jest związana ze starym OpenGL-em. Kontekst dla nowego OpenGL-a trzeba utworzyć przy użyciu rozszerzenia. Wypada wyrazić ubolewanie, że nie jest ono w tej specyfikacji udokumentowane i że nie ma nowszej specyfikacji, w pełni dostosowanej do nowego OpenGL-a.

Listing 3.7. Procedura tworzenia okna X Window do pracy z OpenGL-em

```C
void InitMyGLXWindow ( int argc, char **argv,
                       int major, int minor, int width, int height )
{
  XSetWindowAttributes swa;
  Colormap             colormap;
  XVisualInfo          *xvii;

  InitGLXContext ( major, minor, 0, NULL, &xvii, &glxcontext );
  if ( !(xcolormap = XCreateColormap ( xdisplay, xrootwin,
                                       xvii->visual, AllocNone )) )
    ExitOnError ( "InitMyGLXWindow 0" );
  swa.colormap = xcolormap;
  swa.event_mask = ExposureMask | StructureNotifyMask| ButtonPressMask |
                   ButtonReleaseMask | PointerMotionMask | KeyPressMask ;
  xmywin = XCreateWindow ( xdisplay, xrootwin, 0, 0, width, height,
                           0, xvii->depth, InputOutput, xvii->visual,
                           CWColormap | CWEventMask, &swa );
  XFreeColormap ( xdisplay, xcolormap );
  XFree ( xvii );
  XSetWMProtocols ( xdisplay, xmywin, &DeleteWindow, 1 );
  XMapWindow ( xdisplay, xmywin );
  if ( !glXMakeCurrent ( xdisplay, xmywin, glxcontext ) )
    ExitOnError ( "InitMyGLXWindow 1" );
  GetGLProcAddresses ( major, minor );
} /*InitMyGLXWindow*/

void DeleteMyGLXWindow ( void )
{
  XDestroyWindow ( xdisplay, xmywin );
  glXDestroyContext ( xdisplay, glxcontext );
} /*DeleteMyGLXWindow*/

void MyWinClientMessage ( XClientMessageEvent *xclient )
{
  if ( xclient->message_type == WMProtocols &&
       (Atom)xclient->data.l[0] == DeleteWindow ) {
    terminate = true;
    return;
  }
} /*MyWinClientMessage*/
```

nie są koniecznie potrzebne). Pierwsze z tych pól zawiera identyfikator palety, a w drugim należy podać maskę bitową określającą komunikaty, których odbieraniem przez okno aplikacja jest zainteresowana. Na listingu są wybrane (prawie wszystkie) komunikaty obsługiwane przez procedury pokazane na listingu 3.5; nie ma tylko (nieistniejącej) maski dla komunikatów `ClientMessage`, które i tak aplikacja będzie otrzymywać.

Wywołana w liniach 15–17 procedura `XCreateWindow` tworzy okno o wymiarach i innych własnościach opisanych przez parametry; w szczególności klasa okna `InputOutput` oznacza, że to okno może przyjmować komunikaty wygenerowane przez urządzenia wejściowe i może być widoczne na ekranie. Przedostatni parametr procedury jest maską bitową określającą, które pola struktury — ostatniego parametru — mają nadane wartości. Identyfikator utworzonego okna jest przypisywany zmiennej `xmywin`. Wartość pola `depth` struktury wskazywanej przez zmienną `xvii` to postulowana liczba bitów na piksel, przekazywana jako ósmy parametr procedury `XCreateWindow`.

Procedura `XMapWindow` powoduje ukazanie się okna na ekranie. Procedura `glXMakeCurrent` uaktywnia kontekst OpenGL-a i wiąże go z oknem. Gdy kontekst jest aktywny, można uzyskać dostęp do procedur OpenGL-a za pomocą jednej z opisanych w poprzednim rozdziale bibliotek pomocniczych.

Paleta i opis wizualu po utworzeniu wszystkich okien aplikacji przestają być potrzebne i można je zlikwidować (linie 18 i 19).

Instrukcja w linii 20 uaktywnia możliwość zatrzymywania aplikacji za pomocą **wyskakującego menu** (ang. *popup menu*) obsługiwanego przez menedżera okien[21]. Jeśli użytkownik wyda polecenie zatrzymania aplikacji w ten sposób, to menedżer wyśle komunikat `ClientMessage`, który powinien zostać obsłużony w sposób pokazany w liniach 35–39; oczywiście do procedury obsługi tego komunikatu należy w miarę potrzeb dodać instrukcje odpowiednio reagujące na komunikaty z innymi wartościami pól `message_type` i `data.l`.

Na końcu działania aplikacja powinna wywołać procedurę `DeleteMyGLXWindow`, która likwiduje okno i kontekst oraz procedurę `XCloseDisplay`, która zamyka komunikację z serwerem. Po dokonaniu tego aplikacja może udać się na zasłużony spoczynek.

3.3.3. Procedury wysyłania komunikatów

Listing 3.8 przedstawia wcześniej wspomnianą procedurę `PostExposeEvent`, która może być używana w roli analogicznej do `glutPostWindowRedisplay`, oraz procedurę `PostClientMessageEvent`, która umieszcza komunikat `ClientMessage` w kolejce systemu X Window.

Pierwszy parametr procedury `PostClientMessageEvent` jest identyfikatorem okna — adresata komunikatu. Drugi parametr jest atomem. System X Window, przesyłając komunikat `ClientMessage`, nie interpretuje atomu będącego wartością pola `message_type`. Aplikacja może zatem w komunikatach `ClientMessage` używać do swoich celów dowolnych atomów, ale powinna poprawnie reagować na komunikaty z atomami zdefiniowanymi przez system, w tym przez menedżera okien. Atomy mają nazwy, będące napisami. Za pomocą procedury `XInternAtom` aplikacja może zbadać, czy atom o danej nazwie istnieje lub utworzyć nowe atomy do własnych zastosowań[22].

Parametr `type` powinien mieć wartość 8, 16 albo 32, a parametr `data` powinien wskazywać tablicę z 20 bajtami, 10 liczbami 16-bitowymi lub z pięcioma liczbami 32-bitowymi, które

[21] Biblioteki FreeGLUT i GLFW pracujące w systemie X Window dbają o ten szczegół.

[22] Trzecia aplikacja utworzy atom o nazwie `"aAnimate"`, który będzie podawać w wysyłanych do siebie komunikatach `ClientMessage` w celu zrealizowania animacji.

zostaną przesłane w komunikacie. Wysyłanie komunikatów `ClientMessage` przez aplikację do siebie jest jednym ze sposobów zapewnienia, że odpowiednia procedura aplikacji zostanie wywołana, gdy nie ma innych zadań do wykonania[23].

Jak wspomniałem wcześniej, komunikaty `ClientMessage` mogą do siebie nawzajem przesyłać różne aplikacje działające jednocześnie. W tym celu muszą się najpierw skomunikować i przesłać sobie (np. przez łącze komunikacyjne) identyfikatory okien, które mają odbierać te komunikaty. Tego tematu rozwijać tu nie będziemy, natomiast w dodatku D przedstawię pewien sposób przekazywania danych od dżojstika do aplikacji X Window za pomocą tych komunikatów.

Listing 3.8. Procedury `PostExposeEvent` i `PostClientMessageEvent`

C

```
1: void PostExposeEvent ( Window win, int width, int height )
2: {
3:   XExposeEvent ev;
4:
5:   memset ( &ev, 0, sizeof(ev) );
6:   ev.type = Expose;  ev.send_event = True;
7:   ev.display = xdisplay;  ev.window = win;
8:   ev.width = width;  ev.height = height;
9:   XSendEvent ( xdisplay, win, True, ExposureMask, (XEvent*)&ev );
10: } /*PostExposeEvent*/
11:
12: void PostClientMessageEvent ( Window win, Atom message_type,
13:                               int format, void *data )
14: {
15:   XClientMessageEvent ev;
16:
17:   memset ( &ev, 0, sizeof(ev) );
18:   ev.type = ClientMessage;  ev.send_event = True;
19:   ev.display = xdisplay;  ev.window = win;
20:   ev.message_type = message_type;  ev.format = format;
21:   if ( data )
22:     memcpy ( ev.data.b, data, 20*sizeof(char) );
23:   XSendEvent ( xdisplay, win, True, 0, (XEvent*)&ev );
24: } /*PostClientMessageEvent*/
```

3.3.4. Procedury zarządzania oknami

Procedury zarządzania oknami w bibliotekach FreeGLUT i GLFW przekazują zadaną sobie pracę systemowi okien, z którym współpracują. Jeśli systemem tym jest X Window, to procedury faktycznie wykonujące tę pracę są opisane niżej. Parametry tych procedur są opisane w specyfikacji [11] i na stronach podręcznika systemowego wyświetlanych poleceniem `man`.

[23] A ściślej, gdy komunikat `ClientMessage` znajdzie się na początku kolejki komunikatów, czyli po przetworzeniu komunikatów, które znalazły się w kolejce wcześniej.

Szczegółowe informacje o bieżących wymiarach, położeniu i innych atrybutach okna podają procedury `XGetWindowAttributes` i `XGetGeometry`. Pierwsza z nich wpisuje informacje do struktury typu `XWindowAttributes`, której adres jest parametrem, a druga do zmiennych typu `int` lub `unsigned int` wskazywanych przez osobne parametry.

Do zmieniania wymiarów okna (w tym nadawania mu wymiarów całego ekranu) służy procedura `XResizeWindow`. Zmianę położenia okna na ekranie można uzyskać za pomocą procedury `XMoveWindow`. Jednoczesna zmiana wymiarów i położenia jest wykonywana przez procedurę `XMoveResizeWindow`.

Procedura `XUnmapWindow` usuwa okno z ekranu, a procedura `XIconifyWindow` dodatkowo pozostawia na ekranie ikonę. Aby okno pojawiło się ponownie, użytkownik powinien pstryknąć ikonę, lub aplikacja powinna wywołać procedurę `XMapWindow`.

Procedura `XRaiseWindow` umieszcza okno na wierzchu stosu okien na ekranie, a procedura `XLowerWindow` spycha okno na samo dno. Jest też procedura `XRestackWindows`, która otrzymuje jako parametr tablicę identyfikatorów wielu okien i ustawia je w kolejności podanej w tej tablicy. Procedura `XSetWMName` służy do nadania lub zmiany napisu wyświetlanego na ramce okna.

Procedura `XDefineCursor` określa kształt kursora w oknie. Kształt ten może być określony przez jeden z 77 identyfikatorów „gotowych" kursorów (identyfikatory te można znaleźć w pliku `X11/cursorfont.h`) lub identyfikator kursora określonego na podstawie danych przekazanych przez aplikację.

Do likwidacji okna służy procedura `XDestroyWindow`, która likwiduje także wszystkie podokna tego okna. Osobno należy zlikwidować związany z oknem kontekst OpenGL-a (wywołując procedurę `glXDestroyContext`).

3.3.5. Sygnalizacja błędów w X Window

System X Window wykrywa dwa rodzaje błędów podczas działania aplikacji. Pierwszy ich rodzaj to wywołanie procedur z błędnymi parametrami, na przykład identyfikatorami nieistniejących okien, lub niewykonalne żądania, na przykład utworzenia obiektów niemieszczących się w dostępnej pamięci. Błędy tego rodzaju są *zazwyczaj* spowodowane przez błędy w aplikacji. Drugi rodzaj błędów jest spowodowany awariami, na przykład utratą połączenia między aplikacją a serwerem X Window. Domyślna reakcja na błąd każdego rodzaju to napisanie (w terminalu tekstowym, z którego aplikacja została wywołana) komunikatu diagnostycznego i zatrzymanie aplikacji.

Informacje o błędach, tak jak o innych zdarzeniach, są przekazywane przez komunikaty, ale aplikacja ich nie otrzyma od procedury `XNextEvent`. Aplikacja może przechwytywać komunikaty o błędach po zarejestrowaniu swoich procedur obsługi błędów. Służą do tego procedury `XSetErrorHandler` i `XSetIOErrorHandler`. Parametrem każdej z tych procedur jest adres procedury obsługi błędu, którą aplikacja zamierza zainstalować. Procedura obsługi błędów pierwszego rodzaju może wykonać powrót, po którym aplikacja będzie działać nadal (ale autor aplikacji powinien się głęboko zastanowić, co zrobił nie tak). Każdy błąd drugiego rodzaju jest tak poważny, że trzeba zakończyć działanie aplikacji. Po więcej informacji na ten temat odsyłam do dokumentacji systemu X Window.

3.4. OpenGL w aplikacji systemu Windows

Wielką pomocą w uruchamianiu wszelkich aplikacji (tj. w wyszukiwaniu i poprawianiu błędów) są **wydruki kontrolne**, czyli teksty wyprowadzane do plików `stdout` i `stderr`. Aby z tej pomocy można było korzystać w systemie Windows, trzeba główną procedurę aplikacji nazwać `main`; przykład takiej procedury jest na listingu 3.9. Rozpoczynając wykonywanie aplikacji z procedurą `main`, system Windows otworzy okno terminala[24], w którym pojawią się teksty wyprowadzane przez procedurę `printf`. Jeśli zamiast `main` aplikacja ma procedurę `WinMain` (listing 3.10), to efekty działania procedury `printf` pozostaną niewidoczne, więc dla wszelkich tekstów, które mają pojawić się na ekranie, aplikacja musi otworzyć boks dialogowy lub narysować litery w swoim oknie. Parametr `inst` procedury `WinMain` jest identyfikatorem (w terminologii Windows zwanym **uchwytem**, ang. *handle*) **instancji** aplikacji, czyli działającego procesu. Procedura `main` może go otrzymać za pomocą procedury `GetModuleHandle`.

Listing 3.9. Procedura `main` aplikacji Windows

C

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>
#include "openglheader.h"

#include "utilities.h"
#include "initwglctx.h"  /* prototypy procedur z listingu 3.16 */
#include "myheader.h"

.... /* tu procedury wywoływane przez main */

int main ( int argc, char* argv[] )
{
  Initialise ( GetModuleHandle ( 0 ), true, argc, argv );
  MessageLoop ();
  Cleanup ();
  exit ( 0 );
} /*main*/
```

Makrodefinicja `WIN32_LEAN_AND_MEAN` powoduje „odchudzenie" do rozsądnego minimum informacji pobieranych z pliku nagłówkowego `windows.h`.

[24] Aplikacje bibliotek FreeGLUT i GLFW zawierają procedurę `main`, a więc takie okno Windows otworzy także dla nich. W chwili zatrzymania aplikacji (np. z powodu błędu) okno to zostaje natychmiast zamknięte, przez co nie ma czasu na przeczytanie jego treści. Dlatego uruchamiając aplikację, trzeba (zarówno w Windows, jak i w Linuksie) wywoływać ją z terminala tekstowego otwartego wcześniej, wtedy teksty pojawią się i pozostaną widoczne w nim.

Listing 3.10. Procedura `WinMain`

C

```
.... /* tu te same dyrektywy, może poza #include <stdio.h>, i procedury */

int WINAPI WinMain ( HINSTANCE inst, HINSTANCE prev,
                     LPSTR cmd_line, int show )
{
  Initialise ( inst, show, 1, &cmd_line );
  MessageLoop ();
  Cleanup ();
  return 0;
} /*WinMain*/
```

Zamiast liczby parametrów wywołania i tablicy `argv` procedura `WinMain` otrzymuje od systemu jeden napis z wszystkimi parametrami; interpretacja parametrów wywołania jest pozostawiona procedurze `InitMyWorld` aplikacji.

Procedura pokazana na listingu 3.11 kolejno otwiera możliwość skonstruowania kontekstu OpenGL-a, tworzy okno aplikacji, tworzy kontekst i powoduje ukazanie się okna na ekranie, po czym przygotowuje szadery oraz dane, na podstawie których aplikacja będzie tworzyć obrazy.

Listing 3.11. Procedura inicjalizacji aplikacji Windows

C

```
HWND  window;
HDC   gldc;
HGLRC glcontext;

void Initialise ( HINSTANCE inst, int show, int argc, char *argv[] )
{
  InitWGLExtensions ( 4, 2 );                          /* listing 3.16 */
  window = CreateMyWindow ( inst, 480, 360 );          /* listing 3.12 */
  gldc = GetDC ( window );
  glcontext = InitWGLContext ( gldc, 4, 2, 0, NULL );  /* listing 3.16 */
  ShowWindow ( window, show );
  InitMyWorld ( argc, argv, 480, 360 );
  UpdateWindow ( window );
} /*Initialise*/
```

W procedurze z listingu 3.12, która tworzy okno, nie ma żadnych instrukcji związanych z OpenGL-em; kontekst OpenGL-a będzie skonstruowany osobno i przywiązany do okna później. Tworząc okno, trzeba określić procedurę, która ma przetwarzać komunikaty skierowane do niego. Może nią być procedura pokazana na listingu 3.13.

O ile w systemie X Window (nie licząc rozszerzeń) wystarczy 35 rodzajów komunikatów, o tyle liczba różnych komunikatów zdefiniowanych w systemie Windows jest bliska tysiąca. Aplikacja sama reaguje tylko na komunikaty wybrane przez autora. Pozostałe komunikaty

Listing 3.12. Tworzenie okna w Windows

C

```
1: HWND CreateMyWindow ( HINSTANCE inst, int width, int height )
2: {
3:   WNDCLASSA wclass;
4:   RECT      rect;
5:   DWORD     wstyle;
6:   HWND      window;
7:
8:   memset ( &wclass, 0, sizeof(WNDCLASSA) );
9:   wclass.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
10:  wclass.lpfnWndProc = window_callback;  /* listing 3.13 */
11:  wclass.hInstance = inst;
12:  wclass.hCursor = LoadCursor ( 0, IDC_ARROW );
13:  wclass.hbrBackground = 0;
14:  wclass.lpszClassName = "WGL_window";
15:  if ( !RegisterClassA ( &wclass ) )
16:    ExitOnError ( "Failed to register window class." );
17:  memset ( &rect, 0, sizeof(RECT) );
18:  rect.right = width;  rect.bottom = height;
19:  wstyle = WS_OVERLAPPEDWINDOW;
20:  AdjustWindowRect ( &rect, wstyle, 0 );
21:  window = CreateWindowExA ( 0, wclass.lpszClassName, "OpenGL", wstyle,
22:                  CW_USEDEFAULT, CW_USEDEFAULT, rect.right - rect.left,
23:                  rect.bottom - rect.top, 0, 0, inst, 0 );
24:  if ( !window )
25:    ExitOnError ( "Failed to create window." );
26:  return window;
27: } /*CreateMyWindow*/
```

powinna przekazać dalej, aby system przetworzył je w sposób domyślny. Do tego służy procedura `DefWindowProcA` wywoływana w linii 46.

Komunikaty `WM_CLOSE` i `WM_DESTROY` są wysyłane po zamknięciu okna przez użytkownika. Procedura `PostQuitMessage` wysyła komunikat `WM_QUIT`, który spowoduje wyjście z pętli komunikatów.

Komunikat `WM_PAINT` jest wysyłany, jeśli obraz w oknie trzeba odtworzyć (np. po otwarciu okna lub odsłonięciu jego części wcześniej zasłanianej przez inne okno). Podobnie jak w X Window, po odsłonięciu części okna może być wysłana cała seria takich komunikatów. W przedstawionym tu rozwiązaniu komunikat powoduje nadanie zmiennej `redraw` wartości niezerowej. Procedura wykonująca obraz zostanie wywołana przez opisaną dalej (listing 3.14) procedurę pętli komunikatów.

Procedury w liniach 3–9 ma napisać autor aplikacji; są one wywoływane odpowiednio po zmianie wymiarów okna, naciśnięciu lub zwolnieniu przycisku myszy, obróceniu rolki, przesunięciu myszy oraz naciśnięciu (dowolnego) klawisza, który jeśli reprezentuje literę, cyfrę lub inny znak mający kod ASCII, to jest dodatkowo wywoływana procedura z linii 9. Informacje na temat położenia kursora lub kod znaku są brane z parametrów `wParam` i `lParam`.

Listing 3.13. Procedura obsługi komunikatów okna

C

```
char redraw;

void MyWinResizeFunc ( HWND window, int width, int height ) { .... }
void MyWinMouseFunc ( HWND window, UINT msg, int x, int y ) { .... }
void MyWinScrollFunc ( HWND window, int ticks ) { .... }
void MyWinMotionFunc ( HWND window, int x, int y ) { .... }
void MyWinKeyboardFunc ( HWND window, UINT msg,
                         WPARAM wParam, LPARAM lParam ) { .... }
void MyWinCharFunc ( HWND window, int charcode ) { .... }

LRESULT CALLBACK window_callback ( HWND window, UINT msg,
                                   WPARAM wParam, LPARAM lParam )
{
  LRESULT result = 0;

  switch ( msg ) {
case WM_CLOSE:
case WM_DESTROY:
    PostQuitMessage ( 0 );
    break;
case WM_PAINT:
    result = DefWindowProcA ( window, msg, wParam, lParam );
    redraw = true;
    break;
case WM_SIZE:
    if ( wParam != SIZE_MINIMIZED )
      MyWinResizeFunc ( window, LOWORD(lParam), HIWORD(lParam) );
    break;
case WM_MOUSEMOVE:
    MyWinMotionFunc ( window, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam) );
    break;
case WM_LBUTTONDOWN:  case WM_MBUTTONDOWN:  case WM_RBUTTONDOWN:
case WM_LBUTTONUP:    case WM_MBUTTONUP:    case WM_RBUTTONUP:
    MyWinMouseFunc(window, msg, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
    break;
case WM_MOUSEWHEEL:
    MyWinScrollFunc ( window, HIWORD(wParam) );
    break;
case WM_KEYFIRST:
    MyWinKeyboardFunc ( window, msg, wParam, lParam );
    break;
case WM_CHAR:
    MyWinCharFunc ( window, wParam );
    break;
default:
```

```
46:     result = DefWindowProcA ( window, msg, wParam, lParam );
47:     break;
48:   }
49:   return result;
50: } /*window_callback*/
```

Pętla komunikatów jest realizowana przez procedurę pokazaną na listingu 3.14. Jeśli w kolejce jest komunikat dla aplikacji, to w linii 10 jest on odczytywany, a następnie (w liniach 14 i 15) przekształcany i wysyłany do adresata (tj. do właściwego okna lub boksu dialogowego). Następnie, jeśli zmienna `redraw` ma wartość niezerową, wywoływana jest procedura wykonująca obraz, a po niej procedura `SwapBuffers`, która przesyła go do okna. Jeśli zmienna `idlefunc` ma wartość inną niż `NULL`, to wskazywany przez nią podprogram jest wywoływany (i może przygotować przykładowo następną klatkę animacji), a w przeciwnym razie aplikacja wywołuje procedurę `WaitMessage`, z której powrót nastąpi, gdy w kolejce pojawi się następny komunikat. Dzięki temu, jeśli aplikacja ma reagować tylko na działania użyt-

Listing 3.14. Procedura pętli komunikatów

C

```
1: static void (*idlefunc)(void) = NULL;
2:
3: void MessageLoop ( void )
4: {
5:   char terminate;
6:   MSG  msg;
7:
8:   redraw = true;
9:   for ( terminate = false; !terminate; ) {
10:    if ( PeekMessageA ( &msg, 0, 0, 0, PM_REMOVE ) ) {
11:      if ( msg.message == WM_QUIT )
12:        terminate = true;
13:      else {
14:        TranslateMessage ( &msg );
15:        DispatchMessage ( &msg );
16:      }
17:    }
18:    if ( redraw ) {
19:      RedrawMyWorld ();    /* do napisania przez autora aplikacji */
20:      redraw = false;
21:      SwapBuffers ( gldc );
22:    }
23:    if ( idlefunc )
24:      idlefunc ();
25:    else
26:      WaitMessage ();
27:  }
28: } /*MessageLoop*/
```

kownika, a nie na upływ czasu, zewnętrzna pętla „nie kręci się na pusto", co pozwala CPU ostygnąć lub zająć się czymś innym.

Listing 3.15. Procedura sprzątania

C

```
1: void Cleanup ( void )
2: {
3:   DeleteMyWorld ();  /* do napisania przez autora aplikacji */
4:   wglMakeCurrent ( gldc, 0 );
5:   wglDeleteContext ( glcontext );
6:   DestroyWindow ( window );
7: } /*Cleanup*/
```

Po wyjściu z pętli komunikatów aplikacja powinna posprzątać; procedura pokazana na listingu 3.15 kolejno zwalnia pamięć CPU i GPU zajętą przez dane aplikacji, odaktywnia i likwiduje kontekst OpenGL-a i zamyka okno.

3.4.1. Tworzenie kontekstu OpenGL-a

W aplikacji Windows skonstruowanie kontekstu *nowego* OpenGL-a (tj. dla wersji 3.0 lub nowszej) musi być poprzedzone utworzeniem zgodnego ze starszą specyfikacją kontekstu pomocniczego i pomocniczego okna, aby było do czego przywiązać ten kontekst. Służy on do uzyskania (przez odpowiednie wskaźniki) możliwości wywołania procedur z biblioteki WGL[25], dzięki którym jest możliwa konstrukcja kontekstów, z których aplikacja będzie korzystać.

Listing 3.16 przedstawia dwie procedury, które umieściłem w osobnym pliku źródłowym `initwglctx.c`. Zapisałem w nim też makrodefinicje i deklaracje potrzebne do utworzenia kontekstu[26].

Procedura `InitWGLExtensions` musi być wywołana tylko raz na początku działania aplikacji. Instrukcje w liniach 44–55 tworzą okno pomocnicze. W linii 57 jest otrzymywany uchwyt (identyfikator) kontekstu urządzenia dla okna. W liniach 58–66 powstaje opis formatu pikseli, jaki urządzenie powinno obsługiwać, co jest sprawdzane w liniach 67–71. Kontekst pomocniczy po utworzeniu w linii 72 jest uaktywniany. To umożliwia (procedurze `wglGetProcAddress`) otrzymanie adresów dwóch procedur potrzebnych do skonstruowania kontekstu docelowego, Kontekst pomocniczy umożliwia też uzyskanie dostępu do procedur nowego OpenGL-a, za pomocą instrukcji w linii 81. Potem pomocniczy kontekst i okno będą niepotrzebne, więc ulegają likwidacji.

Kontekst przeznaczony do właściwej pracy konstruuje procedura `InitWGLContext`. Dla każdego okna, w którym ma być używany OpenGL, trzeba ją wywołać, podając jako pierwszy parametr uchwyt kontekstu urządzenia otrzymany po utworzeniu tego okna.

[25] Biblioteka ta jest częścią biblioteki GL przygotowanej dla systemu Windows, analogicznie do biblioteki GLX dla X Window.

[26] Treść linii 11–31 pochodzi z pliku nagłówkowego `wglext.h`, którego tym sposobem można nie używać.

Listing 3.16. Procedura konstrukcji pomocniczego kontekstu OpenGL-a

C

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include "openglheader.h"

#include "utilities.h"
#include "initwglctx.h"

#define WGL_CONTEXT_PROFILE_MASK_ARB     0x9126
#define WGL_CONTEXT_CORE_PROFILE_BIT_ARB 0x00000001
#define WGL_DRAW_TO_WINDOW_ARB           0x2001
#define WGL_ACCELERATION_ARB             0x2003
#define WGL_SUPPORT_OPENGL_ARB           0x2010
#define WGL_DOUBLE_BUFFER_ARB            0x2011
#define WGL_PIXEL_TYPE_ARB               0x2013
#define WGL_COLOR_BITS_ARB               0x2014
#define WGL_DEPTH_BITS_ARB               0x2022
#define WGL_STENCIL_BITS_ARB             0x2023
#define WGL_FULL_ACCELERATION_ARB        0x2027
#define WGL_TYPE_RGBA_ARB                0x202B
#define WGL_CONTEXT_MAJOR_VERSION_ARB    0x2091
#define WGL_CONTEXT_MINOR_VERSION_ARB    0x2092
#define WGL_CONTEXT_FLAGS_ARB            0x2094

typedef HGLRC(WINAPI* PFNWGLCREATECONTEXTATTRIBSARBPROC)
    (HDC hDC, HGLRC hShareContext, const int* attribList);
typedef BOOL(WINAPI* PFNWGLCHOOSEPIXELFORMATARBPROC)
    (HDC hdc, const int* piAttribIList, const FLOAT* pfAttribFList,
     UINT nMaxFormats, int* piFormats, UINT* nNumFormats);

static PFNWGLCREATECONTEXTATTRIBSARBPROC wglCreateContextAttribsARB;
static PFNWGLCHOOSEPIXELFORMATARBPROC    wglChoosePixelFormatARB;

void InitWGLExtensions ( GLint major, GLint minor )
{
  WNDCLASSA             wclass;
  HWND                  dummy_window;
  HDC                   dummy_dc;
  HGLRC                 dummy_context;
  PIXELFORMATDESCRIPTOR pfd;
  int                   pixel_format;

  memset ( &wclass, 0, sizeof(WNDCLASSA) );
```

```
46:   wclass.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
47:   wclass.lpfnWndProc = DefWindowProcA;
48:   wclass.hInstance = GetModuleHandle ( 0 );
49:   wclass.lpszClassName = "Dummy_WGL";
50:   if (!RegisterClassA ( &wclass ) )
51:     ExitOnError ( "Failed to register dummy OpenGL window class." );
52:   dummy_window = CreateWindowExA ( 0, wclass.lpszClassName, "Dummy",
53:         0, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
54:         0, 0, wclass.hInstance, 0 );
55:   if ( !dummy_window )
56:     ExitOnError ( "Failed to create dummy OpenGL window." );
57:   dummy_dc = GetDC ( dummy_window );
58:   memset ( &pfd, 0, sizeof(PIXELFORMATDESCRIPTOR) );
59:   pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
60:   pfd.nVersion = 1;
61:   pfd.iPixelType = PFD_TYPE_RGBA;
62:   pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
63:   pfd.cColorBits = 32;
64:   pfd.cAlphaBits = 8;
65:   pfd.iLayerType = PFD_MAIN_PLANE;
66:   pfd.cDepthBits = 24;
67:   pixel_format = ChoosePixelFormat ( dummy_dc, &pfd );
68:   if ( !pixel_format )
69:     ExitOnError ( "Failed to find a suitable pixel format." );
70:   if ( !SetPixelFormat ( dummy_dc, pixel_format, &pfd ) )
71:     ExitOnError ( "Failed to set the pixel format." );
72:   dummy_context = wglCreateContext ( dummy_dc );
73:   if ( !dummy_context )
74:     ExitOnError ( "Failed to create a dummy OpenGL context." );
75:   if ( !wglMakeCurrent ( dummy_dc, dummy_context ) )
76:     ExitOnError ( "Failed to activate dummy OpenGL context." );
77:   wglCreateContextAttribsARB = (PFNWGLCREATECONTEXTATTRIBSARBPROC)
78:           wglGetProcAddress ( "wglCreateContextAttribsARB" );
79:   wglChoosePixelFormatARB =  (PFNWGLCHOOSEPIXELFORMATARBPROC)
80:           wglGetProcAddress ( "wglChoosePixelFormatARB" );
81:   GetGLProcAddresses ( major, minor );
82:   wglMakeCurrent ( dummy_dc, 0 );
83:   wglDeleteContext ( dummy_context );
84:   ReleaseDC ( dummy_window, dummy_dc );
85:   DestroyWindow ( dummy_window );
86: } /*InitWGLExtensions*/
87:
88: HGLRC InitWGLContext ( HDC real_dc, int major, int minor, int flags,
89:                        int *pixattr )
90: {
91:   int pixel_format_attribs[] = {
92:     WGL_DRAW_TO_WINDOW_ARB, GL_TRUE,
```

```
93:     WGL_SUPPORT_OPENGL_ARB, GL_TRUE,
94:     WGL_DOUBLE_BUFFER_ARB,  GL_TRUE,
95:     WGL_ACCELERATION_ARB,   WGL_FULL_ACCELERATION_ARB,
96:     WGL_PIXEL_TYPE_ARB,     WGL_TYPE_RGBA_ARB,
97:     WGL_COLOR_BITS_ARB,     32,
98:     WGL_DEPTH_BITS_ARB,     24,
99:     WGL_STENCIL_BITS_ARB,   8,
100:    0 };
101:  int ctxattr[] = {
102:    WGL_CONTEXT_MAJOR_VERSION_ARB, 0,
103:    WGL_CONTEXT_MINOR_VERSION_ARB, 0,
104:    WGL_CONTEXT_PROFILE_MASK_ARB,  WGL_CONTEXT_CORE_PROFILE_BIT_ARB,
105:    WGL_CONTEXT_FLAGS_ARB,         0,
106:    0 };
107:  int                   pixel_format;
108:  UINT                  num_formats;
109:  PIXELFORMATDESCRIPTOR pfd;
110:  HGLRC                 gl_context;
111:
112:  if ( !pixattr )
113:    pixattr = pixel_format_attribs;
114:  wglChoosePixelFormatARB ( real_dc, pixattr, NULL,
115:                            1, &pixel_format, &num_formats );
116:  if ( !num_formats )
117:    ExitOnError ( "Failed to find the OpenGL pixel format." );
118:  DescribePixelFormat ( real_dc, pixel_format, sizeof(pfd), &pfd );
119:  if ( !SetPixelFormat ( real_dc, pixel_format, &pfd ) )
120:    ExitOnError ( "Failed to set the OpenGL pixel format." );
121:  ctxattr[1] = major;  ctxattr[3] = minor;  ctxattr[7] = flags;
122:  gl_context = wglCreateContextAttribsARB ( real_dc, 0, ctxattr );
123:  if ( !gl_context )
124:    ExitOnError ( "Failed to create OpenGL context." );
125:  if ( !wglMakeCurrent ( real_dc, gl_context ) )
126:    ExitOnError ( "Failed to activate OpenGL context." );
127:  return gl_context;
128: } /*InitWGLContext*/
```

3.5. Uzupełnienia

3.5.1. Czasomierz

Aplikacje wyświetlające w czasie rzeczywistym „ruchome obrazki", czyli animacje, potrzebują mierzyć upływ czasu, aby obliczać fazy ruchu animowanych obiektów. W systemach Linux i Windows są dostępne procedury odczytu zegara — inne w każdym z nich. Biblioteka GLFW udostępnia własny API, ukrywający wywołania tych procedur. Ale na potrzeby

wszystkich aplikacji opisanych w tej książce napisałem procedury realizujące API zegara dostosowane do obu systemów i możliwe do użycia w aplikacjach bibliotek FreeGLUT i GLFW.

Przedstawiony tu API zegara składa się z procedur `TimerInit`, `TimerTic`, `TimerToc` i `TimerTocTic` oraz globalnych zmiennych `app_time`, `tic_time` i `toc_time`. Wspomniane procedury nadają im wartości, które aplikacja może odczytywać. Procedura `TimerInit` powinna zostać wywołana na początku działania programu (np. przez procedurę `InitMyWorld`). Procedura `TimerTic`, wywołana w dowolnej chwili, uruchamia stoper. Procedura `TimerToc` podaje wskazanie stopera, tj. czas (w sekundach), który upłynął od ostatniego wywołania procedury `TimerTic` (albo `TimerInit`). Procedura `TimerTocTic` podaje wskazanie stopera, jednocześnie uruchamiając go ponownie.

W zmiennej `app_time` każda z procedur zapisuje czas, który upłynął od chwili wywołania `TimerInit`, przy czym procedury `TimerInit`, `TimerTic` i `TimerTocTic` nadają tę samą wartość także zmiennej `tic_time` (i przypisują 0 zmiennej `toc_time`). Czas, który upłynął od ostatniego nadania wartości zmiennej `tic_time` (czyli od uruchomienia stopera) do chwili odczytu wskazania stopera jest przez procedury `TimerToc` i `TimerTocTic` przypisywany zmiennej `toc_time` i przekazywany w instrukcji `return`.

Zwróćmy uwagę na lokalną odmianę problemu milenijnego. Czas jest wielkością analogową, zatem wyniki jego pomiarów przyjmują wartości ułamkowe i do obliczania fazy ruchu animowanych obiektów trzeba je podawać w reprezentacji zmiennopozycyjnej. W miarę upływu czasu od początku działania aplikacji rośnie bezwzględny błąd tej reprezentacji, a więc wyniki pomiarów czasu, który upłynął od uruchomienia aplikacji, stają się coraz mniej dokładne. Po dostatecznie długim czasie może to spowodować zmniejszenie płynności animacji lub inne kłopoty. W reprezentacji zmiennopozycyjnej liczby T można liczyć na dokładność co najwyżej ulp T, tj. wartość najmniej znaczącego bitu mantysy (zobacz s. 1187). Można chyba przyjąć, że dopuszczalny błąd pomiarów czasu na potrzeby animacji nie powinien przekraczać jednej setnej sekundy[27]. Dla zmiennych pojedynczej precyzji (typu `float`) przekroczenie tego poziomu błędu nastąpiłoby po upływie czasu $T > 0.01 \cdot 2^{24}\text{s} \approx 46\,\text{h}$. Dlatego do reprezentowania wyników pomiarów czasu trzeba używać zmiennych podwójnej precyzji (typu `double`). Umożliwi to poprawne działanie aplikacji — gry komputerowej — nawet przez $0.01 \cdot 2^{53}$ s; większości graczy tyle czasu powinno wystarczyć[28].

Listingi 3.17 i 3.18 przedstawiają opisane wyżej procedury dostosowane do systemów Linux i Windows. Procedury odczytu zegara w obu tych systemach przekazują wynik w postaci liczb stałopozycyjnych — w sekundach i nanosekundach (w Linuksie) lub w jednostkach podanych przez procedurę `QueryPerformanceFrequency` (w Windows). Obie implementacje przechowują wyniki odczytu zegara w statycznych (ukrytych przed aplikacją) zmiennych stałopozycyjnych, na których działania dodawania i odejmowania są wykonywane bez błędów zaokrągleń. Błędy te pojawiają się w przejściu do reprezentacji zmiennopozycyjnej liczb opisujących czas.

[27] Aplikacje czasu rzeczywistego z rozmaitych powodów mogą wymagać większej dokładności.

[28] Tego rodzaju problemy stanowią nie lada wyzwanie podczas testowania programów — kto normalny jest w stanie przewidzieć obecność błędu, który nie może być zauważony wcześniej niż po kilkudziesięciu godzinach nieprzerwanego działania?

Listing 3.17. Implementacja API zegara dla systemu Linux

C

```
#ifdef __linux__  /* Linux */
#include <sys/time.h>
#define __USE_POSIX199309
#include <time.h>
#include "openglheader.h"
#include "utilities.h"

double app_time, tic_time, toc_time;

static clockid_t       clockid;
static struct timespec res, tick0, tick1, tick2;

#define TDIFF(t1,t0)
  ((double)(t1.tv_sec-t0.tv_sec) + 1.0e-9*(double)(t1.tv_nsec-t0.tv_nsec))

void TimerInit ( void )
{
  if ( clock_getres ( (clockid = CLOCK_MONOTONIC), &res ) )
    if ( clock_getres ( (clockid = CLOCK_REALTIME), &res ) )
      ExitOnError ( "TimerInit" );
  clock_gettime ( clockid, &tick0 );
  tick1 = tick2 = tick0;
  app_time = tic_time = toc_time = 0.0;
} /*TimerInit*/

double TimerTic ( void )
{
  clock_gettime ( clockid, &tick1 );
  tick2 = tick1;
  toc_time = 0.0;
  return app_time = tic_time = TDIFF ( tick2, tick0 );
} /*TimerTic*/

double TimerToc ( void )
{
  clock_gettime ( clockid, &tick2 );
  app_time = TDIFF ( tick2, tick0 );
  return toc_time = TDIFF ( tick2, tick1 );
} /*TimerToc*/

double TimerTocTic ( void )
{
  struct timespec tick;

  tick = tick1;
```

```
46:   clock_gettime ( clockid, &tick1 );
47:   tick2 = tick1;
48:   app_time = TDIFF ( tick2, tick0 );
49:   toc_time = 0.0;
50:   return toc_time = TDIFF ( tick2, tick );
51: } /*TimerTocTic*/
52: #endif
```

Listing 3.18. Implementacja API zegara dla systemu Windows

C

```
1: #ifdef _WIN32  /* Windows */
2: #define WIN32_LEAN_AND_MEAN
3: #include <Windows.h>
4: #include <profileapi.h>
5: #include "openglheader.h"
6: #include "utilities.h"
7:
8: double app_time, tic_time, toc_time;
9:
10: static double        frequency;
11: static LARGE_INTEGER tick0, tick1, tick2;
12:
13: #define TDIFF(t1,t0) ((double)(t1.QuadPart-t0.QuadPart)/frequency)
14:
15: void TimerInit ( void )
16: {
17:   LARGE_INTEGER fv;
18:
19:   if ( QueryPerformanceFrequency ( &fv ) != 0 ) {
20:     frequency = (double)fv.QuadPart;
21:     QueryPerformanceCounter ( &tick2 );
22:     tick0 = tick1 = tick2;
23:   }
24:   else
25:     ExitOnError ( "TimerInit" );
26:   app_time = toc_time = 0.0;
27: } /*TimerInit*/
28:
29: double TimerTic ( void )
30: {
31:   QueryPerformanceCounter ( &tick2 );
32:   tick1 = tick2;
33:   toc_time = 0.0;
34:   return app_time = tic_time = TDIFF ( tick2, tick0 );
35: } /*TimerTic*/
36:
37: double TimerToc ( void )
```

```
38: {
39:   QueryPerformanceCounter ( &tick2 );
40:   app_time = TDIFF ( tick2, tick0 );
41:   return toc_time = TDIFF ( tick2, tick1 );
42: } /*TimerToc*/
43:
44: double TimerTocTic ( void )
45: {
46:   LARGE_INTEGER tick;
47:
48:   tick = tick1;
49:   QueryPerformanceCounter ( &tick2 );
50:   tick1 = tick2;
51:   app_time = TDIFF ( tick2, tick0 );
52:   toc_time = 0.0;
53:   return toc_time = TDIFF ( tick2, tick );
54: } /*TimerTocTic*/
55: #endif
```

3.5.2. Technologia Optimus

Mnie wiele można zarzucić, no nie że ja nowoczesny.

Kaźmirz Pawlak, *Nie ma mocnych*

Poważnym problemem ludzkości jest nadmierne zużycie energii i dlatego ludzkość wynajduje rozmaite sposoby jej oszczędzania. Innym poważnym problemem jest konkurencja między producentami sprzętu, którzy nie bardzo ułatwiają sobie nawzajem działalność. Jednym ze skutków tego stanu rzeczy jest stworzona przez firmę NVIDIA technologia Optimus, która w nowoczesnych laptopach realizuje współpracę GPU tej firmy z podukładem graficznym wbudowanym w CPU firmy Intel. W tej technologii GPU jest czynna (i zużywa energię) tylko w czasie pracy aplikacji używających GPU. Obrazy tworzone przez GPU są przesyłane do okien systemu X Window, którego działanie organizuje i obsługuje CPU, posługując się swoim podukładem graficznym. Większość wersji tego podukładu nie realizuje standardu OpenGL w wersji 4.0 i dalszych, bo nie po to powstały.

W komunikacji między CPU a GPU w laptopie z Optimusem następują opóźnienia, przy czym nie chodzi tu o czas, tylko o pełną sekwencję komunikatów, które muszą być przesłane w obie strony, aby wszystkie dane dotarły na właściwe miejsca. Obraz widoczny w oknie podczas wyświetlania animacji (np. przez grę) jest przez to opóźniony, zazwyczaj o jedną klatkę[29]. Również informacja o tym, że użytkownik zmienił wymiary okna, dociera do GPU z opóźnieniem. W rezultacie obraz wyświetlony (po wykonaniu procedury `DisplayFunc` aplikacji FreeGLUT-a, procedury `RedrawMyWorld` aplikacji GLFW lub procedury obsługi komunikatu `Expose` aplikacji X Window) bezpośrednio po zmianie tych wymiarów składa

[29] Czyli typowo o 1/60 s, to zależy od sygnałów sterujących monitorem.

się z fragmentów obrazu dostosowanego do okna o poprzednich wymiarach i z obszarów o zawartości nieokreślonej.

Jeśli aplikacja wyświetla obrazy wiele razy na sekundę, to nieudany obraz w oknie jest od razu zastępowany obrazem takim, jak trzeba. Jeśli jednak zawartość okna jest zmieniana *tylko* w odpowiedzi na działania użytkownika, to efekt zmiany wymiarów okna wygląda nieciekawie. Aby temu przeciwdziałać, trzeba po zmianie wielkości okna wymusić kilkakrotne wywołanie procedury rysowania. Nie oznacza to konieczności poniesienia kosztu wielu rysowań, bo wystarczy wykonać właściwy obraz tylko podczas ostatniego wywołania[30].

Można to zrealizować tak: deklarujemy globalną zmienną typu całkowitego, nazwijmy ją `opti`. W aplikacjach bibliotek FreeGLUT i GLFW zrealizowanych przy użyciu opisanych w tym rozdziale szkieletów do procedury `ReshapeFunc`, a w aplikacjach X Window do procedury `MyWinConfigureNotify` należy dodać instrukcję przypisania `opti = 4;`. Listingi 3.19, 3.20 i 3.21 przedstawiają zmienione procedury, których zadaniem jest wywołanie procedury `RedrawMyWorld` we właściwym momencie.

Listing 3.19. Procedura `DisplayFunc` dla aplikacji biblioteki FreeGLUT

C

```
void DisplayFunc ( void )
{
  if ( opti > 0 ) {
    opti --;
    glutPostWindowRedisplay ( WindowHandle );
  }
  else
    RedrawMyWorld ();
  glutSwapBuffers ();
} /*DisplayFunc*/
```

Listing 3.20. Procedura `Redraw` dla aplikacji biblioteki GLFW

C

```
void Redraw ( void )
{
  if ( opti > 0 ) {
    opti --;
    glfwPostEmptyEvent ();
  }
  else {
    RedrawMyWorld ();
    redraw = false;
  }
  glfwSwapBuffers ();
} /*Redraw*/
```

[30] Przedstawione tu rozwiązanie wykoncypowałem na podstawie eksperymentów z zaledwie jednym laptopem i nie mogę zagwarantować, że zawsze będzie skuteczne.

W aplikacji biblioteki GLFW wywołanie procedury `glfwPostEmptyEvent` ma ten cel, aby aplikacja nie czekała zawieszona w procedurze `glfwWaitEvents`, gdy obraz w oknie jest nieodświeżony.

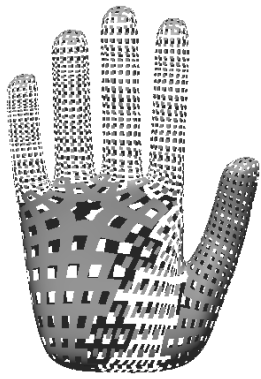
Listing 3.21. Procedura `MyWinExpose` dla aplikacji X Window

C

```
1: void MyWinExpose ( void )
2: {
3:   if ( opti > 0 ) {
4:     opti --;
5:     PostExposeEvent ( xmywin, win_width, win_height );
6:   }
7:   else
8:     RedrawMyWorld ();
9:   glXSwapBuffers ( xdisplay, xmywin );
10: } /*MyWinExpose*/
```

Procedury `glutSwapBuffers`, `glfwSwapBuffers` i `glXSwapBuffers` wykonują ostatni krok podwójnego buforowania, tj. pokazują w oknie zawartość niewidocznego dotąd bufora, w którym powstał obraz. To właśnie jedną z tych procedur (i nic oprócz niej) wystarczy wywołać podczas obsługi dodatkowych poleceń wykonania obrazu.

Z uwagi na znikomy koszt chyba warto (w opisany wyżej sposób) przygotować aplikację do poprawnego działania na laptopach, nawet jeśli pisząc ją na komputerze stacjonarnym, nie przewidujemy, że ktoś będzie chciał jej używać na laptopie.



Rysunek 3.1. Niedokończony obraz wyświetlony podczas animacji na laptopie

Dysponując umiarkowaną (choć wcale niemałą) mocą obliczeniową, GPU w laptopie może nie nadążać z wykonywaniem obrazów bardzo skomplikowanych scen podczas animacji w czasie rzeczywistym, czego skutkiem jest wyświetlanie niedokończonych obrazów. Na rysunku 3.1 jest przedstawiony przykład obrazu wyświetlonego przez aplikację opisaną w rozdziale 35. Obiekt składa się z 82432 trójkątów. GPU w moim laptopie nie zdążyła dokoń-

czyć rysowania, zanim CPU pobrała obraz i wysłała polecenie wykonywania następnego[31], choć nadążała z rysowaniem w mniejszym oknie. Morał stąd taki, że poziom szczegółowości wyświetlanych obiektów trzeba dostosowywać do możliwości sprzętu.

Biorąc pod uwagę kłopoty z tą technologią (w tym także z bardzo nietrywialną instalacją sterownika GPU i związaną z tym konfiguracją systemu X Window), zamieszczonego na stronie firmy NVIDIA tekstu nie należy, jak każdej zresztą reklamy, brać zbyt dosłownie. Cytuję:

> *NVIDIA's Optimus technology works right out of the box. There's absolutely no setup or configuration required—it's that simple. Optimus technology will automatically optimize your notebook PC, providing you the outstanding graphics performance you need, when you need it …*

Ale myli się ten, kto sądzi, że źle oceniam technologię Optimus; firma zrobiła, co było możliwe. Posiadaczom komputerów z Optimusem szczerze życzę dużo cierpliwości i powodzenia.

3.5.3. Wymiary ekranu

Środowiska FreeGLUT, GLFW, X11 i Windows umożliwiają otrzymanie przez aplikację informacji o wymiarach ekranu — szerokości i wysokości w pikselach i w milimetrach — co teoretycznie umożliwia obliczenie współczynnika aspektu (rozdz. 6) w celu uniknięcia zniekształcenia obiektów spowodowanego przez inne skalowanie wymiarów pionowych i poziomych na obrazie. Aplikacja biblioteki FreeGLUT może w tym celu wywołać procedurę `glutGet` kolejno z parametrami `GLUT_SCREEN_WIDTH`, `GLUT_SCREEN_HEIGHT`, `GLUT_SCREEN_WIDTH_MM` i `GLUT_SCREEN_HEIGHT_MM`. Wartość powrotna (typu `int`) to odpowiednio szerokość i wysokość ekranu w pikselach oraz szerokość i wysokość ekranu w milimetrach.

Aplikacja biblioteki GLFW uzyska wymiary ekranu w pikselach za pomocą procedury `glfwGetVideoMode`; jej parametrem jest zmienna typu zamkniętego[32] `GLFWmonitor`, a wartość powrotna jest wskaźnikiem struktury typu `GLFWvidmode`, której pola zawierają m.in. szerokość i wysokość ekranu w pikselach. Fizyczne wymiary ekranu (w milimetrach) podaje procedura `glfwGetMonitorPhysicalSize`.

W systemie X Window mamy do dyspozycji procedury (a raczej makra) `DisplayWidth`, `DisplayHeight`, `DisplayWidthMM` i `DisplayHeightMM`, które podają wymiary ekranu w pikselach i w milimetrach.

Aplikacje systemu Windows otrzymają szerokość i wysokość ekranu w pikselach, wywołując procedurę `GetSystemMetrics` kolejno z parametrami `SM_CXSCREEN` i `SM_CYSCREEN`. Wymiary ekranu w milimetrach aplikacja może poznać za pomocą procedury pokazanej na listingu 3.22.

[31] Co zastanawiające, nie pomogło wywoływanie procedury `glFinish` zamiast `glFlush`, zobacz s. 156.

[32] Wartości typów zamkniętych są liczbami całkowitymi, które służą do identyfikowania rozmaitych obiektów i nie wykonuje się na nich żadnych działań arytmetycznych. Dobrymi przykładami typów zamkniętych są używane w systemie X Window typy `Atom` i `Window`.

Listing 3.22. Procedura odczytywania wymiarów ekranu w Windows

C

```
1: void GetScreenDimensions ( int *widthMM, int *heightMM )
2: {
3:   HWND desktop;
4:   HDC  context;
5:
6:   desktop = GetDesktopWindow ();
7:   context = GetDC ( desktop );
8:   *widthMM = GetDeviceCaps ( context, HORZSIZE );
9:   *heightMM = GetDeviceCaps ( context, VERTSIZE );
10:  ReleaseDC ( desktop, context );
11: } /*GetScreenDimensions*/
```

Trzeba pamiętać, że informacja o wymiarach fizycznych ekranu nie zawsze jest dostępna i nie zawsze jest rzetelna. Na przykład z informacji podanych przez system okien dowiedziałem się, że (czternastocalowy) ekran mojego laptopa ma dwadzieścia cali szerokości[33]. Ponadto system okien można[34] tak skonfigurować, aby tzw. **wirtualny ekran** miał większe wymiary (w pikselach) i wtedy tylko jego część będzie w każdej chwili widoczna. W takim przypadku makra `DisplayWidth` i `DisplayHeight` podają wymiary w pikselach ekranu wirtualnego, a makra `DisplayWidthMM` i `DisplayHeightMM` podają wymiary ekranu fizycznego w milimetrach, co nie daje możliwości poprawnego obliczenia rozdzielczości rastra w pikselach na milimetr (albo cal)[35], ani współczynnika aspektu. Wreszcie, jeśli jest używany rzutnik, to obraz wyświetlany na ścianie jest znacznie większy niż ekran monitora, a komputer nie ma żadnej możliwości zmierzenia go. Dlatego najbezpieczniej jest wymiary ekranu, jeśli są konieczne potrzebne, zapisać w pliku konfiguracyjnym, który aplikacja przeczyta na początku swojego działania. Fizyczne wymiary obrazu (ekranu monitora lub ekranu umieszczonego naprzeciw rzutnika) mogą zostać zapisane w takim pliku podczas instalowania aplikacji lub bezpośrednio przed jej uruchomieniem.

Do czytania pliku konfiguracyjnego, który zapewne powinien być tekstowy, można użyć skanera opisanego w podrozdziale 14.1. Jednak opracowanie składni takiego pliku i odpowiedniego analizatora składni (parsera) zbyt daleko odbiega od głównych tematów, czyli grafiki, OpenGL-a i GLSL-a, aby się tym zajmować w tej książce. Czytelnicy zechcą mi wybaczyć.

[33] To by oznaczało, na mocy twierdzenia Pitagorasa, że do opisania wymiarów tego ekranu konieczne są liczby zespolone: wysokość $\sqrt{(14")^2 - (20")^2}$ to w przybliżeniu 14 cali urojonych.

[34] nie zawsze — to zależy od sterownika „karty graficznej"

[35] Odpowiednie procedury w bibliotekach FreeGLUT i GLFW, skompilowane dla systemu X Window, korzystają z tych makr, a zatem nie mogą wyczarować bardziej rzetelnej informacji.

4

Utensylia

Aplikacje OpenGL-a potrzebują procedur pomocniczych, z których wiele jest na początek opisanych w tym rozdziale. Procedury te umieściłem w osobnym pliku źródłowym (o nazwie `utilities.c`), a ich prototypy w odpowiednim pliku nagłówkowym (`utilities.h`). Będą one użyte we wszystkich opisanych dalej aplikacjach.

4.1. Wypisanie informacji o wersji

Procedura `PrintGLVersion` wypisuje napis zawierający informację o wersji standardu OpenGL i wersji języka GLSL obsługiwanych przez środowisko, w którym została wywołana aplikacja (tj. przez zainstalowane w tym środowisku biblioteki) oraz o producencie implementacji standardu. Warto tę procedurę wywołać na początku działania aplikacji (ale *po* uzyskaniu dostępu do procedury `glGetString`, np. przez wywołanie procedury `gl3wInit` albo `gladLoadGL`), aby dowiedzieć się, czy aplikacja nie wymaga za dużo.

Listing 4.1. Wypisanie informacji na temat wersji OpenGL-a

C

```
1: void PrintGLVersion ( void )
2: {
3:   printf ( "OpenGL %s, GLSL %s\n",
4:             glGetString ( GL_VERSION ),
5:             glGetString ( GL_SHADING_LANGUAGE_VERSION ) );
6: } /*PrintGLVersion*/
```

4.2. Reakcje na błąd

Procedury `_ExitOnError` i `_ExitIfGLError` pokazane na listingu 4.2 są niesłychanie przydatne podczas uruchamiania programu. Wywołanie pierwszej z nich powoduje wypisanie

Listing 4.2. Procedury `ExitOnError` i `ExitIfGLError`

C

```
#define ExitIfGLError(msg) _ExitIfGLError ( __FILE__, __LINE__, msg )
#define ExitOnError(msg) _ExitOnError ( __FILE__, __LINE__, msg )

void _ExitOnError ( const char *file, int line, const char *msg )
{
  fprintf ( stderr, "Error: %s (%d): %s\n",
            file, line, (msg ? msg : "") );
  exit ( 1 );
} /*_ExitOnError*/

static GLchar *errstr[] =
  { "Invalid enum",
    "Invalid value",
    "Invalid operation",
    "Stack overflow",
    "Stack underflow",
    "Out of memory",
    "Invalid framebuffer operation",
    "" };

GLchar *GetGLErrorString ( GLenum err )
{
  switch ( err ) {
case GL_INVALID_ENUM:                  return errstr[0];
case GL_INVALID_VALUE:                 return errstr[1];
case GL_INVALID_OPERATION:             return errstr[2];
case GL_STACK_OVERFLOW:                return errstr[3];
case GL_STACK_UNDERFLOW:               return errstr[4];
case GL_OUT_OF_MEMORY:                 return errstr[5];
case GL_INVALID_FRAMEBUFFER_OPERATION: return errstr[6];
default:
case GL_NO_ERROR:                      return errstr[7];
  }
} /*GetGLErrorString*/

void _ExitIfGLError ( const char *file, int line, const char *msg )
{
  GLenum err;

  if ( (err = glGetError ()) != GL_NO_ERROR ) {
    fprintf ( stderr, "Error %4d: %s,\n    %s (%d): %s\n",
              err, GetGLErrorString ( err ), file, line, (msg ? msg : "") );
    exit ( 1 );
  }
} /*_ExitIfGLError*/
```

miejsca wystąpienia błędu i tekstu przekazanego jako parametr i zatrzymanie programu, co jest zwięzłym zapisem reakcji programu na błąd uniemożliwiający dalsze działanie.

Wywołanie drugiej z tych procedur powinno być umieszczone po instrukcjach wywołujących procedury OpenGL-a; *jeśli* nastąpił błąd, to procedura ta zatrzyma program, ale przedtem wyświetli komunikat diagnostyczny, który może pomóc w zrozumieniu natury błędu i w jego naprawieniu. Warto ją wywoływać dosyć często, bo to ułatwia odkrycie miejsca i przyczyny błędu. Jeśli OpenGL nie odnotował błędu, to następuje powrót z procedury i aplikacja działa dalej jak gdyby nigdy nic.

Procedura `_ExitIfGLError` wywołuje procedurę `glGetError`, której wartość wskazuje, czy nastąpił błąd i jeśli tak, to jaki, a procedura `GetGLErrorString`[1] tłumaczy liczbowy kod błędu na język angielski.

OpenGL zgłasza błędy siedmiu rodzajów: błędny parametr wyliczeniowy (np. przy próbie włączenia nieistniejącej opcji), błędna liczba (np. ujemna długość tablicy lub identyfikator nieistniejącego bufora), błędna operacja, przepełnienie i niedomiar stosu, wyczerpanie dostępnej pamięci oraz niepoprawne użycie bufora ramki. Podczas działania aplikacji może wystąpić kolejno kilka błędów, tego samego lub różnych rodzajów. Zaznaczenie wystąpienia błędu w kontekście OpenGL-a blokuje zapamiętywanie informacji o błędach kolejnych, wskutek czego procedura `glGetError` podaje informację o rodzaju błędu, który wydarzył się najwcześniej[2].

Pierwsze dwa parametry procedur `_ExitOnError` i `_ExitIfGLError` to nazwa pliku źródłowego i numer linii z wywołaniem procedury. Dla ułatwienia procedury te powinny być wywoływane za pośrednictwem makrodefinicji (dalej w książce nazywanych procedurami) `ExitOnError` i `ExitIfGLError`, które mają tylko jeden parametr — dowolny tekst. Kompilując wywołanie procedury, kompilator wygeneruje pierwsze dwa parametry o odpowiednich wartościach w danym miejscu.

4.3. Reprezentacje kodów źródłowych szaderów

Nie zwracając na razie uwagi na składnię języka GLSL, zobaczmy kod źródłowy przykładowego szadera na listingu 4.3. Szader ten można umieścić w pliku na dysku; aby go użyć, aplikacja musi ten plik przeczytać, skompilować i połączyć z innymi szaderami w program szaderów[3].

[1] W bibliotece GLU jest procedura `gluErrorString`, która dokonuje tego tłumaczenia. Ale to jest jedyna procedura z tej biblioteki niezwiązana ze starym OpenGL-em, jej dołączanie do naszych aplikacji to przesada.

[2] Po wykryciu błędu aplikacja może próbować opanować sytuację. Ubocznym efektem procedury `glGetError` jest skasowanie informacji o błędzie w kontekście OpenGL-a, co przywraca mu zdolność do wykrywania kolejnych błędów. Ma to znaczenie, jeśli aplikacja po otrząśnięciu się ze skutków błędu ma działać dalej.

[3] Nazwy i rozszerzenia nazw plików źródłowych szaderów są w zasadzie obojętne, bo to aplikacja jest odpowiedzialna za przeczytanie tych plików i przekazanie ich treści do kompilacji, choć nabiera to znaczenia, jeśli chcemy kompilować szadery do formatu SPIR-V, o czym będzie dalej. Opisany w p. 4.5.1 kompilator rozpoznaje typ szadera po rozszerzeniu, które ma mieć postać `.vert`, `.tesc`, `.tese`, `.geom`, `.frag` albo `.comp`, przy czym dopuszczalne są rozszerzenia podwójne, na przykład `.vert.glsl`. Można tak skonfigurować edytor, aby dla plików o powyższych rozszerzeniach wyróżniał (odpowiednimi kolorami) literały, komentarze i słowa kluczowe języka GLSL, co ułatwia programowanie szaderów.

Listing 4.3. Przykładowy szader wierzchołków

GLSL

```
#version 430

uniform MyGC {
    mat4 pm;      /* macierz rzutowania */
    vec4 fg, bk;  /* kolory obiektu i tła */
  } gc;

layout(location=0) in vec4 in_Position;

void main ( void )
{
  gl_Position = gc.pm * in_Position;
} /*main*/
```

Są dwa drobne kłopoty. Po pierwsze, szadery mogą mieć spore wspólne części kodu źródłowego (takie jak widoczny w przykładzie opis bloku zmiennych jednolitych `MyGC`). Chciałoby się nie powielać takich fragmentów kodu.

W języku C do tego celu służy dyrektywa `#include`; niestety, nie jest ona dostępna w GLSL-u[4]. Wielokrotne pisanie czegokolwiek jest procesem podatnym na błędy. Dlatego możemy podzielić kod w GLSL-u na krótsze fragmenty i składać źródła szaderów z takich fragmentów. Ale wtedy będziemy mieli na dysku wiele króciutkich plików, których obecność obok pliku skompilowanej aplikacji w C musimy zapewnić — i to jest drugi kłopot, istotny zwłaszcza wtedy, gdy chcemy program rozpowszechniać.

Kompilator języka GLSL będący częścią implementacji OpenGL-a umożliwia podanie źródeł szadera w postaci ciągu napisów ASCIIZ. Należy przekazać tablicę wskaźników do tych napisów; można je przeczytać z plików, co jest wygodne podczas tworzenia, uruchamiania i testowania szaderów. Natomiast po ich uruchomieniu lepiej jest je w aplikację wbudować, to znaczy umieścić odpowiednie napisy w kodzie źródłowym programu w C. Ten sam szader co na listingu 4.3 wbudowany w źródła aplikacji może wyglądać tak jak na listingu 4.4. Zauważmy, że kod źródłowy szadera pozostał perfekcyjnie powcinany i czytelny, a przy tym spacje wcięć i komentarze nie przejdą do kodu skompilowanej aplikacji[5], a zatem nie powiększą objętości jej pliku binarnego i (co dla autora programu może mieć znaczenie) nie ułatwią rozszyfrowania kodu szadera przez osoby wścibskie. Poszczególne napisy można umieścić w tablicach „linii" kodu innych szaderów, co efektywnie zastępuje nieodżałowaną dyrektywę `#include`. Należy tylko pamiętać, że kompilator C może ograniczyć długość napisu (w ANSI C napis może mieć co najwyżej 508 znaków), a więc dłuższe szadery trzeba podzielić na więcej kawałków.

[4] Istnieje rozszerzenie języka GLSL udostępniające dyrektywę `#include` (nieidentyczną z dyrektywą `#include` preprocesora C), ale przecież chcemy dbać o przenośność.

[5] ¡No pasarán!

Listing 4.4. Ten sam przykładowy szader wierzchołków w programie w C

GLSL w C

```
                                        static const char sh_version430[] =
  "#version 430\n"
                                          ;
                                        static const char sh_gc[] =
  "uniform MyGC {"
      "mat4 pm;"      /* macierz rzutowania */
      "vec4 fg, bk;"  /* kolory obiektu i tła */
    "} gc;"
                                          ;
                                        static const char sh_vert[] =
  "layout(location=0) in vec4 in_Position;"

  "void main ( void )"
  "{"
    "gl_Position = gc.pm * in_Position;"
  "}" /*main*/
                                        ;
static const char *shader_v_src[] =
  { sh_version430, sh_gc, sh_vert };
```

4.4. Kompilowanie szaderów i łączenie programów

Procedura `CompileShaderStrings` pokazana na listingu 4.5 otrzymuje następujące parametry: `shader_type` musi być jedną z sześciu określających typ szadera stałych o nazwach `GL_VERTEX_SHADER`, `GL_TESS_CONTROL_SHADER`, `GL_TESS_EVALUATION_SHADER`, `GL_GEOMETRY_SHADER`, `GL_FRAGMENT_SHADER` lub `GL_COMPUTE_SHADER`; `nsl` jest liczbą napisów — fragmentów kodu źródłowego szadera; parametr `srclines` jest wskaźnikiem tablicy wskaźników początków tych napisów, które (w tym podprogramie) muszą być zakończone bajtem zerowym (czyli być napisami ASCIIZ).

Zadaniem procedury `CompileShaderStrings` jest utworzenie szadera, skompilowanie go i wypisanie komunikatów diagnostycznych, jeśli wystąpił błąd kompilacji. Pierwsze zadanie wykonuje procedura `glCreateShader`, która przekazuje identyfikator obiektu szadera. Jego kod źródłowy rejestruje w tym obiekcie procedura `glShaderSource`.

Kompilacja jest wykonywana przez procedurę `glCompileShader`. W linii 13 następuje sprawdzenie, czy kompilacja zakończyła się sukcesem. Jeśli kompilator wykrył błędy, to jego komunikat diagnostyczny jest wyciągany przez procedurę `glGetShaderInfoLog` (wcześniej ustala się długość tego komunikatu) i wypisywany do pliku `stderr`, a potem szader jest kasowany i procedura przekazuje `0`, czyli pusty identyfikator szadera. Jeśli kompilacja szadera zakończyła się sukcesem, to jako wartość procedury jest przekazywany jego identyfikator. Aby użyć tej procedury do skompilowania szadera z listingu 4.4, należy ją wywołać w taki sposób:

```
s_id[i] = CompileShaderStrings ( GL_VERTEX_SHADER, 3, shader_v_src,
                                 NULL );
```

Ostatni parametr może wskazywać tablicę dowolnych napisów, których ma być tyle, z ilu części składa się tekst szadera; to mogą być nazwy plików źródłowych. Jeśli kompilator wykryje błędy, to te napisy zostaną wyprowadzone przed komunikatem diagnostycznym. Podczas uruchamiania aplikacji korzystającej z wielu szaderów ułatwi to odnalezienie szadera z błędem.

W tablicy `s_id`, której elementy są typu `GLuint`, zapamiętujemy identyfikatory wszystkich szaderów, które chcemy połączyć w program; tablicę tę przekażemy następnie jako parametr procedury `LinkShaderProgram`.

Listing 4.5. Procedura `CompileShaderStrings`

C

```
1: GLuint CompileShaderStrings ( GLenum shader_type, int nsl,
2:                               const GLchar **srclines, const char **names )
3: {
4:   GLuint shader_id;
5:   GLint  success, logsize;
6:   GLchar *log;
7:   int    i;
8:
9:   if ( (shader_id = glCreateShader ( shader_type )) != 0 ) {
10:    glShaderSource ( shader_id, nsl, srclines, NULL );
11:    glCompileShader ( shader_id );
12:    glGetShaderiv ( shader_id, GL_COMPILE_STATUS, &success );
13:    if ( !success ) {
14:      if ( names ) {
15:        fprintf ( stderr, "shader: " );
16:        for ( i = 0; i < nsl; i++ )
17:          fprintf ( stderr, "%s%s", names[i], i < nsl-1 ? ", " : "\n" );
18:      }
19:      glGetShaderiv ( shader_id, GL_INFO_LOG_LENGTH, &logsize );
20:      if ( logsize > 1 ) {
21:        if ( (log = malloc ( logsize+1 )) != 0 ) {
22:          glGetShaderInfoLog ( shader_id, logsize, &logsize, log );
23:          fprintf ( stderr, "\%s\n", log );
24:          free ( log );
25:        }
26:      }
27:      glDeleteShader ( shader_id );
28:      return 0;
29:    }
30:  }
31:  ExitIfGLError ( "CompileShaderStrings" );
32:  return shader_id;
33: } /*CompileShaderStrings*/
```

Listing 4.6 przedstawia procedurę, która czyta pliki o podanych nazwach, umieszcza ich zawartość w odpowiedniej tablicy i wywołuje opisaną wyżej procedurę `CompileShaderStrings`. Pierwszy parametr określa typ szadera, drugi liczbę plików, a trzeci jest tablicą napisów ASCIIZ, które są nazwami tych plików.

Listing 4.6. Procedura `CompileShaderFiles`

C

```
GLuint CompileShaderFiles ( GLenum shader_type, int nfiles,
                            const char **filenames )
{
  GLuint shader_id = 0;
  FILE   *f;
  int    i;
  GLint  *fsize = NULL, totalsize;
  GLchar *src = NULL, **srclines = NULL;

  if ( !(fsize = malloc ( nfiles*sizeof(GLint) )) ||
       !(srclines = malloc ( nfiles*sizeof(GLchar*) )) )
    goto way_out;
  for ( i = 0, totalsize = nfiles;  i < nfiles;  i++ ) {
    if ( !(f = fopen ( filenames[i], "rb" )) )
      goto way_out;
    fseek ( f, 0, SEEK_END );
    fsize[i] = ftell ( f );
    totalsize += fsize[i];
    fclose ( f );
  }
  if ( !(src = malloc ( totalsize )) )
    goto way_out;
  for ( i = 0, totalsize = 0;  i < nfiles;  i++ ) {
    if ( !(f = fopen ( filenames[i], "rb" )) )
      goto way_out;
    srclines[i] = &src[totalsize];
    if ( fread ( srclines[i], sizeof(char), fsize[i], f ) != fsize[i] )
      goto way_out;
    srclines[i][fsize[i]] = 0;
    totalsize += fsize[i]+1;
    fclose ( f );
  }
  shader_id = CompileShaderStrings ( shader_type, nfiles,
                                     (const GLchar**)srclines, filenames );
way_out:
  if ( fsize )    free ( fsize );
  if ( srclines ) free ( srclines );
  if ( src )      free ( src );
  return shader_id;
} /*CompileShaderFiles*/
```

Instrukcje w liniach 13–20 znajdują długości tych plików (w bajtach). Następnie jest rezerwowany odpowiedni bufor, po czym w pętli w liniach 23–32 pliki są czytane do bufora i tworzona jest tablica wskaźników do początków przeczytanych napisów. Jako ostatni parametr procedury `CompileShaderStrings` jest podawany wskaźnik tablicy z nazwami plików; nazwy te zostaną więc wypisane w razie błędu kompilacji. Procedura `CompileShaderFiles` przekazuje jako wartość identyfikator skompilowanego szadera.

Listing 4.7. Procedura `LinkShaderProgram`

C

```
1: GLuint LinkShaderProgram ( int nsh, const GLuint *shaders,
2:                            const char *name )
3: {
4:   GLuint program_id;
5:   int    i;
6:   GLint  success, logsize;
7:   GLchar *log;
8:
9:   if ( (program_id = glCreateProgram ()) ) {
10:    for ( i = 0; i < nsh; i++ )
11:      glAttachShader ( program_id, shaders[i] );
12:    glLinkProgram ( program_id );
13:    glGetProgramiv ( program_id, GL_LINK_STATUS, &success );
14:    if ( !success ) {
15:      if ( name )
16:        fprintf ( stderr, "shader program: \%s\n", name );
17:      glGetProgramiv ( program_id, GL_INFO_LOG_LENGTH, &logsize );
18:      if ( logsize > 1 ) {
19:        if ( (log = malloc ( logsize+1 )) ) {
20:          glGetProgramInfoLog ( program_id, logsize, &logsize, log );
21:          fprintf ( stderr, "\%s\n", log );
22:          free ( log );
23:        }
24:      }
25:      glDeleteProgram ( program_id );
26:      return 0;
27:    }
28:  }
29:  ExitIfGLError ( "LinkShaderProgram" );
30:  return program_id;
31: } /*LinkShaderProgram*/
```

Procedura `LinkShaderProgram` na listingu 4.7 tworzy nowy program szaderów, dopisuje do niego skompilowane szadery i łączy program szaderów. Parametr `nsh` jest liczbą szaderów, których identyfikatory (otrzymane jako wartości procedury `glCreateShader`, a następnie przekazane przez `CompileShaderStrings`) są zapamiętane w tablicy `shaders`. Procedura `glCreateProgram` tworzy nowy, początkowo pusty obiekt programu i podaje

jego identyfikator. Szadery są doczepiane do programu przez `glAttachShader`. Łączenie wykonuje procedura `glLinkProgram`; jeśli wystąpiły błędy (to jest sprawdzane w linii 14), to procedura wyciąga tekst diagnostyczny i wypisuje go do pliku `stderr`, poprzedzając go nazwą programu, tj. napisem podanym jako ostatni parametr, a potem likwiduje nieudany program. Wartością procedury `LinkShaderProgram` jest identyfikator programu, przy czym liczba `0` jest identyfikatorem pustym — każdy szader, program szaderów, bufor, tekstura i dowolny inny obiekt utworzony przez aplikację ma identyfikator dodatni. Program może być uaktywniony przez wywołanie procedury `glUseProgram` z tym identyfikatorem podanym jako parametr.

Po zbudowaniu programu szaderów poszczególne obiekty szaderów w zasadzie przestają być potrzebne i można je zlikwidować; służy do tego procedura `glDeleteShader` (podobnie, do likwidowania niepotrzebnych programów służy procedura `glDeleteProgram`). Ale jeden szader może być częścią wielu programów — wystarczy go skompilować tylko raz. Ponadto programy można przebudowywać w trakcie działania aplikacji, usuwając szadery z programu za pomocą procedury `glDetachShader` i dodając nowe (skompilowane) szadery za pomocą `glAttachShader`. Po takiej zmianie program trzeba na nowo połączyć za pomocą procedury `glLinkProgram`[6].

4.5. *Uzupełnienia

Czytelnicy, którzy dopiero zapoznają się z OpenGL-em, być może docenią ten podrozdział, czytając tę książkę któryś kolejny raz[7].

4.5.1. SPIR-V

SPIR-V jest to binarny format (a właściwie język) dla szaderów i innych programów działających na GPU. Kod źródłowy szaderów napisanych w GLSL-u (lub w innym języku) można skompilować przy użyciu zewnętrznego kompilatora, otrzymując ich reprezentację w kodzie SPIR-V i rozpowszechniać je w takiej postaci[8]; jest ona niezależna od architektury GPU, czyli tak przetworzone szadery powinny działać na GPU różnych producentów[9]. Należy mieć na uwadze, że

- język SPIR-V nie zawiera dokładnych odpowiedników wszystkich konstrukcji języka GLSL, a kompilator dodatkowo ogranicza ich zestaw[10], a zatem nie realizuje pełnej specyfikacji GLSL-a, wskutek czego

[6] Indeksy zmiennych jednolitych i ich bloków zmieniają się przy tym, zatem po każdym połączeniu programu trzeba je na nowo z niego odczytać.

[7] Tym zdaniem uprzedzam, że to jest podrozdział z gwiazdką.

[8] nieczytelnej dla ludzi, choć możliwej do zdeasemblowania

[9] co koniecznie trzeba przetestować *przed* rozpowszechnianiem programu

[10] Chciałbym wierzyć, że to się zmieni. Tymczasem odkryłem, że kompilator GLSL-a wbudowany w bibliotekę `libGL.so` dostarczoną przez producenta mojej karty graficznej ma lepszą diagnostykę błędów niż opisany niżej kompilator zewnętrzny.

- nie każda poprawna konstrukcja języka GLSL jest dopuszczalna w szaderach, które mają być skompilowane do SPIR-V, co więcej
- kod w SPIR-V jest zwykle dłuższy niż kod źródłowy szadera w GLSL-u (chyba że ten ostatni jest obficie skomentowany), ale
- korzyścią z tej postaci jest skrócenie czasu uruchamiania aplikacji (o czas kompilowania szaderów, niezbyt długi w aplikacjach opisanych w tej książce) i możliwość pisania aplikacji dla okrojonych środowisk, w których nie ma kompilatora GLSL-a, działających na przykład w urządzeniach przenośnych (telefonach), a ponadto
- SPIR-V jest podstawową (a właściwie wymaganą) reprezentacją szaderów w aplikacjach standardu Vulkan.

W opisanych dalej aplikacjach będziemy korzystać z procedur opisanych wcześniej w tym rozdziale. Umożliwiają one aplikacji czytanie plików źródłowych w GLSL-u i ich kompilowanie i łączenie; taka postać jest najłatwiejsza do uruchamiania szaderów i aplikacji. Po uruchomieniu i przetestowaniu można jednak posłużyć się opisanym niżej sposobem, aby skompilować szadery razem z aplikacją, a nawet wbudować do niej szadery skompilowane (można je przetworzyć na tekst w C, zawierający deklaracje tablic liczb typu `unsigned int` podanych np. w układzie szesnastkowym), dzięki czemu po skompilowaniu aplikacji z szaderami powstanie jeden plik binarny.

Kompilator i sposób jego instalowania można znaleźć w Internecie na stronie [10]; plik wykonywalny kompilatora nazywa się `glslangValidator`. Kod źródłowy w GLSL-u należy zapisać w pliku, którego nazwa kończy się jednym z sześciu rozszerzeń: `.vert`, `.tesc`, `.tese`, `.geom`, `.frag` albo `.comp`, po którym kompilator rozpoznaje rodzaj szadera[11]. Przypuśćmy, że kod źródłowy szadera jest w pliku `app1b0.vert.glsl` i nie ma w nim błędów. Polecenie kompilacji tego szadera może mieć postać

```
glslangValidator -G app1b0.vert.glsl -o app1b0.vert.spv
```

Opcja `-G` wybiera postać kodu wynikowego odpowiednią do współpracy z aplikacją OpenGL-a (alternatywna opcja `-V` obowiązuje dla aplikacji Vulkana). Po skompilowaniu powstaje plik `app1b0.vert.spv`. Kompilator wymaga, aby szader zawierał procedurę `main` i wszystkie wywoływane przez nią podprogramy (oprócz wbudowanych, zobacz podrozdz. 9.13), co uniemożliwia osobną kompilację plików źródłowych składających się na jeden szader[12].

Opcja `-H` powoduje wypisanie skompilowanego kodu w postaci tekstu; można go przeczytać, aby zaspokoić ciekawość lub poddać szader drobiazgowej analizie. Minusem tej opcji jest to, że włącza ona opcję `-V`, a zatem pisząc szader dla aplikacji OpenGL-a, trzeba spełnić też wszystkie ograniczenia obowiązujące szadery przeznaczone do współpracy z Vulkanem.

[11] Obecnie dostępny kompilator dopuszcza używanie rozszerzeń podwójnych, `.vert.glsl`, `.tesc.glsl` itd. Wersja wcześniejsza (najnowsza w czasie, gdy pisałem pierwsze wydanie) zakładała, że rodzaj szadera jest określony przez ostatnie rozszerzenie nazwy pliku.

[12] Można kompilować osobne pliki i wtedy zamiast skompilowanego szadera dostaniemy komunikat o błędzie. Ale jeśli w plikach są inne błędy, to one też zostaną wykryte i opisane.

W aplikacji, która ma korzystać z szaderów skompilowanych w ten sposób, należy podczas inicjalizacji sprawdzić, czy obecna w systemie implementacja OpenGL-a jest zgodna ze specyfikacją co najmniej 4.6, a jeśli nie, to czy obsługuje rozszerzenie standardu o nazwie `"GL_ARB_gl_spirv"`. Możemy w tym celu użyć procedury przedstawionej na listingu 4.8, wywołując ją z parametrem — napisem będącym nazwą rozszerzenia. W linii 6 procedura dowiaduje się, ile rozszerzeń jest obecnych w implementacji, a następnie w pętli otrzymuje adresy kolejnych nazw rozszerzeń i porównuje je z nazwą podaną jako parametr. Procedura kończy działanie po znalezieniu podanej nazwy lub po jej nieznalezieniu.

Listing 4.8. Procedura `IsGLExtensionPresent`

C

```
1: char IsGLExtensionPresent ( const char *name )
2: {
3:   GLint         extn, i;
4:   const GLubyte *ext;
5:
6:   glGetIntegerv ( GL_NUM_EXTENSIONS, &extn );
7:   for ( i = 0; i < extn; i++ ) {
8:     ext = glGetStringi ( GL_EXTENSIONS, i );
9:     if ( !strcmp ( name, (char*)ext ) )
10:      return true;
11:  }
12:  return false;
13: } /*IsGLExtensionPresent*/
```

Jeśli rozszerzenie `"GL_ARB_gl_spirv"` jest obecne, to aplikacja może działać dalej. W tym celu będzie potrzebować procedury `glSpecializeShaderARB`. Jeśli do łączenia aplikacji z biblioteką GL nie używamy biblioteki glad lub wymagamy specyfikacji wcześniejszej niż 4.6, to adres tej procedury trzeba „wyciągnąć” indywidualnie (bo tego nie robi procedura `glewInit` ani `gl3wInit`, ich zadanie polega na uzyskaniu dostępu do procedur niebędących częściami rozszerzeń[13]). W aplikacji trzeba zadeklarować typ i zmienną

```
typedef void (*PFNGLSPECIALIZESHADERARB) (GLuint shader,
            const GLchar* pEntryPoint, GLuint numSpecializationConstants,
            const GLuint* pConstantIndex, const GLuint* pConstantValue);
PFNGLSPECIALIZESHADERARB glSpecializeShaderARB = NULL;
```

[13] Dodatkowa przewaga bibliotek gl3w i glad nad GLEW polega na włączeniu (poprzez plik `gl3w.h` lub `glad.h`) deklaracji i makrodefinicji związanych z jawnie wskazaną wersją standardu OpenGL lub ze wszystkimi jego wersjami. W moim przypadku wersja obsługiwana przez sprzęt i sterownik to 4.6, ale ograniczyłem wymagania aplikacji do wersji 4.5. Próbując zbudować aplikację pierwszą C (rozdział 11) z szaderami SPIR-V i z biblioteką GLEW, otrzymałem błąd kompilacji spowodowany brakiem definicji stałej `GL_SHADER_BINARY_FORMAT_SPIR_V_ARB`, potrzebnej jako parametr procedury `glShaderBinary`, a to dlatego, że plik nagłówkowy `glew.h` zawierał definicje tylko dla wersji OpenGL 1.0–4.5. Aby pokonać tę przeszkodę, wystarczy dopisać odpowiednią makrodefinicję do kodu aplikacji. Skąd wziąć stałą? Z dokumentacji rozszerzenia ze strony [6] lub z pliku `glcorearb.h`. Chyba jednak lepiej jest używać biblioteki gl3w lub glad.

Zmiennej `glSpecializeShaderARB` należy przypisać wartość podaną (zależnie od używanej biblioteki współpracującej z systemem okien) przez procedurę `glXGetProcAddress`, `wglGetProcAddress`, `glutGetProcAddress` albo `glfwGetProcAddress`.

Na listingu 4.9 są pokazane dwie procedury; pierwsza z nich tworzy szader z danych przekazanych jako parametr, a druga czyta dane z pliku (który powinien był zostać utworzony przez program `glslangValidator`) i wywołuje pierwszą procedurę. Każda z tych procedur zwraca identyfikator szadera, który należy zapamiętać w tablicy, a następnie wywołać procedurę `LinkShaderProgram` z listingu 4.7. Dalsze postępowanie jest takie samo jak z szaderami otrzymanymi w wyniku kompilacji tekstów źródłowych w GLSL-u podanych przez aplikację.

Procedura `glShaderBinary` (wywoływana zamiast `glShaderSource`) może otrzymać tablicę z wieloma identyfikatorami szaderów, choć w przykładzie jest tylko jeden. Celem procedury `glSpecializeShaderARB` jest określenie nazwy punktu wejściowego szadera (w tym przykładzie jest to procedura `main`) oraz nadanie wartości tzw. **stałym specjalizacji** szadera (temu służą ostatnie trzy parametry, których wartości w przykładzie na listingu 4.9 oznaczają zaniechanie tej czynności). Stałe specjalizacji mogą na przykład wyznaczać wielkości pewnych tablic lub zapotrzebowanie na inne zasoby (np. liczbę używanych tekstur), które można ustalić w chwili instalowania szadera w postaci SPIR-V w kontekście OpenGL-a na początku działania aplikacji (co umożliwia dostosowanie szadera do sprzętu, na którym ta aplikacja zostaje uruchomiona). Dalsze wiadomości na ten temat daleko wykraczają poza zakres tego kursu; zainteresowanych Czytelników odsyłam do strony [6] i innych źródeł wiedzy dostępnych w Internecie.

Listing 4.9. Procedury `CreateSPIRVShader` i `LoadSPIRVFile`

C

```
1: GLuint CreateSPIRVShader ( GLenum shader_type,
2:                            int size, const GLuint *spirv )
3: {
4:   GLuint shader_id;
5:   GLint  success;
6:
7:   if ( (shader_id = glCreateShader ( shader_type )) != 0 ) {
8:     glShaderBinary ( 1, &shader_id, GL_SHADER_BINARY_FORMAT_SPIR_V_ARB,
9:                      spirv, size );
10:    glSpecializeShaderARB ( shader_id, "main", 0, NULL, NULL );
11:    glGetShaderiv ( shader_id, GL_COMPILE_STATUS, &success );
12:    if ( !success ) {
13:      glDeleteShader ( shader_id );
14:      shader_id = 0;
15:    }
16:  }
17:  ExitIfGLError ( "CompileSPIRVString" );
18:  return shader_id;
19: } /*CreateSPIRVShader*/
20:
```

```
21: GLuint LoadSPIRVFile ( GLenum shader_type, const char *filename )
22: {
23:   GLuint shader_id = 0;
24:   FILE   *f;
25:   int    size;
26:   GLuint *spirv;
27:
28:   if ( !(f = fopen ( filename, "rb" )) )
29:     return 0;
30:   fseek ( f, 0, SEEK_END );
31:   size = ftell ( f );
32:   rewind ( f );
33:   if ( !(spirv = malloc ( size )) )
34:     goto way_out;
35:   if ( fread ( spirv, sizeof(char), size, f ) != size )
36:     goto way_out;
37:   shader_id = CreateSPIRVShader ( shader_type, size, spirv );
38: way_out:
39:   fclose ( f );
40:   if ( spirv ) free ( spirv );
41:   return shader_id;
42: } /*LoadSPIRVFile*/
```

W punkcie 11.5.2 jest przykład przeróbek szadera, które były konieczne w celu uruchomienia go w aplikacji korzystającej z kodu SPIR-V.

4.5.2. Opcje programów szaderów

Po utworzeniu obiektu programu szaderów, przed łączeniem programu, tj. przed wywołaniem procedury `glLinkProgram`, możemy włączyć pewne opcje[14]. Służy do tego procedura `glProgramParameteri`, która ma trzy parametry: identyfikator programu, nazwę opcji i argument opcji o wartości `GL_FALSE` (który opcję wyłącza) albo `GL_TRUE` (który ją włącza). Są dwie opcje, domyślnie obie są wyłączone.

Włączenie opcji `GL_PROGRAM_BINARY_RETRIEVABLE_HINT` umożliwia późniejsze odczytanie przez aplikację z kontekstu OpenGL-a złączonego programu szaderów, tj. otrzymanie jego kopii w odpowiedniej tablicy, której zawartość aplikacja może zapamiętać w pliku, aby później (np. przy następnym wywołaniu) odczytać program z pliku i użyć go bez ponownego kompilowania i łączenia. To rozwiązanie jest bardziej radykalne niż użycie szaderów w formacie SPIR-V. Może ono się przydać, jeśli aplikacja używa bardzo wielu rozbudowanych programów szaderów, których kompilacja i łączenie trwają odczuwalnie długo. Trzeba jednak podkreślić, że programy szaderów w takiej postaci *są nieprzenośne*. Przedstawiana tu opcja przydaje się podczas instalowania aplikacji (np. gry lub pakietu CAD) w danym sprzęcie. Można wtedy jednorazowo skompilować, złączyć i zapamiętać wszystkie potrzebne programy szaderów.

[14] Wymaga to drobnych modyfikacji procedury `LinkShaderProgram` z listingu 4.7.

Aby zapamiętać program, po włączeniu tej opcji i złączeniu programu należy wywołać procedurę `glGetProgramiv`, podając jako parametry identyfikator programu, stałą `GL_PROGRAM_BINARY_LENGTH` i adres zmiennej całkowitej, której zostanie przypisana długość programu w bajtach. Binarny kod programu wpisze do podanej tablicy procedura `glGetProgramBinary`. Ma ona 5 parametrów: identyfikator programu, długość w bajtach tablicy (która powinna być nie krótsza niż liczba bajtów programu), adres zmiennej, której długość programu (ta sama, co odczytana wcześniej przez `glGetProgramiv`) będzie przypisana, adres zmiennej typu `GLenum`, której zostanie przypisany **identyfikator binarnego formatu** programu, i adres początku tablicy, do której zostanie wpisany kod programu.

Aby otrzymany w ten sposób kod przygotować do pracy (np. po odczytaniu go z pliku), trzeba utworzyć nowy obiekt programu (za pomocą procedury `glCreateProgram`), a następnie użyć procedury `glProgramBinary`. Jej parametry to identyfikator programu, identyfikator binarnego formatu (koniecznie trzeba go też zapamiętać po wywołaniu procedury `glGetProgramBinary`), adres początku tablicy z kodem programu i jego długość.

Druga opcja, którą można włączyć przed przystąpieniem do łączenia programu szaderów, ma nazwę `GL_PROGRAM_SEPARABLE`. Ma ona na celu łączenie niekompletnych programów szaderów, tj. takich, które nie zawierają *wszystkich* szaderów potrzebnych do utworzenia obrazu. Szadery w takich programach powinny zawierać dyrektywę

```
#extension GL_ARB_separate_shader_objects : enable
```

Aby ich użyć, trzeba utworzyć jeden lub więcej **obiektów potoku programów** (*program pipelines*), wywołując procedurę `glGenProgramPipelines`. Poszczególne programy należy zarejestrować w takim obiekcie, do czego służy procedura `glUseProgramStages`. Wreszcie, aby użyć tak utworzonego potoku programów, zamiast `glUseProgram` trzeba wywołać procedurę `glBindProgramPipeline`, której parametr jest identyfikatorem obiektu.

Uwaga: Poszczególne programy ustawione w rozważane tu potoki mają *osobne* domyślne bloki zmiennych jednolitych, wskutek czego szadery wchodzące w skład każdego z tych programów nie mają dostępu do zmiennych jednolitych pozostałych programów. Programując szadery, trzeba skrupulatnie uzgodnić ich zmienne interfejsu, za pomocą których dane są przekazywane do kolejnych programów ustawionych w potok, bo ewentualne błędy (niezgodności interfejsu) nie są sygnalizowane przez procedurę łączenia programu szaderów.

W tej książce nie rozwijam przedstawionych w tym punkcie tematów, odsyłając Czytelników do bardziej zaawansowanej literatury (np. [23]) i do specyfikacji OpenGL-a.

4.5.3. Przechwytywanie wywołań procedur OpenGL-a

Wykonywanie programu krok po kroku połączone z podglądaniem wartości zmiennych przy użyciu narzędzi zwanych po indoeuropejsku *debuggerami* jest bardzo skutecznym sposobem wyszukiwania błędów, ale w pewnych przypadkach jeszcze skuteczniejsza bywa klasyczna technika zwana **wydrukami kontrolnymi** — teksty wypisywane przez (tymczasowo) dodane w odpowiednich miejscach instrukcje stanowią zapis działania programu, który może być

poddany bardziej dogłębnej analizie niż bieżący stan programu wyświetlany przez debugger. Im większa aplikacja, tym ważniejsze jest jej gruntowne przebadanie i tym bardziej pracochłonne i kłopotliwe jest dopisanie instrukcji uruchomieniowych *wszędzie*, gdzie są potrzebne. Dość skomplikowany sposób łączenia aplikacji z bibliotekami OpenGL-a, opisany w rozdziale 2, umożliwia jednak przechwycenie wybranych procedur dla sprawdzenia, czy są one wywoływane we właściwej kolejności i z właściwymi parametrami. Zobaczmy przykład.

Przypuśćmy, że chcemy zbadać, czy aplikacja poprawnie sprząta pamięć GPU, tj. czy każdy bufor po utworzeniu i wykorzystaniu jest poprawnie likwidowany. W tym celu przechwycimy procedury `glGenBuffers` i `glDeleteBuffers`, powodując, by *każde* wywołanie tych procedur wiązało się z wywołaniem odpowiedniej procedury, która wypisze miejsce wywołania i wartości parametrów. Do tego służą makrodefinicje na listingu 4.10, umiesz-

Listing 4.10. Makrodefinicje przechwytujące procedury rezerwacji buforów

C

```
#define DEBUG_BUFFERS_ALLOCATION  /* po uruchomieniu można wykomentować */

#ifdef DEBUG_BUFFERS_ALLOCATION
#ifdef USE_GL3W
#define t_glGenBuffers    gl3w_glGenBuffers
#define t_glDeleteBuffers gl3w_glDeleteBuffers
#else
#ifdef USE_GLAD
#define t_glGenBuffers    glad_glGenBuffers
#define t_glDeleteBuffers glad_glDeleteBuffers
#else
#error "use glad or gl3w"
#endif
#endif

void traceglGenBuffers ( GLsizei n, GLuint *buffers, char *file, int line );
#undef glGenBuffers
#define glGenBuffers(n,b) { \
  t_glGenBuffers ( n, b ); \
  traceglGenBuffers ( n, b, __FILE__, __LINE__ ); \
}

void traceglDeleteBuffers ( GLsizei n, GLuint *buffers,
                            char *file, int line );
#undef glDeleteBuffers
#define glDeleteBuffers(n,b) { \
  traceglDeleteBuffers ( n, b, __FILE__, __LINE__ ); \
  t_glDeleteBuffers ( n, b ); \
}

void DumpBufferIdentifiers ( void );
#endif
```

czone w dodatkowym pliku nagłówkowym, który za pomocą dyrektywy `#include` można włączyć do pliku `utilities.h`, włączanego do *wszystkich* plików źródłowych aplikacji.

Plik nagłówkowy każdej z bibliotek glad, gl3w lub GLEW zawiera deklaracje zmiennych, których wartości są adresami procedur w bibliotece GL, oraz makrodefinicje, których nazwy to nazwy procedur używanych w aplikacji. W liniach 17 i 25 makrodefinicje te są usuwane, po czym następują nowe makrodefinicje o tych nazwach, zawierające wywołania procedur wskazywanych przez wspomniane zmienne oraz wywołania procedur śledzących rezerwowanie i likwidowanie buforów. Makrodefinicje `__FILE__` i `__LINE__`, podane jako parametry tych procedur, kompilator zamienia na nazwę pliku źródłowego i numer linii, w której zostaje wywołana przechwycona procedura, dzięki czemu informacje te znajdą się w wydruku kontrolnym.

Uwaga: Jeśli przechwycona procedura jest wywoływana w instrukcji warunkowej, takiej że po jej wywołaniu występuje słowo kluczowe `else`, to trzeba usunąć średnik po liście parametrów albo (lepiej) ująć wywołanie procedury w klamry — drobne zmiany kodu źródłowego na potrzeby uruchamiania mogą być więc konieczne, kompilator wskaże odpowiednie miejsca.

Procedury śledzące mogą obsługiwać własny wykaz zarezerwowanych identyfikatorów buforów (niezależny od ukrytego przed aplikacją wykazu buforów będącego częścią kontekstu OpenGL-a); wypisanie zawartości tego wykazu w chwili zatrzymania aplikacji (przez procedurę `DumpBufferIdentifiers`, której prototyp jest w linii 31) umożliwi sprawdzenie, czy ona zostawiła po sobie porządek, tj. zlikwidowała wszystkie bufory.

Podany wyżej przykład można (i warto) rozbudować. Przechwycenie wywołań procedur `glBindBuffer`, `glBindBufferBase`, `glBufferData`, `glBufferSubData` i `glGetBufferSubData` umożliwia aplikacji sprawdzanie, czy operacje przesyłania danych między pamięcią CPU a buforem nie powodują pisania lub czytania danych za końcem bufora.

Przechwytywanie procedur OpenGL-a może też przydać się podczas pracy z debuggerem; dodatkowe procedury wykonywane przy ich wywołaniach mogą przeprowadzać dowolne obliczenia i testy, których wyniki będzie można podglądać na bieżąco [15].

W następnym punkcie jest przedstawione inne narzędzie do wyszukiwania błędów w aplikacjach OpenGL-a: konteksty uruchomieniowe. Pozwalają one na zarejestrowanie procedury, która będzie wywoływana natychmiast po wykryciu błędu lub innej sytuacji podbramkowej, co umożliwia wzięcie uruchamianej aplikacji pod lupę. Ale, rozwijając tę metaforę, przechwytywanie wywołań procedur OpenGL-a sposobem opisanym wyżej można uznać za mikroskop, przyrząd cięższy niż lupa, ale dostarczający znacznie bardziej szczegółowych obrazów. Procedura `glGetError` i procedura diagnostyczna używana w kontekście uruchomieniowym służą do znajdowania miejsca, w którym błąd się *objawił*. Przechwytywanie wywołań procedur OpenGL-a może ułatwić znalezienie miejsca, w którym błąd *powstał*.

[15] Programiści aplikacji standardu Vulkan mają do dyspozycji tzw. **warstwy walidacyjne** (*validation layers*), które przechwytują wywołania procedur biblioteki Vulkan w sposób podobny do opisanego w tym punkcie, aby ułatwić znajdowanie błędów. O ile wiem, nie ma publicznie dostępnych warstw dla OpenGL-a, a szkoda. Liczę w tej sprawie na Czytelników.

4.5.4. Uruchomieniowe konteksty OpenGL-a

… *Errare* — wszak — *humanum est.*
Jan Sztaudynger: *Rzecz ludzka*

Procedura `glGetError` jest narzędziem do wykrywania błędów w aplikacjach OpenGL-a użytecznym, ale niezbyt wygodnym, bo aby znaleźć instrukcję, w której błąd się objawia, trzeba wywoływać ją często, a informacje na temat natury błędu są przez nią dawkowane oszczędnie. W standardzie OpenGL 4.3 i nowszych istnieje możliwość utworzenia **kontekstu uruchomieniowego**. Kosztem pewnego spowolnienia obliczeń umożliwia on aplikacji zarejestrowanie procedury wywoływanej natychmiast po wykryciu błędu; parametry tej procedury podają bardziej szczegółowe informacje niż procedura `glGetError`. Po znalezieniu i poprawieniu wszystkich błędów można zmienić aplikację tak, aby korzystała ze „zwykłego" kontekstu OpenGL-a.

Aby utworzyć kontekst uruchomieniowy, aplikacja biblioteki FreeGLUT powinna poprzedzić utworzenie okna wykonaniem instrukcji

```
glutInitContextFlags ( GLUT_FORWARD_COMPATIBLE | GLUT_DEBUG );
```

Aplikacja biblioteki GLFW ma w tym celu wykonać instrukcję

```
glfwWindowHint ( GLFW_OPENGL_DEBUG_CONTEXT, GLFW_TRUE );
```

Natywna aplikacja systemu X Window utworzy kontekst uruchomieniowy, wywołując procedurę `InitGLXContext` z listingu 3.6 z (trzecim) parametrem `flags` równym

```
GLX_CONTEXT_DEBUG_BIT_ARB,
```

a w aplikacji systemu Windows tę samą rolę spełni procedura `InitWGLContext` (zobacz listing 3.16) wywołana z (czwartym) parametrem `flags` o wartości[16]

```
WGL_CONTEXT_DEBUG_BIT_ARB.
```

Przykładowa procedura wywoływana po wystąpieniu błędu jest przedstawiona na listingu 4.11. Można ją zarejestrować w kontekście OpenGL-a za pomocą instrukcji

```
glDebugMessageCallback ( DebugOutput, NULL );
```

Drugi parametr procedury `glDebugMessageCallback`, który może wskazywać dowolną strukturę danych aplikacji, będzie przekazywany procedurze `DebugOutput` jako jej ostatni parametr.

Procedura `glDebugMessageControl` umożliwia wybieranie powodów, dla których procedura `DebugOutput` ma być wywoływana. Pierwsze trzy parametry mogą przyjmować

[16] W obu przypadkach wartość parametru `flags` będzie ta sama, `0x00000001`, ale nie należy uważać tego za okazję do oszczędzania sobie pracy.

Listing 4.11. Procedura DebugOutput

C

```
void DebugOutput ( GLenum source, GLenum type, unsigned int id,
                   GLenum severity, GLsizei length, const char *message,
                   const void *userData )
{
  fprintf ( stderr, "Debug message %d: %s\n", id, message );
  switch ( source ) {
case GL_DEBUG_SOURCE_API:
    fprintf ( stderr, "API\n" );
    break;
case GL_DEBUG_SOURCE_WINDOW_SYSTEM:
    fprintf ( stderr, "System\n" );
    break;
case GL_DEBUG_SOURCE_SHADER_COMPILER:
    fprintf ( stderr, "Shader compiler\n" );
    break;
case GL_DEBUG_SOURCE_THIRD_PARTY:
    fprintf ( stderr, "Third party\n" );
    break;
case GL_DEBUG_SOURCE_APPLICATION:
    fprintf ( stderr, "Application\n" );
    break;
case GL_DEBUG_SOURCE_OTHER:
    fprintf ( stderr, "Other\n" );
    break;
default:
    break;
  }
  switch ( type ) {
case GL_DEBUG_TYPE_ERROR:
    fprintf ( stderr, "Error\n" );
    break;
case GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR:
    fprintf ( stderr, "Deprecated behaviour\n" );
    break;
case GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR:
    fprintf ( stderr, "Undefined behaviour\n" );
    break;
case GL_DEBUG_TYPE_PORTABILITY:
    fprintf ( stderr, "Portability\n" );
    break;
case GL_DEBUG_TYPE_PERFORMANCE:
    fprintf ( stderr, "Performance\n" );
    break;
case GL_DEBUG_TYPE_OTHER:
    fprintf ( stderr, "Other\n" );
```

```
46:     break;
47: default:
48:     break;
49:   }
50:   switch ( severity ) {
51: case GL_DEBUG_SEVERITY_HIGH:
52:     fprintf ( stderr, "Severity HIGH\n" );
53:     break;
54: case GL_DEBUG_SEVERITY_MEDIUM:
55:     fprintf ( stderr, "Severity MEDIUM\n" );
56:     break;
57: case GL_DEBUG_SEVERITY_LOW:
58:     fprintf ( stderr, "Severity LOW\n" );
59:     break;
60: case GL_DEBUG_SEVERITY_NOTIFICATION:
61:     fprintf ( stderr, "Severity notification\n" );
62:     break;
63: default:
64:     break;
65:   }
66: } /*DebugOutput*/
```

wartości nadawane parametrom `source`, `type` i `severity` (zobacz listing 4.11) lub wartość specjalną `GL_DONT_CARE`, a ostatni parametr, `GL_FALSE` albo `GL_TRUE`, odpowiednio blokuje albo uaktywnia wywoływanie procedury z parametrami o podanych wartościach. Zatem wykonanie instrukcji

```
glDebugMessageControl ( GL_DONT_CARE, GL_DONT_CARE, GL_DONT_CARE,
                        0, NULL, GL_TRUE );
```

spowoduje przekazywanie *wszystkich* możliwych komunikatów[17]. Aby zaś odbierać *tylko* komunikaty o poważnych błędach, trzeba wykonać instrukcje

```
glDebugMessageControl ( GL_DONT_CARE, GL_DONT_CARE, GL_DONT_CARE,
                        0, NULL, GL_FALSE );
glDebugMessageControl ( GL_DONT_CARE, GL_TYPE_ERROR, GL_SEVERITY_HIGH,
                        0, NULL, GL_TRUE );
```

Piątym parametrem procedury `glDebugMessageControl` może być tablica o długości podanej jako czwarty parametr; w tej tablicy należy podać identyfikatory (tj. numery) komunikatów, których odbieraniem autor aplikacji jest zainteresowany, albo niezainteresowany.

Uaktywnienie zarejestrowanej procedury następuje po wykonaniu instrukcji

```
glEnable ( GL_DEBUG_OUTPUT );
glEnable ( GL_DEBUG_OUTPUT_SYNCHRONOUS );
```

[17] których może być bardzo dużo; zalew informacji nie pomaga w znalezieniu tych istotnych

a do zatrzymania jej wywołań służy procedura `glDisable`. Możemy zatem wybierać także części aplikacji (np. procedury rysujące tylko niektóre obiekty), z których będą nadawane komunikaty.

Procedura `DebugOutput` może oczywiście wykonywać bardziej skomplikowane obliczenia niż tylko wypisywanie komunikatów. W szczególności podczas uruchamiania aplikacji przy użyciu debuggera możliwe jest ustawienie pułapki (*breakpoint*) na instrukcji wykonywanej po wystąpieniu błędu określonego rodzaju i zbadanie wartości zmiennych w chwili zatrzymania aplikacji w pułapce, a jeśli debugger to umożliwia, również obejrzenie zawartości stosu wywołań procedur aplikacji.

5

Działania na wektorach i macierzach

Blady atoli strach padał na wszystkich, kiedy król, dla niespodzianego kaprysu, ogłaszał zgadywanki. Z dawien dawna lubował się w nich i jeszcze wielkiego kanclerza podczas koronacji zaskoczył pytaniem, jak sądzi, czy pacierz i macierz różnią się czymś między sobą, a jeśli tak, to czym?

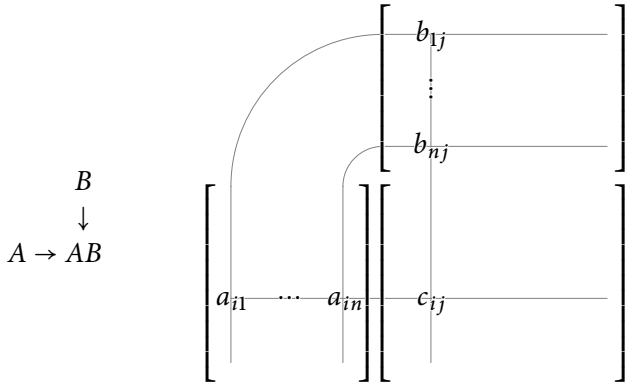
Stanisław Lem: *Cyberiada*

Działania na wektorach w $\mathbb{R}^4$ i macierzach 4×4 są nadzwyczaj intensywnie wykorzystywane w grafice trójwymiarowej. Wektory reprezentują punkty przestrzeni i wektory swobodne, a także wektory normalne płaszczyzn (które nie są wektorami swobodnymi), macierze zaś reprezentują przekształcenia afiniczne i rzutowe. Nie próbuję streścić w tym rozdziale wykładu z algebry liniowej z geometrią, ale staram się przypomnieć własności działań na macierzach i własności reprezentacji punktów i wektorów swobodnych w postaci współrzędnych kartezjańskich, barycentrycznych i jednorodnych oraz jednorodną reprezentację przekształceń afinicznych, albowiem mają one zasadnicze znaczenie dla działania wszelkich programów tworzących grafikę. Wprawdzie większość działań na wektorach i macierzach będzie wykonywać GPU, dla której to są operacje elementarne, ale podstawowy zestaw procedur w C, wykonujących te działania na CPU, jest też potrzebny. Dzięki niemu CPU może konstruować macierze przekształceń, które potem GPU zastosuje do tysięcy punktów.

5.1. Działania na macierzach

Znacie? To nie czytajcie[1]. Macierz rzeczywista $m \times n$ jest obiektem przedstawianym zazwyczaj jako tabela liczb (współczynników) złożona z m wierszy i n kolumn. Symbol a_{ij} oznacza współczynnik macierzy w i-tym wierszu i j-tej kolumnie. Dowolną macierz można **pomnożyć przez liczbę**, mnożąc przez nią wszystkie współczynniki. Dwie macierze o takich samych wymiarach można **dodać**, obliczając sumy współczynników na odpowiadających sobie pozycjach. Można też **pomnożyć macierze** A i B, jeśli mają one wymiary $m \times n$ i $n \times l$; **iloczyn**, macierz $C = AB$, ma wymiary $m \times l$ i współczynniki $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.

[1] Ale jeśli nie znacie, to owszem. Ewentualnie później, ale koniecznie.



Rysunek 5.1. Schemat Falka

Uwaga: W tym miejscu wiersze i kolumny macierzy są numerowane od 1, tak jak to się robi tradycyjnie w algebrze i w języku FORTRAN. W programach napisanych w C i w GLSL-u będziemy je numerować, zaczynając od 0.

Wiele o mnożeniu macierzy można się dowiedzieć, oglądając **schemat Falka** (rys. 5.1). Uwidocznia on, od których współczynników macierzy A i B zależy konkretny współczynnik c_{ij} ich iloczynu, dzięki czemu pewne własności mnożenia macierzy łatwiej jest dostrzec z tego schematu niż z definiującego to działanie wzoru. Przypomnijmy najważniejsze własności. Mnożenie macierzy jest działaniem **łącznym**, tj. zawsze $(AB)C = A(BC)$, ale istnieją takie macierze A i B, że $AB \neq BA$, jest to więc działanie **nieprzemienne**[2]. Natomiast mnożenie macierzy jest **rozdzielne względem dodawania**: $A(B+C) = AB+AC$, $(D+E)F = DF+EF$.

Macierz transponowana do A jest to macierz A^T, której wiersze i kolumny „zostały zamienione". Jeśli więc macierz A ma wymiary $m \times n$ i współczynniki a_{ij}, to A^T ma n wierszy i m kolumn i współczynniki a_{ji}. Dla dowolnych macierzy A, B, które można pomnożyć, zachodzi równość $(AB)^T = B^T A^T$.

Macierz jednostkowa $n \times n$ ma współczynniki $a_{ii} = 1$ dla $i = 1, \ldots, n$ oraz $a_{ij} = 0$ dla $i \neq j$. Oznaczamy ją symbolem I lub (jeśli trzeba zaznaczyć wymiary) I_n. Dla dowolnych macierzy A i B o wymiarach $m \times n$ i $n \times l$ jest $AI = A$ oraz $IB = B$.

Macierz kwadratowa ma tyle samo kolumn i wierszy. Macierz kwadratowa A jest **nieosobliwa**, jeśli istnieje taka macierz M, że $AM = I$. Wtedy także $MA = I$; macierz M jest nazywana **odwrotnością macierzy** A i jest oznaczana symbolem A^{-1}. Z łączności mnożenia macierzy wynika wzór na odwrotność iloczynu macierzy kwadratowych nieosobliwych: $(AB)^{-1} = B^{-1}A^{-1}$.

Macierz $m \times 1$ to tak zwana **macierz kolumnowa**; zbiór wszystkich takich macierzy (najczęściej nazywanych **wektorami**, choć to pojęcie jest znacznie szersze) to **przestrzeń liniowa** $\mathbb{R}^m$. Macierze kolumnowe są uporządkowanymi m-tkami liczb, które w tekście

[2]Istnienie iloczynu AB nie jest równoznaczne z istnieniem iloczynu BA, bo wymiary mogą „nie pasować". Jeśli $m = n = l > 1$, to oba iloczyny istnieją i mają te same wymiary $n \times n$, ale wtedy też może być $AB \neq BA$.

wygodniej jest zapisywać w postaci $(a_1, \ldots, a_m)$, pomijając zbędny indeks kolumny. Dalej wektory będą oznaczane pismem pogrubionym, np. $\boldsymbol{a}$, $\boldsymbol{v}_j$ itp. **Wektor zerowy**, oznaczony symbolem **0**, składa się z samych zer.

Wektory $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ są **liniowo zależne**, jeśli istnieją liczby $a_1, \ldots, a_k$, z których co najmniej jedna nie jest zerem, takie że $a_1\boldsymbol{v}_1 + \cdots + a_k\boldsymbol{v}_k = \mathbf{0}$. Jeśli takie liczby nie istnieją, to rozpatrywane wektory są **liniowo niezależne**. Macierz kwadratowa jest nieosobliwa, gdy jej kolumny są liniowo niezależne.

Ważnym pojęciem jest **wyznacznik**. Jest to funkcja (oznaczana symbolem det), która macierzy kwadratowej przyporządkowuje liczbę i która ma następujące własności: jeśli pierwszymi kolumnami macierzy A i B są wektory $\boldsymbol{a}_1$ i $\boldsymbol{b}_1$, pierwsza kolumna macierzy C jest równa $a\boldsymbol{a}_1 + b\boldsymbol{b}_1$ (dla dowolnych liczb a, b), a wszystkie pozostałe kolumny macierzy A są też odpowiednimi kolumnami macierzy B i C, to $\det C = a \det A + b \det B$; przestawienie dowolnych dwóch kolumn zmienia znak wyznacznika na przeciwny i wyznacznik macierzy jednostkowej jest równy 1. Jest tylko jedna funkcja o tych własnościach i (dla macierzy $n \times n$) można ją przedstawić wzorem

$$\det A = \sum_{\sigma \in S_n} \operatorname{sgn} \sigma \prod_{i=1}^{n} a_{i,\sigma(i)}, \tag{5.1}$$

w którym S_n oznacza zbiór wszystkich permutacji zbioru $\{1, \ldots, n\}$, a $\operatorname{sgn} \sigma$ to **znak permutacji** σ.[3]

Macierz A jest nieosobliwa wtedy i tylko wtedy, gdy jej wyznacznik nie jest zerem. Dla dowolnych macierzy A, B o wymiarach $n \times n$ jest spełniona równość $\det(AB) = \det A \det B$; fakt ten jest znany jako twierdzenie Cauchy'ego.

5.2. Punkty i wektory swobodne

Obiekty do narysowania konstruujemy w trójwymiarowej **przestrzeni afinicznej**, która składa się z **punktów**. Mamy w tej przestrzeni działanie **odejmowania punktów**; jego wynikiem jest **wektor swobodny**. Zbiór wszystkich wektorów swobodnych jest trójwymiarową **przestrzenią liniową**. Trzeba pamiętać, że przestrzeń afiniczna i przestrzeń liniowa są *różnymi* strukturami algebraicznymi, ponieważ w każdej z nich mamy określone inne działania — nawet jeśli reprezentacje punktów i wektorów w programie *mogą być* identyczne.

Elementy przestrzeni liniowej możemy mnożyć przez skalary (liczby rzeczywiste) i dodawać; w ogólności możemy tworzyć dowolne **kombinacje liniowe**, czyli wyrażenia postaci $\sum_{i=1}^{k} \boldsymbol{v}_i a_i$, gdzie $\boldsymbol{v}_i$ to wektory, a a_i to dowolne liczby. Wartość takiego wyrażenia jest wektorem. Mamy też wyróżniony **wektor zerowy**, **0**: dla każdego wektora $\boldsymbol{v}$ jest $\boldsymbol{v}+\mathbf{0} = \boldsymbol{v}$. Natomiast definicja przestrzeni afinicznej nie wyróżnia żadnego jej punktu. Działanie odejmowania punktów umożliwia określenie **dodawania wektora do punktu**; jego wynikiem jest punkt. Na tej podstawie można określić dalsze dwa działania na punktach. Jeśli $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_k$ to punkty przestrzeni afinicznej, to wyrażenie postaci $\sum_{i=0}^{k} \boldsymbol{p}_i a_i$ ma sens wtedy, gdy $a_0 + \cdots + a_k = 1$

[3] Jeśli liczba **inwersji**, czyli par liczb (i, j), takich że $i < j$ oraz $\sigma(i) > \sigma(j)$, jest parzysta, to $\operatorname{sgn} \sigma = +1$, w przeciwnym razie $\operatorname{sgn} \sigma = -1$.

albo $a_0 + \cdots + a_k = 0$. W pierwszym przypadku mamy punkt zwany **kombinacją afiniczną** punktów $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_k$ (o współczynnikach $a_0, \ldots, a_k$), a w drugim otrzymujemy **kombinację wektorową** (albo **różnicę uogólnioną**) punktów; jest ona wektorem swobodnym.

Dobre określenie działań kombinacji afinicznej i kombinacji wektorowej polega na tym, że wyniki tych działań są jednoznacznie określone i nie zależą od reprezentacji punktów i wektorów, w tym od rodzaju używanych współrzędnych ani od konkretnego układu współrzędnych. Na przykład punkt $\boldsymbol{c} = \frac{1}{3}\boldsymbol{p}_0 + \frac{1}{3}\boldsymbol{p}_1 + \frac{1}{3}\boldsymbol{p}_2$ jest zawsze tym samym środkiem ciężkości trójkąta o wierzchołkach $\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2$ i podobnie wektor $\boldsymbol{v} = -1\boldsymbol{p}_0 + \frac{1}{2}\boldsymbol{p}_1 + \frac{1}{2}\boldsymbol{p}_2$ określa przesunięcie, które przeprowadza wierzchołek $\boldsymbol{p}_0$ na środek przeciwległego boku tego trójkąta. Opisane niżej współrzędne umożliwiają szczególnie łatwe implementowanie tych działań.

5.3. Współrzędne kartezjańskie, jednorodne i barycentryczne

Układ współrzędnych kartezjańskich w trójwymiarowej przestrzeni afinicznej otrzymamy, wybierając **układ odniesienia**, na który składa się dowolny punkt $\boldsymbol{o}$ (**początek układu**) i trzy niezależne liniowo wektory swobodne $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, $\boldsymbol{v}_3$, zwane **wersorami osi układu**. Dla każdego punktu $\boldsymbol{p}$ przestrzeni istnieje jednoznacznie określona trójka liczb x, y, z, taka że $\boldsymbol{p} = \boldsymbol{o} + \boldsymbol{v}_1 x + \boldsymbol{v}_2 y + \boldsymbol{v}_3 z$. Układy odniesienia można wybierać na nieskończenie wiele sposobów, dlatego utożsamienie punktu $\boldsymbol{p}$ z wektorem $(x, y, z) \in \mathbb{R}^3$ jego współrzędnych kartezjańskich ma sens wtedy, gdy wiadomo, w jakim układzie te współrzędne są podane.

Opisane wyżej działania na punktach i wektorach możemy sprowadzić do rachunków na wektorach w przestrzeni $\mathbb{R}^3$ (czyli trójkach liczb), składających się ze współrzędnych kartezjańskich w ustalonym układzie współrzędnych. Dowolny wektor swobodny $\boldsymbol{v} = \boldsymbol{v}_1 x + \boldsymbol{v}_2 y + \boldsymbol{v}_3 z$ jest wtedy utożsamiony z trójką liczb (x, y, z). Jednak ta reprezentacja nie zawiera podstawowej informacji: co dana trójka liczb reprezentuje — punkt, czy wektor swobodny. W każdym układzie współrzędnych kartezjańskich początek tego układu jest reprezentowany przez wektor zerowy, a jego wersory osi odpowiednio przez wektory $\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3 \in \mathbb{R}^3$; i-ta współrzędna wektora $\boldsymbol{e}_i$ jest jedynką, a pozostałe dwie są zerem.

Punkt w przestrzeni trójwymiarowej można reprezentować również przez czwórkę liczb, (X, Y, Z, W), zwanych **współrzędnymi jednorodnymi**[4]; współrzędne kartezjańskie możemy z niej otrzymać w wyniku dzielenia:

$$x = \frac{X}{W}, \quad y = \frac{Y}{W}, \quad z = \frac{Z}{W}. \tag{5.2}$$

Ta reprezentacja jest oczywiście nadmiarowa: każdy punkt jest reprezentowany przez jeden wektor współrzędnych kartezjańskich i przez nieskończenie wiele wektorów współrzędnych jednorodnych; jeśli $W \neq 0$ i $a \neq 0$, to wektory (X, Y, Z, W) i (aX, aY, aZ, aW) reprezentują ten sam punkt. W szczególności, mając współrzędne kartezjańskie x, y, z punktu, otrzymamy jego współrzędne jednorodne, „doczepiając" jedynkę: $(x, y, z) \to (x, y, z, 1)$. Ostatnia współrzędna jednorodna (W) ma nazwę **współrzędnej wagowej** albo **wagi**.

[4] Należałoby mówić o współrzędnych kartezjańskich jednorodnych, ponieważ dalej „ujednorodnimy" też współrzędne barycentryczne, otrzymując współrzędne barycentryczne jednorodne. Ale dla skrótu słowo „kartezjańskie" przed słowem „jednorodne" będziemy pomijać.

Jeśli wektory współrzędnych jednorodnych reprezentujące punkty ograniczymy do takich, których współrzędna wagowa jest równa 1 (co oznacza, że pierwsze trzy współrzędne jednorodne są również współrzędnymi kartezjańskimi punktu), to odejmowanie punktów jest tożsame z odejmowaniem ich wektorów współrzędnych jednorodnych, przy czym wynik — reprezentacja wektora swobodnego — ma współrzędną wagową równą 0. Możemy też obliczać kombinacje liniowe wektorów swobodnych, dodawać wektory swobodne do punktów oraz obliczać kombinacje afiniczne i wektorowe punktów, wykonując odpowiednie działania liczbowe na wektorach współrzędnych jednorodnych. Wyniki tych działań mają współrzędną wagową 0 albo 1, dzięki czemu zawsze wiadomo, jakie obiekty (wektory swobodne czy punkty) reprezentują. Możliwość jednolitego reprezentowania punktów i wektorów swobodnych (pod warunkiem przestrzegania opisanego tu ograniczenia) jest jedną z wielu korzyści używania współrzędnych jednorodnych.

Współrzędne barycentryczne otrzymamy, wybierając układ odniesienia składający się z czterech punktów, $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, $\boldsymbol{p}_3$, nieleżących w jednej płaszczyźnie. Dowolny punkt $\boldsymbol{p}$ możemy wtedy zapisać w postaci

$$\boldsymbol{p} = \boldsymbol{p}_0 t_0 + \boldsymbol{p}_1 t_1 + \boldsymbol{p}_2 t_2 + \boldsymbol{p}_3 t_3, \tag{5.3}$$

przy użyciu czwórki liczb (t_0, t_1, t_2, t_3), których suma jest równa 1. Mamy również pełną swobodę wybierania układów współrzędnych barycentrycznych w przestrzeni[5]. Jeśli zamiast punktów $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_3$ podstawimy wektory ich współrzędnych kartezjańskich (w dowolnym ustalonym układzie), to podany wyżej wzór umożliwia obliczenie współrzędnych kartezjańskich punktu $\boldsymbol{p}$ (w tymże układzie) na podstawie jego współrzędnych barycentrycznych.

Współrzędne barycentryczne można również „ujednorodnić"; w tym celu warunek $t_0 + t_1 + t_2 + t_3 = 1$ zastępujemy przez $T_0 + T_1 + T_2 + T_3 \neq 0$. „Zwykłe" współrzędne barycentryczne $t_0, \ldots, t_3$ otrzymamy z barycentrycznych współrzędnych jednorodnych, dzieląc każdą z nich przez sumę $T_0 + T_1 + T_2 + T_3$, która odpowiada wadze wektora współrzędnych. Mając jednorodne współrzędne barycentryczne T_0, T_1, T_2, T_3 punktu $\boldsymbol{p}$, możemy obliczyć jego wektor współrzędnych jednorodnych $\boldsymbol{P}$ na podstawie wzoru

$$\boldsymbol{P} = \boldsymbol{P}_0 T_0 + \boldsymbol{P}_1 T_1 + \boldsymbol{P}_2 T_2 + \boldsymbol{P}_3 T_3, \tag{5.4}$$

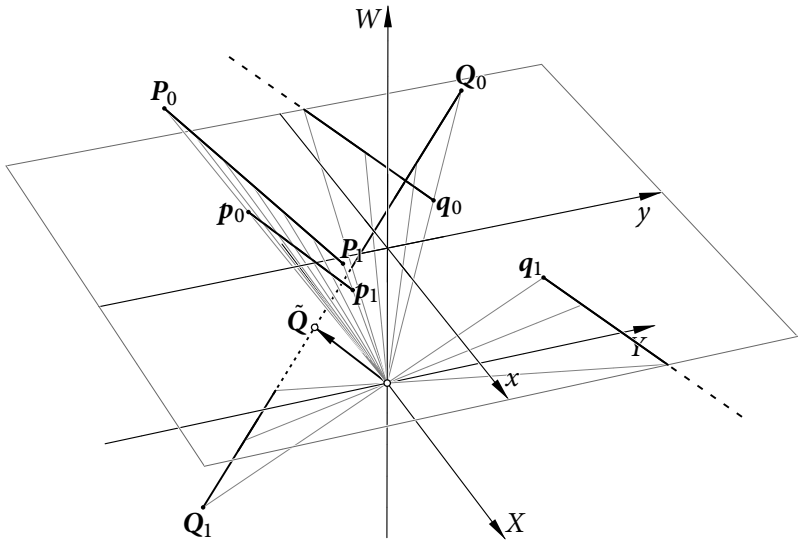
w którym występują wektory $\boldsymbol{P}_0, \ldots, \boldsymbol{P}_3$ współrzędnych jednorodnych punktów $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_3$, pod warunkiem, że współrzędne wagowe we wszystkich tych wektorach mają *tę samą wartość*[6].

Ograniczenie wektorów współrzędnych barycentrycznych punktów do takich, których suma współrzędnych jest równa 1, umożliwia dokładnie taką samą odpowiedniość między działaniami na punktach i wektorach swobodnych oraz rachunkami na wektorach współrzędnych barycentrycznych. Odejmując dwa takie wektory, otrzymamy wektor, którego

[5] Punkt $\boldsymbol{p}$ jest kombinacją afiniczną punktów $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_3$ o współczynnikach $t_0, \ldots, t_3$. Zauważmy, że w układzie współrzędnych kartezjańskich o początku $\boldsymbol{o} = \boldsymbol{p}_0$ i wersorach osi $\boldsymbol{v}_1 = \boldsymbol{p}_1 - \boldsymbol{p}_0$, $\boldsymbol{v}_2 = \boldsymbol{p}_2 - \boldsymbol{p}_0$ i $\boldsymbol{v}_3 = \boldsymbol{p}_3 - \boldsymbol{p}_0$ punkt $\boldsymbol{p}$ ma współrzędne $x = t_1$, $y = t_2$ i $z = t_3$.

[6] Zauważmy, że jeśli współrzędna wagowa wektorów $\boldsymbol{P}_0, \ldots, \boldsymbol{P}_3$ jest równa 1, to otrzymamy wektor współrzędnych jednorodnych punktu $\boldsymbol{p}$ o współrzędnej wagowej $T_0 + T_1 + T_2 + T_3$.

suma współrzędnych jest zerem — a więc wektory w $\mathbb{R}^4$ o zerowej sumie współrzędnych barycentrycznych reprezentują wektory swobodne w przestrzeni trójwymiarowej.



Rysunek 5.2. Odcinki w przestrzeni współrzędnych jednorodnych i odpowiadające im figury

W pewnych sytuacjach reprezentacje punktów w postaci wektorów współrzędnych jednorodnych o różnych wagach (lub wektorów współrzędnych barycentrycznych jednorodnych o różnych sumach) bardzo się przydają, najczęściej, gdy mamy do czynienia z krzywymi lub powierzchniami wymiernymi (np. w aplikacji przedstawionej w rozdziałach 15–26). W zasadzie znak współrzędnej wagowej może być dowolny, ale rozważmy odcinek $\boldsymbol{p}_0\boldsymbol{p}_1$. Składa się on z punktów $(1-t)\,\boldsymbol{p}_0 + t\,\boldsymbol{p}_1$ dla $t \in [0,1]$. Jeśli końce tego odcinka reprezentujemy za pomocą wektorów współrzędnych jednorodnych $\boldsymbol{P}_0$, $\boldsymbol{P}_1$, to punkty odcinka $\boldsymbol{P}_0\boldsymbol{P}_1$ reprezentują punkty odcinka $\boldsymbol{p}_0\boldsymbol{p}_1$ tylko wtedy, gdy współrzędne wagowe obu końców mają ten sam znak. Jeśli znaki wag są różne, to mamy jednorodną reprezentację prostej z usuniętym wnętrzem tego odcinka. Na przykład odcinek $\boldsymbol{Q}_0\boldsymbol{Q}_1$ na rysunku 5.2 reprezentuje dwie półproste wychodzące z punktów $\boldsymbol{q}_0$ i $\boldsymbol{q}_1$.[7] Może to być źródłem tajemniczych błędów w obliczeniach, dlatego jeśli decydujemy się na używanie współrzędnych jednorodnych z różnymi wagami, to powinniśmy przyjąć ograniczenie, że wszystkie wagi mają być dodatnie, lub zachować szczególną ostrożność[8].

[7] Zaznaczony na odcinku $\boldsymbol{Q}_0\boldsymbol{Q}_1$ wektor $\tilde{\boldsymbol{Q}}$ ma wagę równą 0. Wektor ten ma kierunek prostej przechodzącej przez punkty $\boldsymbol{q}_0$, $\boldsymbol{q}_1$.

[8] Ostrożność przyda się zwłaszcza podczas programowania szaderów geometrii, które dzielą odcinki i trójkąty na mniejsze części. Ponadto etap obcinania odcinków i trójkątów, których wierzchołki są reprezentowane przez wektory współrzędnych jednorodnych o niedodatniej wadze, może dawać błędne wyniki. Opis algorytmu obcinania trójkątów, umożliwiający m.in. zrozumienie powodu powstawania takich błędów, zamieściłem w p. 19.8.7.

5.4. Przekształcenia afiniczne

Przekształcenie afiniczne przestrzeni trójwymiarowej jest dane wzorem

$$f(\boldsymbol{p}) = L\boldsymbol{p} + \boldsymbol{t}, \tag{5.5}$$

w którym występuje macierz L o wymiarach 3×3 opisująca **część liniową** przekształcenia f i **wektor przesunięcia** $\boldsymbol{t} \in \mathbb{R}^3$; w podanym wzorze symbole $\boldsymbol{p}$ i $f(\boldsymbol{p})$ oznaczają wektory współrzędnych kartezjańskich punktu $\boldsymbol{p}$ i jego obrazu[9].

Długo by pisać o podstawowych własnościach przekształceń afinicznych, ale wychodzę z założenia, że Czytelnik je zna. Zatem, opiszę tylko ich **jednorodną reprezentację**. Jest nią macierz A o wymiarach 4×4, która zawiera bloki L i $\boldsymbol{t}$:

$$A = \left[\begin{array}{ccc|c} & L & & \boldsymbol{t} \\ \hline 0 & 0 & 0 & 1 \end{array}\right]. \tag{5.6}$$

Aby poddać punkt $\boldsymbol{p}$ przekształceniu, wystarczy obliczyć iloczyn $A\boldsymbol{P}$ macierzy A i wektora współrzędnych jednorodnych punktu $\boldsymbol{p}$. Jeśli współrzędna wagowa wektora $\boldsymbol{P}$ jest równa 1, to iloczyn — wektor współrzędnych jednorodnych punktu $f(\boldsymbol{p})$ — ma również współrzędną wagową równą 1, a więc pierwsze trzy współrzędne jednorodne są współrzędnymi kartezjańskimi tego punktu.

Przekształcenie wektora swobodnego odpowiadające przekształceniu afinicznemu f polega na zastosowaniu do niego przekształcenia liniowego l reprezentowanego przez macierz L: $l(\boldsymbol{v}) = L\boldsymbol{v}$. Iloczyn macierzy A i wektora $\boldsymbol{V} \in \mathbb{R}^4$ otrzymanego przez dopisanie współrzędnej wagowej 0 do wektora $\boldsymbol{v}$ składa się z iloczynu $L\boldsymbol{v}$ i wagi 0. Zatem reprezentacja jednorodna w postaci macierzy A umożliwia jednolite traktowanie przekształcanych punktów i wektorów swobodnych, o ile tylko przestrzegamy ograniczenia, że wagi punktów są równe 1, a ostatni wiersz macierzy A składa się z trzech zer na początku i jedynki.

Jednorodność powyższej reprezentacji oznacza, że macierz aA, dla dowolnej stałej $a \neq 0$, reprezentuje to samo przekształcenie co macierz A.[10] Jedną z zalet tej reprezentacji jest prostota wzoru opisującego złożenie przekształceń; jeśli przekształcenia afiniczne f_1 i f_2 są reprezentowane przez macierze A_1 i A_2, to złożenie $f_2 \circ f_1$ reprezentuje iloczyn A_2A_1. Iloczyn

[9] Punkt $\boldsymbol{p}$ utożsamiliśmy z trójką liczb (x, y, z) — jego współrzędnych kartezjańskich. Do wzoru (5.5) w miejscu $\boldsymbol{p}$ podstawiamy macierz kolumnową

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

i podobną macierz współrzędnych wektora $\boldsymbol{t}$. Symbole $\mathbb{R}^3$ i $\mathbb{R}^4$ oznaczają przestrzenie liniowe, których elementami są trójki i czwórki liczb; w tekście zapisujemy je w postaci (x, y, z), ale w wyrażeniach macierzowych traktujemy je jak macierze kolumnowe.

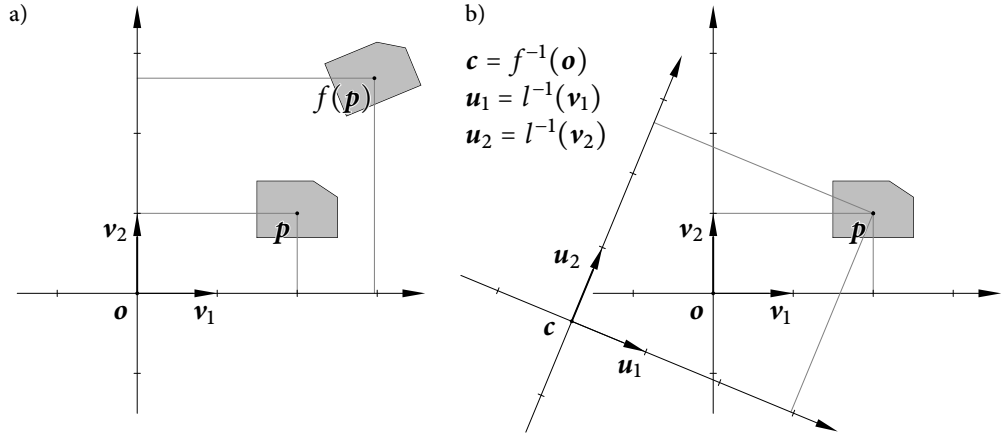
[10] Oczywiście, dopuszczając macierze z ostatnim wierszem $[0, 0, 0, a]$, w którym $a \neq 1$, rezygnujemy z otrzymywania wektorów współrzędnych jednorodnych punktów o wadze 1, wskutek czego tracimy opisane wcześniej utożsamienie odejmowania punktów z odejmowaniem ich wektorów współrzędnych jednorodnych.

ten reprezentuje również złożenie związanych z f_2 i f_1 przekształceń liniowych l_2 i l_1, którym podlegają wektory swobodne.

Macierz przekształcenia afinicznego możemy przedstawić w taki sposób:

$$A = \begin{bmatrix} \boldsymbol{w}_1 & \boldsymbol{w}_2 & \boldsymbol{w}_3 & \boldsymbol{t} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Wtedy widzimy, że jej kolumny są jednorodnymi reprezentacjami trzech wektorów swobodnych i punktu. Punkt ten jest obrazem początku układu współrzędnych w przekształceniu f, a wektory $\boldsymbol{w}_1$, $\boldsymbol{w}_2$, $\boldsymbol{w}_3$ są obrazami wersorów osi w przekształceniu l będącym częścią liniową przekształcenia f.



Rysunek 5.3. Przekształcenie afiniczne i jego dualna interpretacja

Przekształcenia afiniczne mają **dualną interpretację**, z której również będziemy intensywnie korzystać w aplikacjach OpenGL-a. W interpretacji dualnej wektor $f(\boldsymbol{p}) \in \mathbb{R}^3$ obliczony ze wzoru (5.5) reprezentuje *ten sam* punkt co wektor $\boldsymbol{p}$, ale w *nowym* układzie współrzędnych. Rozważmy dwa układy współrzędnych: pierwszy o wersorach osi $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$ i początku $\boldsymbol{o}$ oraz drugi o wersorach osi $\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3$ i początku $\boldsymbol{c}$, takie że wersory osi i początek *pierwszego* układu mają *w drugim* układzie wektory współrzędnych kartezjańskich $\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3, \boldsymbol{t} \in \mathbb{R}^3$. Macierz A, której kolumny powstały przez dołączenie trzech zer i jedynki do tych wektorów, opisuje (w postaci jednorodnej) przejście od pierwszego układu współrzędnych do drugiego; macierz przejścia od układu drugiego do pierwszego jest jej odwrotnością[11]. Zatem przekształcenie afiniczne f reprezentowane przez macierz A jest w interpretacji dualnej przejściem do układu współrzędnych, którego układ odniesienia otrzymamy, stosując przekształcenie f^{-1} i jego część liniową l^{-1} do elementów wyjściowego układu odniesienia: $\boldsymbol{c} = f^{-1}(\boldsymbol{o})$, $\boldsymbol{u}_1 = l^{-1}(\boldsymbol{v}_1)$, $\boldsymbol{u}_2 = l^{-1}(\boldsymbol{v}_2)$ i $\boldsymbol{u}_3 = l^{-1}(\boldsymbol{v}_3)$ (zobacz rysunek 5.3b; nie ma na nim wersorów $\boldsymbol{v}_3$ i $\boldsymbol{u}_3$, bo nie leżą w płaszczyźnie kartki). Temat ten wielokrotnie będzie wracał w opisie aplikacji.

[11] Przekształcenie f musi być różnowartościowe, co ma miejsce, gdy macierz $L = [\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3]$ jest nieosobliwa.

Dualna interpretacja przydaje się w różnych sytuacjach. Najprostszy przykład to konstrukcja macierzy obrotu wokół osi przechodzącej przez pewien punkt $\boldsymbol{p}$ na podstawie macierzy R obrotu wokół osi o tym samym kierunku, przechodzącej przez punkt $\boldsymbol{o}$ — początek układu współrzędnych. Jeśli T oznacza macierz przesunięcia o wektor $\boldsymbol{p}-\boldsymbol{o}$, to poszukiwana macierz jest iloczynem TRT^{-1}. Czynnik T^{-1} opisuje przejście do układu, który ma początek w punkcie $\boldsymbol{p}$ (i w którym oś obrotu ma ten sam kierunek), R reprezentuje obrót w tym układzie, a macierz T odpowiada za powrót do układu wyjściowego.

Jeśli pewien obiekt ma być poddany kolejno n przekształceniom opisanym w *tym samym* układzie współrzędnych odpowiednio przez macierze $A_1,\dots,A_n$, to wektor współrzędnych jednorodnych $\boldsymbol{P}$ każdego punktu (wierzchołka) tego obiektu mnożymy przez kolejne macierze z lewej strony, otrzymując wyrażenie $A_n\dots A_1\boldsymbol{P}$. Stąd macierz złożenia tych przekształceń jest iloczynem $A_n\dots A_1$.

Jeśli poddajemy obiekt serii przekształceń określonych w układzie współrzędnych, który *porusza się razem z obiektem*, to kolejność czynników jest odwrotna — $A_1\dots A_n$. Wystarczy to pokazać dla złożenia dwóch kolejnych przekształceń.

Niech A_1 oznacza (nieosobliwą) macierz pierwszego przekształcenia obiektu. Przekształceniu temu poddajemy też układ odniesienia (tj. początek i wersory osi) związany z obiektem układu współrzędnych, zatem przejście od układu wyjściowego do układu przekształconego razem z obiektem jest opisane przez macierz A_1^{-1}. Jeśli drugie przekształcenie jest w nowym układzie reprezentowane przez macierz A_2, to jego macierz w pierwszym układzie jest iloczynem $M = A_1A_2A_1^{-1}$ — kolejne czynniki (od prawej strony) opisują przejście od układu oryginalnego do przekształconego, drugie przekształcenie w układzie przekształconym i powrót do układu oryginalnego. Mając macierze obu przekształceń w tym samym (wyjściowym) układzie, możemy obliczyć iloczyn

$$MA_1 = A_1A_2A_1^{-1}A_1 = A_1A_2. \quad \square$$

Składanie serii przekształceń danych w układzie współrzędnych związanym z obiektem przydaje się na przykład w animacji ruchu samochodu, który porusza się do *swojego* przodu lub do tyłu — w kierunku, w którym jest chwilowo ustawiony w świecie, po którym jeździ.

Choć to rzadziej będzie potrzebne, zobaczmy, jak można znaleźć macierz przekształcenia afinicznego f, przez którą chcielibyśmy mnożyć wektor współrzędnych barycentrycznych punktu $\boldsymbol{p}$, aby otrzymać wektor współrzędnych barycentrycznych punktu $f(\boldsymbol{p})$. Ze wzoru (5.4) wynika, że macierz $B = [\boldsymbol{P}_0, \boldsymbol{P}_1, \boldsymbol{P}_2, \boldsymbol{P}_3]$, której kolumny są wektorami współrzędnych jednorodnych punktów układu odniesienia ze współrzędną wagową równą 1, opisuje przejście od współrzędnych barycentrycznych dowolnego punktu do kartezjańskich (iloczyn $\boldsymbol{P} = B\boldsymbol{T}$ macierzy B i wektora $\boldsymbol{T}$ współrzędnych barycentrycznych punktu $\boldsymbol{p}$ jest wektorem współrzędnych jednorodnych punktu $\boldsymbol{p}$ z wagą 1). Zatem poszukiwana macierz jest iloczynem $B^{-1}AB$ — reprezentuje on złożenie trzech przekształceń: przejścia do współrzędnych kartezjańskich, przekształcenia f zapisanego we współrzędnych kartezjańskich i powrotu do współrzędnych barycentrycznych[12].

[12] Macierz B jest nieosobliwa, gdy punkty $\boldsymbol{p}_0,\dots,\boldsymbol{p}_3$ nie leżą w jednej płaszczyźnie.

5.5. Prostopadłość

Pojęcie prostopadłości (np. prostych) w przestrzeni afinicznej jest związane z **iloczynem skalarnym** określonym w przestrzeni wektorów swobodnych; jest to funkcja, która parze wektorów przyporządkowuje liczbę i która *w pewnym układzie współrzędnych kartezjańskich* może być zapisana wzorem

$$\langle \boldsymbol{v}, \boldsymbol{w} \rangle = x_v x_w + y_v y_w + z_v z_w. \tag{5.7}$$

W notacji macierzowej $\langle \boldsymbol{v}, \boldsymbol{w} \rangle = \boldsymbol{v}^T \boldsymbol{w} = \boldsymbol{w}^T \boldsymbol{v}$. Przyjmiemy, że ten wzór obowiązuje w **układzie współrzędnych świata**, którego definicja jest podana w podrozdziale 6.6. Dwa wektory są wzajemnie prostopadłe, jeśli ich iloczyn skalarny jest zerem. Przestrzeń wyposażona w iloczyn skalarny to **przestrzeń euklidesowa**.

Iloczyn skalarny umożliwia określenie **długości wektora**,

$$\|\boldsymbol{v}\| = \sqrt{\langle \boldsymbol{v}, \boldsymbol{v} \rangle},$$

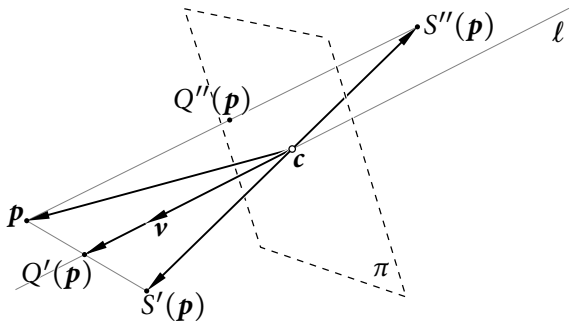
a dalej **odległości punktów**,

$$\rho(\boldsymbol{p}, \boldsymbol{q}) = \|\boldsymbol{p} - \boldsymbol{q}\|,$$

a także pojęcia kąta (nieskierowanego) φ między niezerowymi wektorami $\boldsymbol{v}$ a $\boldsymbol{w}$; kąt ten można otrzymać ze wzoru

$$\cos\varphi = \frac{\langle \boldsymbol{v}, \boldsymbol{w} \rangle}{\|\boldsymbol{v}\| \|\boldsymbol{w}\|}.$$

Wektory $\boldsymbol{v}$ i $\boldsymbol{w}$ tworzą kąt ostry, jeśli $\langle \boldsymbol{v}, \boldsymbol{w} \rangle > 0$, prosty dla $\langle \boldsymbol{v}, \boldsymbol{w} \rangle = 0$ i rozwarty, gdy $\langle \boldsymbol{v}, \boldsymbol{w} \rangle < 0$. Jeśli wektory te są **jednostkowe** (tj. o długości 1), to $\cos\varphi = \langle \boldsymbol{v}, \boldsymbol{w} \rangle$.



Rysunek 5.4. Rzuty prostopadłe i odbicia symetryczne względem prostej i płaszczyzny

Wektor jednostkowy $\boldsymbol{v}$ może być użyty do określenia **rzutów ortogonalnych**, czyli przekształceń wektorów swobodnych, opisanych wzorami

$$P'(\boldsymbol{w}) = \boldsymbol{v}\langle \boldsymbol{v}, \boldsymbol{w} \rangle = \boldsymbol{v}\boldsymbol{v}^T \boldsymbol{w}, \tag{5.8}$$

$$P''(\boldsymbol{w}) = \boldsymbol{w} - \boldsymbol{v}\langle \boldsymbol{v}, \boldsymbol{w} \rangle = (I - \boldsymbol{v}\boldsymbol{v}^T)\boldsymbol{w}. \tag{5.9}$$

Obrazy dowolnego wektora $\boldsymbol{w}$ w tych przekształceniach są do siebie prostopadłe, a ponadto jest $P'\big(P'(\boldsymbol{w})\big) = P'(\boldsymbol{w})$, $P''\big(P''(\boldsymbol{w})\big) = P''(\boldsymbol{w})$. Na ich podstawie jest określone rzutowanie

punktów prostopadle na prostą i płaszczyznę. Rozważmy dowolny punkt $\boldsymbol{c}$ i przechodzącą przez ten punkt prostą ℓ o kierunku wektora $\boldsymbol{v}$ oraz płaszczyznę π do tej prostej prostopadłą. Wartościami przekształceń

$$Q'(\boldsymbol{p}) = \boldsymbol{c} + P'(\boldsymbol{p} - \boldsymbol{c}), \tag{5.10}$$

$$Q''(\boldsymbol{p}) = \boldsymbol{c} + P''(\boldsymbol{p} - \boldsymbol{c}) \tag{5.11}$$

są rzuty prostopadłe punktu $\boldsymbol{p}$ na prostą ℓ i płaszczyznę π. Rzuty ortogonalne umożliwiają skonstruowanie **odbić symetrycznych**; wzory opisujące odbicia wektorów są następujące:

$$R'(\boldsymbol{w}) = 2P'(\boldsymbol{w}) - \boldsymbol{w} = (2\boldsymbol{w}\boldsymbol{w}^T - I)\boldsymbol{w}, \tag{5.12}$$

$$R''(\boldsymbol{w}) = \boldsymbol{w} - 2P'(\boldsymbol{w}) = -R'(\boldsymbol{w}), \tag{5.13}$$

a odbicia symetryczne punktu $\boldsymbol{p}$ względem prostej l oraz płaszczyzny π otrzymamy, podstawiając ten punkt do wzorów

$$S'(\boldsymbol{p}) = \boldsymbol{c} + R'(\boldsymbol{p} - \boldsymbol{c}), \tag{5.14}$$

$$S''(\boldsymbol{p}) = \boldsymbol{c} + R''(\boldsymbol{p} - \boldsymbol{c}). \tag{5.15}$$

Umiejętność mierzenia odległości umożliwia zdefiniowanie **izometrii** — to są takie przekształcenia f, że dla dowolnych punktów $\boldsymbol{p}$, $\boldsymbol{q}$ ma miejsce równość

$$\rho\big(f(\boldsymbol{p}), f(\boldsymbol{q})\big) = \rho(\boldsymbol{p}, \boldsymbol{q}).$$

Na przykład opisane wzorami (5.14) i (5.15) przekształcenia S' i S'' są izometriami.

Każda izometria w afinicznej przestrzeni euklidesowej jest przekształceniem afinicznym, którego część liniowa jest opisana przez **macierz ortogonalną**. Kolumny macierzy ortogonalnej są wektorami jednostkowymi, przy czym każda z nich jest prostopadła do pozostałych. Stąd wynika, że macierz L jest ortogonalna wtedy i tylko wtedy, gdy jej transpozycja jest jej odwrotnością: $L^{-1} = L^T$. Jeśli przejście od (opisanego w następnym rozdziale) układu współrzędnych świata, w którym obowiązuje wzór (5.7), do dowolnego innego jest izometrią, to w tym innym układzie iloczyn skalarny też może być obliczony ze wzoru (5.7); takie układy współrzędnych nazwiemy **układami izometrycznymi**[13].

Złożenie izometrii jest również izometrią. Wszystkie izometrie afinicznej przestrzeni trójwymiarowej są przesunięciami, obrotami wokół pewnych prostych, odbiciami symetrycznymi względem płaszczyzn lub prostych albo złożeniami wymienionych wyżej przekształceń. Co więcej, okazuje się, że każda taka izometria jest przesunięciem, odbiciem symetrycznym lub obrotem („zwykłym" lub złożonym z przesunięciem równoległym do prostej lub płaszczyzny odbicia albo osi obrotu) albo złożeniem obrotu z odbiciem symetrycznym względem płaszczyzny prostopadłej do osi obrotu.

W trójwymiarowej przestrzeni euklidesowej jest określony **iloczyn wektorowy** dwóch wektorów, dobrze znanym wzorem

$$\begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix} \wedge \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = \begin{bmatrix} y_v z_w - y_w z_v \\ z_v x_w - z_w x_v \\ x_v y_w - x_w y_v \end{bmatrix}, \tag{5.16}$$

[13] Izometrycznymi z układem świata.

w którym występują współrzędne kartezjańskie wektorów w (dowolnym) układzie izometrycznym. Iloczyn wektorowy jest wektorem prostopadłym do swoich czynników, a jego długość jest iloczynem długości tych czynników i sinusa kąta między nimi:

$$\|\boldsymbol{w} \wedge \boldsymbol{v}\| = \|\boldsymbol{w}\|\|\boldsymbol{v}\| \sin \varphi.$$

Iloczyn wektorowy *nie jest działaniem przemiennym* (bo zawsze $\boldsymbol{v} \wedge \boldsymbol{w} = -\boldsymbol{w} \wedge \boldsymbol{v}$, a nie zawsze $\boldsymbol{v} \wedge \boldsymbol{w} = \boldsymbol{0}$) i *nie jest też łączny* (bo dla dowolnych niezależnych liniowo wektorów $\boldsymbol{v}$ i $\boldsymbol{w}$ jest $(\boldsymbol{v} \wedge \boldsymbol{w}) \wedge \boldsymbol{w} \neq \boldsymbol{0} = \boldsymbol{v} \wedge (\boldsymbol{w} \wedge \boldsymbol{w})$).

Macierze 3×3, których kolejne kolumny są iloczynami wektorowymi ustalonego wektora $\boldsymbol{v} \in \mathbb{R}^3$ z kolumnami $\boldsymbol{e}_1$, $\boldsymbol{e}_2$ i $\boldsymbol{e}_3$ macierzy jednostkowej, przy czym wektor $\boldsymbol{v}$ jest odpowiednio pierwszym i drugim argumentem, oznaczymy symbolami $\boldsymbol{v} \wedge I$ oraz $I \wedge \boldsymbol{v}$. Mamy

$$\boldsymbol{v} \wedge I = \begin{bmatrix} 0 & -z_{\boldsymbol{v}} & y_{\boldsymbol{v}} \\ z_{\boldsymbol{v}} & 0 & -x_{\boldsymbol{v}} \\ -y_{\boldsymbol{v}} & x_{\boldsymbol{v}} & 0 \end{bmatrix}, \quad I \wedge \boldsymbol{v} = \begin{bmatrix} 0 & z_{\boldsymbol{v}} & -y_{\boldsymbol{v}} \\ -z_{\boldsymbol{v}} & 0 & x_{\boldsymbol{v}} \\ y_{\boldsymbol{v}} & -x_{\boldsymbol{v}} & 0 \end{bmatrix} = -\boldsymbol{v} \wedge I = (\boldsymbol{v} \wedge I)^T.$$

Dla każdego wektora $\boldsymbol{w} \in \mathbb{R}^3$ jest $(\boldsymbol{v} \wedge I)\boldsymbol{w} = \boldsymbol{v} \wedge \boldsymbol{w}$ oraz $\boldsymbol{w}^T(I \wedge \boldsymbol{v}) = (\boldsymbol{v} \wedge \boldsymbol{w})^T$.

W różnych konstrukcjach trzeba obliczać iloczyn skalarny danego wektora $\boldsymbol{u}$ z iloczynem wektorowym pewnych wektorów $\boldsymbol{v}$ i $\boldsymbol{w}$. Okazuje się, że jest to wyznacznik macierzy 3×3 utworzonej z tych wektorów:

$$\langle \boldsymbol{u}, \boldsymbol{v} \wedge \boldsymbol{w} \rangle = \det[\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}].$$

W szczególności wyznacznik macierzy $[\boldsymbol{w}, \boldsymbol{v}, \boldsymbol{w} \wedge \boldsymbol{v}]$ jest nieujemny:

$$\det[\boldsymbol{w}, \boldsymbol{v}, \boldsymbol{w} \wedge \boldsymbol{v}] = \langle \boldsymbol{w} \wedge \boldsymbol{v}, \boldsymbol{w} \wedge \boldsymbol{v} \rangle = \|\boldsymbol{w}\|^2 \|\boldsymbol{v}\|^2 \sin^2 \varphi.$$

Niech $\boldsymbol{q}$ oznacza dowolny punkt w przestrzeni, a $\boldsymbol{n}$ niezerowy wektor. Równanie

$$\langle \boldsymbol{p} - \boldsymbol{q}, \boldsymbol{n} \rangle = 0$$

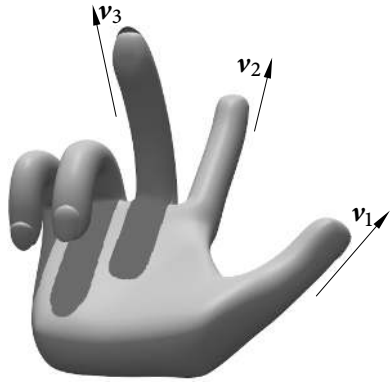
jest spełnione przez punkty $\boldsymbol{p}$ pewnej płaszczyzny π. Dla *dowolnego* punktu $\boldsymbol{p}$ liczbę

$$d_{\boldsymbol{p}} = \langle \boldsymbol{p} - \boldsymbol{q}, \boldsymbol{n} \rangle$$

nazwiemy **odległością ze znakiem** punktu $\boldsymbol{p}$ od płaszczyzny π. Odległość ta, określona w jednostkach odwrotnie proporcjonalnych do długości wektora $\boldsymbol{n}$, jest zerem, gdy $\boldsymbol{p}$ jest punktem płaszczyzny π, jest dodatnia, gdy punkt $\boldsymbol{p}$ znajduje się po jej stronie wskazywanej przez wektor $\boldsymbol{n}$, i jest ujemna, gdy leży po przeciwnej stronie.

5.6. Orientacja

Pojęcie **orientacji** pary wektorów na płaszczyźnie lub trójki wektorów w przestrzeni trójwymiarowej jest związane z uporządkowaniem tych wektorów. Rozważmy trójki wektorów



Rysunek 5.5. Określenie prawoskrętnych układów współrzędnych

liniowo niezależnych w przestrzeni trójwymiarowej, $(\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3)$ i $(\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3)$. Istnieje nieosobliwa macierz L, taka że $[\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3] = L[\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3]$. Orientacje obu trójek są **zgodne**, jeśli $\det L > 0$, i **przeciwne**, jeśli $\det L < 0$. W szczególności, jeśli jedna trójka powstała z drugiej przez przestawienie dowolnych dwóch wektorów, to $\det L = -1$ i orientacje trójek są przeciwne.

Pojęcie orientacji dotyczy m.in. trójek wersorów osi układów współrzędnych kartezjańskich. Jest kwestią *umowy* nazwanie pewnego wyróżnionego układu współrzędnych (np. układu świata) **układem prawoskrętnym** — i wtedy wszystkie układy, których wersory osi są zorientowane zgodnie z trójką wersorów tego układu, są prawoskrętne, a wszystkie zorientowanie przeciwnie są **lewoskrętne**. Przyjęta dosyć powszechnie umowa jest zilustrowana na rysunku 5.5. Palce prawej dłoni: kciuk, wskazujący i środkowy, wyznaczają (w tej kolejności) taką trójkę wektorów $(\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3)$, że każdy układ współrzędnych kartezjańskich, którego wersory osi są zorientowane zgodnie z nią, uznajemy za układ prawoskrętny.

Macierz L opisuje część liniową l przekształcenia afinicznego f, które zależnie od znaku wyznacznika tej macierzy zachowuje albo zmienia orientację. Rozważmy trójkąt o wierzchołkach $\boldsymbol{p}_i, \boldsymbol{p}_j, \boldsymbol{p}_k$, będący częścią brzegu rysowanej bryły. Niech $\boldsymbol{w}_1 = \boldsymbol{p}_j - \boldsymbol{p}_i$, $\boldsymbol{w}_2 = \boldsymbol{p}_k - \boldsymbol{p}_i$ i niech $\boldsymbol{w}_3 = \boldsymbol{w}_1 \wedge \boldsymbol{w}_2$. Dla takich wektorów zawsze jest $\det[\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3] > 0$. Wektor $\boldsymbol{w}_3$ jest prostopadły do płaszczyzny trójkąta i jest on skierowany na zewnątrz albo do wewnątrz bryły. Jeśli bryła zostanie poddana przekształceniu f, to wektory $l(\boldsymbol{w}_1)$, i $l(\boldsymbol{w}_2)$ będą odpowiadały bokom obrazu trójkąta w przekształceniu f, a wektor $l(\boldsymbol{w}_3)$ będzie skierowany tak samo na zewnątrz albo do wewnątrz obrazu bryły (przy czym nie musi być wektorem normalnym płaszczyzny obrazu trójkąta). Ale jeśli $\det L < 0$, to trójka $\big(l(\boldsymbol{w}_1), l(\boldsymbol{w}_2), l(\boldsymbol{w}_3)\big)$ będzie miała orientację przeciwną niż trójka $\big(l(\boldsymbol{w}_1), l(\boldsymbol{w}_2), l(\boldsymbol{w}_1) \wedge l(\boldsymbol{w}_2)\big)$, a to znaczy, że wektor $l(\boldsymbol{w}_1) \wedge l(\boldsymbol{w}_2)$ wskazuje przeciwną stronę płaszczyzny obrazu trójkąta niż wektor $l(\boldsymbol{w}_3)$.

Jakie to ma znaczenie w programowaniu grafiki? Rysując bryłę (czyli trójkąty, z których składa się jej brzeg), możemy dla oszczędności czasu włączyć pomijanie trójkątów „odwróconych tyłem" do obserwatora. OpenGL, mając wierzchołki obrazu trójkąta poddanego przekształceniu zrealizowanemu przez szadery, wykona algorytm równoważny obliczeniu wekto-

rów odpowiadających bokom i ich iloczynu wektorowego, aby na jego podstawie zdecydować o dalszym przetwarzaniu lub odrzuceniu trójkąta. Jeśli nie zadbamy o właściwą orientację (tj. kolejność wierzchołków) trójkątów i podczas ustawiania przełączników OpenGL-a sterujących odrzucaniem nie uwzględnimy zmiany orientacji przez wprowadzone przekształcenia, to możemy otrzymać na obrazie tylko te trójkąty, które właśnie należało odrzucić.

5.7. Procedury

Czytelnicy zainteresowani programowaniem aplikacji OpenGL-a w języku C++ mają do dyspozycji bibliotekę GLM [18], która składa się z procedur wykonujących potrzebne w grafice działania na macierzach i wektorach. Ale nie można jej używać w programach napisanych w C, dlatego w tej książce będziemy posługiwać się procedurami opisanymi niżej.

Dla macierzy o wymiarach 4×4 i mniejszych język GLSL ma zdefiniowane typy podstawowe (o nazwach takich jak `mat4` lub `mat4x4`). Współczynniki takich macierzy są upakowane jeden za drugim, przy czym ich porządek w pamięci jest *kolumnowy*: współczynniki pierwszej kolumny poprzedzają współczynniki drugiej kolumny itd. To jest inny porządek niż najczęściej stosowany w C wierszowy porządek elementów tablic dwuwymiarowych. Dlatego macierze 4×4 przetwarzane przez procedury opisane niżej są reprezentowane jako tablice jednowymiarowe; do ułatwienia dostępu do współczynników służy makrodefinicja `IND4` (zobacz listing 5.5), której pierwszy parametr jest numerem wiersza, a drugi — kolumny (numerowanych od 0 do 3).

Listingi 5.1–5.5 przedstawiają dalsze części pliku `utilities.c`. Na pierwszym z nich są procedury konstrukcji macierzy elementarnych przekształceń afinicznych: przesunięć, skalowań i obrotów. Procedura `M4x4Identf` wpisuje do tablicy przekazanej jako parametr współczynniki macierzy jednostkowej, reprezentującej przekształcenie tożsamościowe.

Procedura `M4x4Translatef` wpisuje do tablicy przekazanej jako parametr współczynniki macierzy T przesunięcia o wektor o podanych współrzędnych; macierz ta jest określona wzorem

$$T = \left[\begin{array}{ccc|c} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ \hline 0 & 0 & 0 & 1 \end{array}\right].$$

Procedura `M4x4Scalef` wpisuje do tablicy przekazanej jako parametr współczynniki macierzy skalowania osi o podane czynniki. Górny lewy blok 3×3 tej macierzy (odpowiadający macierzy L we wzorach (5.5) i (5.6)) jest równy

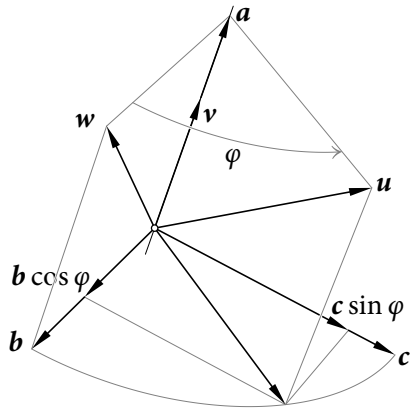
$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix},$$

a ostatnia kolumna i wiersz są takie, jak w macierzy jednostkowej.

Procedury `M4x4RotateXf`, `M4x4RotateYf`, `M4x4RotateZf` wpisują do tablicy współczynniki macierzy obrotu o kąt φ (podany w radianach), odpowiednio wokół osi x, y i z.

Macierze tych obrotów mają ostatni wiersz i kolumnę takie jak macierz jednostkowa i bloki 3×3 w górnym lewym rogu określone wzorami (w których $s = \sin\varphi$, $c = \cos\varphi$)

$$R_{x,\varphi} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix}, \qquad R_{y,\varphi} = \begin{bmatrix} c & 0 & s \\ 0 & 1 & 0 \\ -s & 0 & c \end{bmatrix}, \qquad R_{z,\varphi} = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$



Rysunek 5.6. Konstrukcja obrotu wokół osi określonej przez wektor $\boldsymbol{v}$

Obraz $\boldsymbol{u}$ wektora $\boldsymbol{w}$ w obrocie o kąt φ wokół osi o kierunku *jednostkowego* wektora $\boldsymbol{v}$ można znaleźć w sposób pokazany na rysunku 5.6. Wektory $\boldsymbol{a} = \boldsymbol{v}\langle \boldsymbol{v}, \boldsymbol{w}\rangle$ i $\boldsymbol{b} = \boldsymbol{w} - \boldsymbol{a}$ są obrazami wektora $\boldsymbol{w}$ w rzutach prostopadłych na oś obrotu i na płaszczyznę do niej prostopadłą. Wektor $\boldsymbol{c} = \boldsymbol{v} \wedge \boldsymbol{b} = \boldsymbol{v} \wedge \boldsymbol{w}$ jest prostopadły do wektorów $\boldsymbol{w}$, $\boldsymbol{a}$, $\boldsymbol{b}$ i ma długość taką jak wektor $\boldsymbol{b}$. Wektor $\boldsymbol{u}$ jest sumą $\boldsymbol{a} + \boldsymbol{b}\cos\varphi + \boldsymbol{c}\sin\varphi$, a zatem

$$\boldsymbol{u} = \boldsymbol{v}\langle \boldsymbol{v}, \boldsymbol{w}\rangle + \cos\varphi(\boldsymbol{w} - \boldsymbol{v}\langle \boldsymbol{v}, \boldsymbol{w}\rangle) + \sin\varphi \boldsymbol{v} \wedge \boldsymbol{w}. \tag{5.17}$$

Procedura `M4x4RotateVf` wpisuje do tablicy współczynniki macierzy obrotu o kąt φ wokół przechodzącej przez początek układu współrzędnych osi o kierunku wektora mającego współrzędne `x`, `y`, `z`. Wektor ten musi być niezerowy; po podzieleniu go przez jego długość powstaje wektor jednostkowy $\boldsymbol{v}$. Na podstawie wzoru (5.17) macierz części liniowej obrotu jest równa

$$R_{\boldsymbol{v},\varphi} = \boldsymbol{v}\boldsymbol{v}^T + c(I - \boldsymbol{v}\boldsymbol{v}^T) + s\boldsymbol{v} \wedge I. \tag{5.18}$$

We wzorze tym występuje wektor jednostkowy $\boldsymbol{v}$ otrzymany przez podzielenie wektora (x, y, z) przez jego długość, macierz jednostkowa I o wymiarach 3×3 oraz liczby $s = \sin\varphi$, $c = \cos\varphi$.

Uwaga: Choć przedstawione tu procedury konstruują macierze reprezentowane w pojedynczej precyzji, dla parametrów, których wartości są miarami kątów, przyjąłem typ `double`. Motywem tej decyzji są zastosowania w animacji: chwilowy kąt obrotu wirującego obiektu, obliczony na podstawie czasu i prędkości kątowej, może być zbyt duży, aby reprezentacja typu `float` była wystarczająco dokładna. Zbadajmy to. Odległość liczby zmiennopozycyj-

nej φ od najbliższych liczb tego samego typu oznaczamy symbolem $\operatorname{ulp}\varphi$ (zobacz s. 1187). Jeśli więc $\operatorname{ulp}\varphi > \pi/180 = 1^\circ$, to najmniejszy możliwy przyrost wartości *zmiennej* φ jest większy niż 1 stopień, jeśli zaś $\operatorname{ulp}\varphi > 2\pi$, to błąd reprezentacji jest większy niż miara kąta pełnego, a wtedy zmienna φ nie zawiera żadnej informacji o położeniu kątowym naszego obiektu. Jeśli t oznacza liczbę bitów mantysy, to $\operatorname{ulp}\varphi > 2^{-t-1}|\varphi|$. Dokładniejszy rachunek pokazuje, że jeśli kąt φ jest przechowywany w zmiennej pojedynczej precyzji (z $t = 23$), to warunek $\operatorname{ulp}\varphi > 1^\circ$ będzie spełniony, gdy $|\varphi|$ jest większa niż ok. $1.31 \cdot 10^5$ radianów, czyli w przybliżeniu 20860 pełnych obrotów. Jest mała szansa na zauważenie spowodowanego przez to błędu (objawiającego się szarpanym ruchem obrotowym animowanych obiektów) podczas uruchamiania i testowania aplikacji i całkiem spora szansa na objawienie się go już po jej wypuszczeniu w świat. W podwójnej precyzji mantysa ma $t = 52$ bity i błędy powyżej 1° pojawią się dopiero po wykonaniu ok. $1.12 \cdot 10^{13}$ obrotów.

Często zamiast podawać współrzędne wektora przesunięcia w osobnych parametrach, wygodniej jest przekazać je w tablicy. Konstrukcje macierzy przesunięcia z parametrem tablicowym realizują procedury `M4x4Translatefv` i `M4x4InvTranslatefv` (ta ostatnia konstruuje przesunięcie odwrotne, tj. w przeciwną stronę). Procedura `M4x4Scalefv` otrzymuje tablicę z trzema współczynnikami skalowania osi x, y, z. Podobnie macierz obrotu wokół osi, której wektor kierunkowy jest podany w tablicy, jest konstruowana przez procedurę `M4x4RotateVfv`.

Listing 5.1. Procedury konstrukcji macierzy przekształceń elementarnych

C

```
1: void M4x4Identf ( GLfloat a[16] )
2: {
3:   memset ( a, 0, 16*sizeof(GLfloat) );
4:   a[0] = a[5] = a[10] = a[15] = 1.0;
5: } /*M4x4Identf*/
6:
7: void M4x4Translatef ( GLfloat a[16], float x, float y, float z )
8: {
9:   M4x4Identf ( a );
10:  a[12] = x;  a[13] = y;  a[14] = z;
11: } /*M4x4Translatef*/
12:
13: void M4x4Translatefv ( GLfloat a[16], const float t[3] )
14: {
15:   M4x4Identf ( a );
16:   a[12] = t[0];  a[13] = t[1];  a[14] = t[2];
17: } /*M4x4Translatefv*/
18:
19: void M4x4InvTranslatefv ( GLfloat a[16], const float t[3] )
20: {
21:   M4x4Identf ( a );
22:   a[12] = -t[0];  a[13] = -t[1];  a[14] = -t[2];
23: } /*M4x4InvTranslatefv*/
```

```
24:
25: void M4x4Scalef ( GLfloat a[16], float sx, float sy, float sz )
26: {
27:   M4x4Identf ( a );
28:   a[0] = sx;  a[5] = sy;  a[10] = sz;
29: } /*M4x4Scalef*/
30:
31: void M4x4Scalefv ( GLfloat a[16], const float s[3] )
32: {
33:   M4x4Identf ( a );
34:   a[0] = s[0];  a[5] = s[1];  a[10] = s[2];
35: } /*M4x4Scalefv*/
36:
37: void M4x4RotateXf ( GLfloat a[16], double phi )
38: {
39:   M4x4Identf ( a );
40:   a[5] = a[10] = cos ( phi );  a[9] = -(a[6] = sin ( phi ));
41: } /*M4x4RotateXf*/
42:
43: void M4x4RotateYf ( GLfloat a[16], double phi )
44: {
45:   M4x4Identf ( a );
46:   a[0] = a[10] = cos ( phi );  a[2] = -(a[8] = sin ( phi ));
47: } /*M4x4RotateYf*/
48:
49: void M4x4RotateZf ( GLfloat a[16], double phi )
50: {
51:   M4x4Identf ( a );
52:   a[0] = a[5] = cos ( phi );  a[4] = -(a[1] = sin ( phi ));
53: } /*M4x4RotateZf*/
54:
55: void M4x4RotateVf ( GLfloat a[16], float x, float y, float z, double phi )
56: {
57:   float l, s, c, c1;
58:
59:   M4x4Identf ( a );
60:   l = 1.0/sqrt ( x*x + y*y + z*z );  x *= l;  y *= l;  z *= l;
61:   s = sin ( phi );  c = cos ( phi );  c1 = 1.0-c;
62:   a[0] = x*x*c1 + c;  a[1] = a[4] = x*y*c1;  a[5] = y*y*c1 + c;
63:   a[2] = a[8] = x*z*c1;  a[6] = a[9] = y*z*c1;  a[10] = z*z*c1 + c;
64:   a[6] += s*x;  a[2] -= s*y;  a[1] += s*z;
65:   a[9] -= s*x;  a[8] += s*y;  a[4] -= s*z;
66: } /*M4x4RotateVf*/
67:
68: void M4x4RotateVfv ( GLfloat a[16], const float v[3], double phi )
69: {
70:   M4x4RotateVf ( a, v[0], v[1], v[2], phi );
```

```
71: } /*M4x4RotateVfv*/
```

Nieraz trzeba konstruować macierze reprezentujące złożenia dwóch lub większej liczby przekształceń. Dla wygody i poprawy czytelności kodu aplikacji warto używać procedur przedstawionych na listingu 5.2. Mają one takie same parametry jak procedury konstruujące macierze przesunięć, skalowań i obrotów na listingu 5.1, ale pierwszy parametr każdej z nich, tj. tablica a, początkowo przechowuje współczynniki pewnej (dowolnej) macierzy A. Każda z tych procedur oblicza iloczyn AB albo BA, gdzie B oznacza macierz pewnego przekształcenia elementarnego, i zapamiętuje jego współczynniki w tablicy a. Nazwy procedur powstały przez wstawienie litery M do nazw procedur z listingu 5.1 przed albo po nazwie przekształcenia reprezentowanego przez macierz B. Jeśli litera M poprzedza nazwę tego przekształcenia, to macierz A jest pierwszym, a w przeciwnym razie drugim czynnikiem. Na przykład procedury M4x4MTranslatef i M4x4TranslateMf obliczają odpowiednio iloczyny AT oraz TA, gdzie T oznacza macierz przesunięcia określonego przez ostatnie trzy parametry.

Uwaga: Mnożenie macierzy 4×4 wymaga wykonania 64 mnożeń i 48 dodawań i w pewnych przypadkach tyle działań trzeba wykonać, ale jeśli jeden z czynników reprezentuje przesunięcie, obrót lub skalowanie, to iloczyn daje się otrzymać znacznie taniej. Tam, gdzie to było możliwe, dokonałem odpowiednich optymalizacji.

Procedury, których nazwy mają na końcu literę v, zamiast trzech osobnych parametrów opisujących wektor przesunięcia, współczynniki skalowania osi lub wektor osi obrotu mają jeden parametr tablicowy. Na listingu podałem tylko nagłówki procedur. Zainteresowanych Czytelników odsyłam do kodów źródłowych tych procedur.

Listing 5.2. Procedury konstruujące złożenia przekształceń

C

```
1: #define M4x4Copyf(a,b) memcpy ( a, b, 16*sizeof(GLfloat) )
2:
3: void M4x4MTranslatef ( GLfloat a[16], float x, float y, float z );
4: void M4x4MTranslatefv ( GLfloat a[16], const float t[3] );
5: void M4x4MInvTranslatefv ( GLfloat a[16], const float t[3] );
6: void M4x4MScalef ( GLfloat a[16], float sx, float sy, float sz );
7: void M4x4MScalefv ( GLfloat a[16], const float s[3] );
8: void M4x4MRotateXf ( GLfloat a[16], double phi );
9: void M4x4MRotateYf ( GLfloat a[16], double phi );
10: void M4x4MRotateZf ( GLfloat a[16], double phi );
11: void M4x4MRotateVf ( GLfloat a[16], float x, float y, float z, double phi );
12: void M4x4MRotateVfv ( GLfloat a[16], const float v[3], double phi );
13:
14: void M4x4TranslateMf ( GLfloat a[16], float x, float y, float z );
15: void M4x4TranslateMfv ( GLfloat a[16], const float t[3] );
16: void M4x4InvTranslateMfv ( GLfloat a[16], const float t[3] );
17: void M4x4ScaleMf ( GLfloat a[16], float sx, float sy, float sz );
18: void M4x4ScaleMfv ( GLfloat a[16], const float s[3] );
19: void M4x4RotateXMf ( GLfloat a[16], double phi );
```

```
20: void M4x4RotateYMf ( GLfloat a[16], double phi );
21: void M4x4RotateZMf ( GLfloat a[16], double phi );
22: void M4x4RotateVMf ( GLfloat a[16], float x, float y, float z, double phi );
23: void M4x4RotateVMfv ( GLfloat a[16], const float v[3], double phi );
```

Na listingu 5.3 są pokazane jeszcze dwie procedury, które służą do konstruowania macierzy obrotów wokół prostych przechodzących przez podany punkt (niekoniecznie przez początek układu współrzędnych $\boldsymbol{o}$). Parametry procedury `M4x4RotatePVf` to tablica a na współczynniki macierzy obrotu oraz współrzędne punktu $\boldsymbol{p}$ i wektora $\boldsymbol{v}$ oraz kąt φ. Obliczana przez tę procedurę macierz jest równa $A = TR_{\boldsymbol{v},\varphi}T^{-1}$, przy czym macierz T opisuje przesunięcie o wektor $\boldsymbol{p} - \boldsymbol{o}$, a $R_{\boldsymbol{v},\varphi}$ jest macierzą obrotu o kąt φ wokół przechodzącej przez punkt $\boldsymbol{o}$ osi o kierunku wektora $\boldsymbol{v}$. Macierz A jest zatem macierzą obrotu o kąt φ wokół osi o kietunku wektora $\boldsymbol{v}$ przechodzącej przez punkt $\boldsymbol{p}$.

Listing 5.3. Procedury `M4x4RotatePVf` i `M4x4RotateP2Vf`

C

```
1: void M4x4RotatePVf ( GLfloat a[16],
2:                      const float p[3], const float v[3], double phi )
3: {
4:   GLfloat b[16], c[16];
5:
6:   M4x4RotateVf ( b, v[0], v[1], v[2], phi );
7:   M4x4Translatef ( c, -p[0], -p[1], -p[2] );
8:   M4x4Multf ( a, b, c );
9:   a[12] += p[0];  a[13] += p[1];  a[14] += p[2];
10: } /*M4x4RotatePVf*/
11:
12: char M4x4RotateP2Vf ( GLfloat a[16], const float p[3],
13:                       const float v1[3], const float v2[3] )
14: {
15:   float   v[3], sa, ca, ang;
16:   GLfloat b[16], c[16];
17:
18:   V3CrossProductf ( v, v1, v2 );
19:   sa = sqrt ( V3DotProductf ( v, v ) );
20:   if ( sa > 0.0 ) {
21:     ca = V3DotProductf ( v1, v2 );
22:     ang = atan2 ( sa, ca );
23:     M4x4RotateVf ( a, v[0], v[1], v[2], ang );
24:     if ( p ) {
25:       M4x4Translatef ( b, -p[0], -p[1], -p[2] );
26:       M4x4Multf ( c, a, b );
27:       M4x4Translatef ( b, p[0], p[1], p[2] );
28:       M4x4Multf ( a, b, c );
29:     }
30:     return true;
```

```
31:   }
32:   else
33:     return false;
34: } /*M4x4RotateP2Vf*/
```

Zadaniem procedury `M4x4RotateP2Vf` jest skonstruowanie macierzy obrotu wokół osi przechodzącej przez punkt $\boldsymbol{p}$, przy czym kierunek i kąt tego obrotu ma być tak dobrany, aby obraz wektora $\boldsymbol{v}_1$ miał kierunek i zwrot wektora $\boldsymbol{v}_2$. Oś obrotu ma być prostopadła do wektorów $\boldsymbol{v}_1$ i $\boldsymbol{v}_2$ i, aby była dobrze określona, wektory muszą mieć różne kierunki. Wtedy oś obrotu ma kierunek wektora $\boldsymbol{v} = \boldsymbol{v}_1 \wedge \boldsymbol{v}_2$, a kąt obrotu jest kątem między wektorami $\boldsymbol{v}_1$ a $\boldsymbol{v}_2$. Można podać parametr p (wskaźnik tablicy ze współrzędnymi punktu $\boldsymbol{p}$) pusty (`NULL`) i wtedy oś obrotu ma przechodzić przez początek układu współrzędnych. Jeśli wektory $\boldsymbol{v}_1$ i $\boldsymbol{v}_2$ są liniowo zależne, to procedura informuje o niepowodzeniu konstrukcji, przekazując wartość `false`.

Procedura `M4x4MultMVf` na listingu 5.4 oblicza iloczyn macierzy A i wektora $\boldsymbol{v} \in \mathbb{R}^4$. Procedura `M4x4MultMTVf` oblicza iloczyn $A^T\boldsymbol{v}$. Aby można było prościej programować pewne konstrukcje, wprowadziłem procedury `M4x4MultMV3f`, `M4x4MultMTV3f` i `M4x4MultMP3f`. Ostatni parametr tych procedur jest wektorem $\boldsymbol{v} \in \mathbb{R}^3$. Wynik obliczeń pierwszej procedury jest iloczynem tego wektora i górnego lewego bloku 3×3 macierzy A podanej jako drugi parametr; jeśli macierz ta ma ostatni wiersz $[0, 0, 0, 1]$, to wynik ten jest obrazem wektora $\boldsymbol{v}$ w przekształceniu opisującym część liniową przekształcenia afinicznego reprezentowanego przez macierz A.

Procedura `M4x4MultMTV3f` oblicza iloczyn transpozycji górnego lewego bloku 3×3 macierzy A i danego wektora $\boldsymbol{v}$.

Procedura `M4x4MultMP3f` oblicza pierwsze trzy współrzędne iloczynu $A\boldsymbol{P}$ macierzy A o wymiarach 4×4 i wektora $\boldsymbol{P} \in \mathbb{R}^4$, którego pierwsze trzy współrzędne są dane w tablicy podanej jako trzeci parametr, a ostatnia współrzędna, przez domniemanie, jest równa 1. Otrzymujemy w ten sposób współrzędne kartezjańskie obrazu punktu w przekształceniu afinicznym reprezentowanym przez macierz A (pod warunkiem, że jej ostatni wiersz … itd.).

Listing 5.4. Procedury działań na macierzach i wektorach

C

```
1: void M4x4MultMVf ( GLfloat av[4], const GLfloat a[16], const GLfloat v[4] )
2: {
3:   int i, j;
4:
5:   for ( i = 0; i < 4; i++ ) {
6:     av[i] = a[IND4(i,0)]*v[0];
7:     for ( j = 1; j < 4; j++ ) av[i] += a[IND4(i,j)]*v[j];
8:   }
9: } /*M4x4MultMVf*/
10:
11: void M4x4MultMTVf ( GLfloat av[4], const GLfloat a[16], const GLfloat v[4] )
12: {
```

```
13:   int i, j;
14:
15:   for ( i = 0; i < 4; i++ ) {
16:     av[i] = a[IND4(0,i)]*v[0];
17:     for ( j = 1; j < 4; j++ ) av[i] += a[IND4(j,i)]*v[j];
18:   }
19: } /*M4x4MultMTVf*/
20:
21: void M4x4MultMV3f ( GLfloat av[3], const GLfloat a[16], const GLfloat v[3] )
22: {
23:   int i, j;
24:
25:   for ( i = 0; i < 3; i++ ) {
26:     av[i] = a[IND4(i,0)]*v[0];
27:     for ( j = 1; j < 3; j++ ) av[i] += a[IND4(i,j)]*v[j];
28:   }
29: } /*M4x4MultMV3f*/
30:
31: void M4x4MultMTV3f ( GLfloat atv[3], const GLfloat a[16],
32:                      const GLfloat v[3] )
33: {
34:   int i, j;
35:
36:   for ( i = 0; i < 3; i++ ) {
37:     atv[i] = a[IND4(0,i)]*v[0];
38:     for ( j = 1; j < 3; j++ ) atv[i] += a[IND4(j,i)]*v[j];
39:   }
40: } /*M4x4MultMTV3f*/
41:
42: void M4x4MultMP3f ( GLfloat ap[3], const GLfloat a[16], const GLfloat p[3] )
43: {
44:   int i, j;
45:
46:   for ( i = 0; i < 3; i++ ) {
47:     ap[i] = a[IND4(i,3)];
48:     for ( j = 0; j < 3; j++ ) ap[i] += a[IND4(i,j)]*p[j];
49:   }
50: } /*M4x4MultMP3f*/
51:
52: GLfloat V3DotProductf ( const GLfloat v1[3], const GLfloat v2[3] )
53: {
54:   return v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2];
55: } /*V3DotProductf*/
56:
57: void V3CrossProductf ( GLfloat v1xv2[3],
58:                        const GLfloat v1[3], const GLfloat v2[3] )
59: {
```

```
60:   v1xv2[0] = v1[1]*v2[2] - v1[2]*v2[1];
61:   v1xv2[1] = v1[2]*v2[0] - v1[0]*v2[2];
62:   v1xv2[2] = v1[0]*v2[1] - v1[1]*v2[0];
63: } /*V3CrossProductf*/
64:
65: float V3Normalisef ( GLfloat v[3] )
66: {
67:   float s;
68:
69:   s = sqrt ( V3DotProductf ( v, v ) );
70:   if ( s > 0.0 )
71:     { v[0] /= s, v[1] /= s, v[2] /= s; }
72:   return s;
73: } /*V3Normalisef*/
74:
75: void V3ReflectPointf ( GLfloat r[3], const GLfloat c[3],
76:                        const GLfloat nv[3], const GLfloat p[3] )
77: {
78:   GLfloat g;
79:
80:   r[0] = p[0] - c[0];     r[1] = p[1] - c[1];     r[2] = p[2] - c[2];
81:   g = 2.0*V3DotProductf ( nv, r )/V3DotProductf ( nv, nv );
82:   r[0] = p[0] - g*nv[0];  r[1] = p[1] - g*nv[1];  r[2] = p[2] - g*nv[2];
83: } /*V3ReflectPointf*/
```

Procedury `V3DotProductf` i `V3CrossProductf` obliczają odpowiednio iloczyn skalarny $\langle \boldsymbol{v}_1, \boldsymbol{v}_2 \rangle$ i iloczyn wektorowy $\boldsymbol{v}_1 \wedge \boldsymbol{v}_2$ danych wektorów $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \mathbb{R}^3$.

Procedura `V3Normalisef` dokonuje **normalizacji** wektora, tj. dzieli dany wektor przez jego długość, aby otrzymać wektor jednostkowy o tym samym kierunku i zwrocie. Długość wektora danego jest przekazywana jako wartość powrotna procedury.

Procedura `V3ReflectPointf` dokonuje odbicia punktu $\boldsymbol{p}$ względem płaszczyzny danej za pomocą punktu $\boldsymbol{c}$ i wektora normalnego $\boldsymbol{n}$; współrzędne otrzymanego punktu są wpisywane do tablicy `r`. Procedura realizuje przekształcenie S'' określone wzorem (5.15).

Procedury na listingu 5.5 wykonują podstawowe działania algebraiczne na macierzach 4×4. Procedura `M4x4Multf` oblicza iloczyn dwóch macierzy, A i B, i umieszcza go w tablicy `ab`. Macierz ta reprezentuje złożenie przekształceń realizowanych przez macierze B i A. Procedura ta jest wywoływana przez niektóre procedury z listingu 5.2.

Procedura `M4x4LUDecompf` wyznacza (metodą eliminacji Gaussa z częściowym wyborem elementu głównego) rozkład danej macierzy A na czynniki trójkątne, dolny L i górny U, spełniające równość $LU = PA$ dla pewnej macierzy permutacji P:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}.$$

Współczynniki macierzy L i U są wpisywane do tablicy `lu` zgodnie z następującym schematem:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ l_{21} & u_{22} & u_{23} & u_{24} \\ l_{31} & l_{32} & u_{33} & u_{34} \\ l_{41} & l_{42} & l_{43} & u_{44} \end{bmatrix},$$

a w pomocniczej tablicy `p` jest zapisywana reprezentacja permutacji P — składa się ona z trzech liczb całkowitych, będących numerami wierszy, z którymi w trakcie obliczeń zostały przestawione wiersze pierwszy, drugi i trzeci rozkładanej macierzy. Czynniki tego rozkładu mogą być użyte do rozwiązywania układów równań liniowych $A\boldsymbol{x} = \boldsymbol{b}$ i do znalezienia macierzy odwrotnej do A.[14] Procedura przekazuje wartość `true` (czyli liczbę 1), jeśli czynniki rozkładu udało się znaleźć, co jest możliwe, jeśli macierz A jest nieosobliwa, albo `false` (liczbę 0) w przeciwnym razie.

Procedura `M4x4LUSolvef` oblicza rozwiązanie układu równań liniowych $A\boldsymbol{x} = \boldsymbol{v}$, czyli wektor $\boldsymbol{x} = A^{-1}\boldsymbol{v} = U^{-1}L^{-1}P\boldsymbol{v}$, korzystając z czynników L, U, P rozkładu macierzy nieosobliwej A znalezionych przez procedurę `M4x4LUDecompf`.

Listing 5.5. Procedury podstawowych działań na macierzach

C

```
#define IND4(i,j) ((i)+4*(j))

void M4x4Multf ( GLfloat ab[16], const GLfloat a[16], const GLfloat b[16] )
{
  int i, j, k;

  for ( i = 0; i < 4; i++ )
    for ( j = 0; j < 4; j++ ) {
      ab[IND4(i,j)] = a[IND4(i,0)]*b[IND4(0,j)];
      for ( k = 1; k < 4; k++ )
        ab[IND4(i,j)] += a[IND4(i,k)]*b[IND4(k,j)];
    }
} /*M4x4Multf*/

char M4x4LUDecompf ( GLfloat lu[16], int p[3], const GLfloat a[16] )
{
  int     i, j, k;
  GLfloat d;

  M4x4Copyf ( lu, a );
  memset ( p, 0, 3*sizeof(int) );
```

[14] Każdy wie ze szkoły, że do rozwiązywania układów równań liniowych służą wyznaczniki. Nic bardziej mylnego. Właśnie wyznaczników (wzorów Cramera) *nie należy* stosować do numerycznego rozwiązywania układów równań liniowych, bo jest to sposób kosztowny, a skutkiem błędów zaokrągleń może być mała dokładność rozwiązania. Eliminacja Gaussa działa szybciej i daje dokładniejsze wyniki.

```
22:   for ( j = 0; j < 3; j++ ) {
23:     d = fabs ( lu[IND4(j,j)] );  p[j] = j;
24:     for ( i = j+1; i < 4; i++ )
25:       if ( fabs ( lu[IND4(i,j)] ) > d )
26:         { d = fabs ( lu[IND4(i,j)] );  p[j] = i; }
27:     if ( d == 0.0 )
28:       return false;
29:     if ( p[j] != j ) {
30:       i = p[j];
31:       for ( k = 0; k < 4; k++ ) {
32:         d = lu[IND4(i,k)]; lu[IND4(i,k)] = lu[IND4(j,k)]; lu[IND4(j,k)] = d;
33:       }
34:     }
35:     for ( i = j+1; i < 4; i++ ) {
36:       d = lu[IND4(i,j)] /= lu[IND4(j,j)];
37:       for ( k = j+1; k < 4; k++ )
38:         lu[IND4(i,k)] -= d*lu[IND4(j,k)];
39:     }
40:   }
41:   return lu[15] != 0.0;
42: } /*M4x4LUDecompf*/
43:
44: void M4x4LUSolvef ( GLfloat aiv[4], const GLfloat lu[16],
45:                     const int p[3], const GLfloat v[4] )
46: {
47:   int     i, j;
48:   GLfloat d;
49:
50:   memcpy ( aiv, v, 4*sizeof(GLfloat) );
51:   for ( i = 0; i < 3; i++ )
52:     if ( p[i] != i ) { d = aiv[i];  aiv[i] = aiv[p[i]];  aiv[p[i]] = d; }
53:   for ( i = 1; i < 4; i++ )
54:     for ( j = 0; j < i; j++ ) aiv[i] -= lu[IND4(i,j)]*aiv[j];
55:   for ( i = 3; i >= 0; i-- ) {
56:     for ( j = i+1; j < 4; j++ ) aiv[i] -= lu[IND4(i,j)]*aiv[j];
57:     aiv[i] /= lu[IND4(i,i)];
58:   }
59: } /*M4x4LUSolvef*/
60:
61: char M4x4Invertf ( GLfloat ai[16], const GLfloat a[16] )
62: {
63:   int i, p[3];
64:   GLfloat lu[16], e[4];
65:
66:   if ( !M4x4LUDecompf ( lu, p, a ) )
67:     return false;
68:   for ( i = 0; i < 4; i++ ) {
```

```
69:     memset ( e, 0, 4*sizeof(GLfloat) );  e[i] = 1.0;
70:     M4x4LUSolvef ( &ai[4*i], lu, p, e );
71:   }
72:   return true;
73: } /*M4x4Invertf*/
74:
75: void M4x4UTLTSolvef ( GLfloat ativ[4], const GLfloat lu[16],
76:                       const int p[3], const GLfloat v[4] )
77: {
78:   int     i, j;
79:   GLfloat d;
80:
81:   memcpy ( ativ, v, 4*sizeof(GLfloat) );
82:   for ( i = 0; i <= 3; i++ ) {
83:     for ( j = 0; j < i; j++ ) ativ[i] -= lu[IND4(j,i)]*ativ[j];
84:     ativ[i] /= lu[IND4(i,i)];
85:   }
86:   for ( i = 2; i >= 0; i-- )
87:     for ( j = i+1; j < 4; j++ ) ativ[i] -= lu[IND4(j,i)]*ativ[j];
88:   for ( i = 2; i >= 0; i-- )
89:     if ( p[i] != i ) { d = ativ[i]; ativ[i] = ativ[p[i]]; ativ[p[i]] = d; }
90:  }
91: } /*M4x4UTLTSolvef*/
92:
93: char M4x4TInvertf ( GLfloat ati[16], const GLfloat a[16] )
94: {
95:   int     i, p[3];
96:   GLfloat lu[16], e[4];
97:
98:   if ( !M4x4LUDecompf ( lu, p, a ) )
99:     return false;
100:  for ( i = 0; i < 4; i++ ) {
101:    memset ( e, 0, 4*sizeof(GLfloat) );  e[i] = 1.0;
102:    M4x4UTLTSolvef ( &ati[4*i], lu, p, e );
103:  }
104:  return true;
105: } /*M4x4TInvertf*/
106:
107: void M4x4Transposef ( GLfloat at[16], const GLfloat a[16] )
108: {
109:   int i, j;
110:
111:   for ( i = 0; i < 4; i++ )
112:     for ( j = 0; j < 4; j++ ) at[IND4(i,j)] = a[IND4(j,i)];
113: } /*M4x4Transposef*/
114:
115: float M4x4LUDetf ( const GLfloat lu[16], const int p[3] )
```

```
116: {
117:   float det;
118:   int   i;
119:
120:   det = lu[0]*lu[5]*lu[10]*lu[15];
121:   for ( i = 0; i < 3; i++ )
122:     if ( p[i] != i ) det = -det;
123:   return det;
124: } /*M4x4LUDetf*/
```

Procedura `M4x4Invertf` znajduje odwrotność macierzy nieosobliwej A i przekazuje wartość `true`; jeśli macierz A jest osobliwa, to procedura przekazuje wartość `false`.

Uwaga: Jeśli macierz A jest ortogonalna (np. jest to macierz obrotu lub odbicia symetrycznego), to dla zasady nie należy rozwiązywać tego układu ani znajdować odwrotności przy użyciu powyższych procedur. Znacznie lepszy (szybszy i dokładniejszy) sposób opiera się na fakcie, że odwrotność macierzy ortogonalnej jest jej transpozycją — można zatem użyć opisanej wcześniej procedury `M4x4MultMTVf` albo `M4x4Transposef`.

Procedury `M4x4UTLTSolvef` i `M4x4TInvertf` służą do rozwiązywania układu równań $A^T\boldsymbol{x} = \boldsymbol{v}$ i do znajdowania odwrotności transpozycji macierzy nieosobliwej A; parametry `lu` i `p` pierwszej z nich opisują czynniki rozkładu L, U i P macierzy A znalezione wcześniej przez procedurę `M4x4LUDecompf`. Odwrotność transpozycji macierzy A oznaczamy symbolem A^{-T}; macierz taka często jest potrzebna w obliczeniach oświetlenia.

Procedura `M4x4Transposef` znajduje macierz transponowaną do danej macierzy A.

Wyznaczniki są niezwykle użyteczne, ale w obliczeniach numerycznych, jeśli to możliwe (czyli prawie zawsze), należy ich unikać. Dlatego funkcja `M4x4LUDetf`, obliczająca wyznacznik macierzy 4×4 na podstawie czynników jej rozkładu znalezionych przez procedurę `M4x4LUDecompf`, nie będzie potrzebna w żadnej opisanej tu aplikacji. Napisałem ją po to, aby Czytelnik, jeśli *musi* obliczyć wyznacznik, nie używał wzoru (5.1), który w teorii jest świetny.

Używana przez opisane tu procedury arytmetyka zmiennopozycyjna pojedynczej precyzji jest w wielu typowych zastosowaniach w grafice wystarczająca, ale może być nie dość dokładna na przykład w zastosowaniach CAD. W razie potrzeby należy skorzystać z procedur działających w podwójnej precyzji; można takie procedury otrzymać, zmieniając typ `GLfloat` na `GLdouble` w procedurach opisanych wyżej i modyfikując ich nazwy: przez analogię do konwencji przyjętych w nazewnictwie procedur OpenGL-a końcówkę `f` wypada zmienić na `d`.[15]

[15] W „poważnych" aplikacjach warto też rozważyć użycie profesjonalnej biblioteki procedur numerycznej algebry liniowej, takiej jak LAPACK.

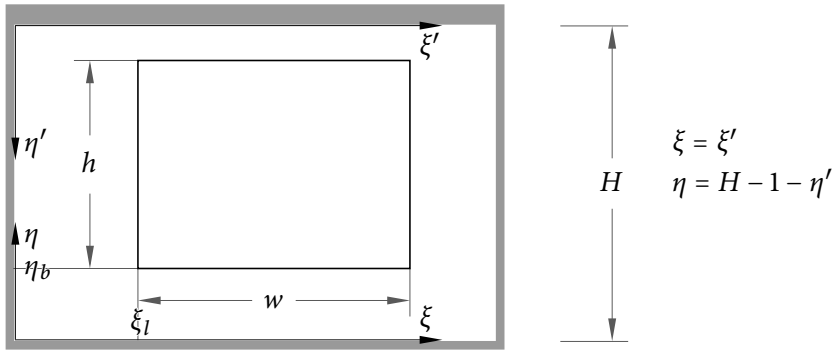
6

Rzutowanie

Dwa rodzaje rzutowania przestrzeni trójwymiarowej na płaszczyznę mają największe znaczenie w grafice komputerowej: rzutowanie równoległe i perspektywiczne. W obu przypadkach obrazem prostej w przestrzeni jest prosta lub punkt na płaszczyźnie i oba rodzaje rzutowania mogą być dokonane za pomocą liniowych przekształceń wektorów współrzędnych jednorodnych. W tym rozdziale są opisane konstrukcje macierzy takich przekształceń i procedury realizujące te konstrukcje.

6.1. Klatka, aspekt ekranu i kostka standardowa

Klatka (*viewport*) jest to prostokątny obszar w oknie, w którym ma powstać obraz. Klatka jest określona przez podanie czterech liczb całkowitych (typu `GLint`): ξ_l, η_b, w, h. Liczby ξ_l, η_b są współrzędnymi dolnego lewego piksela klatki[1], w i h to jej szerokość i wysokość. Jednostki osi to odpowiednio szerokość i wysokość jednego piksela.



Rysunek 6.1. Klatka w oknie

[1]Używam tu greckich liter ξ, η, aby odróżnić współrzędne na ekranie od współrzędnych x, y, z w układach w przestrzeni trójwymiarowej, na przykład w układzie kostki standardowej.

Domyślnie OpenGL posługuje się układem współrzędnych, którego początek znajduje się w *dolnym lewym* narożniku okna, oś ξ jest zorientowana w prawo, a oś η do góry. Systemy okien zazwyczaj używają układu ξ', η' o początku w górnym narożniku okna i z osią η' zorientowaną przeciwnie do osi η (zobacz rys. 6.1). W podanym tu opisie układu współrzędnych OpenGL-a jest pewna nieścisłość, którą wyjaśnię w podrozdziale 27.1.

Często (choć nie zawsze) chcemy, aby klatka była całym oknem (albo całym podoknem np. FreeGLUT-a). Aby ustawić wielkość i położenie w oknie klatki, w której chcemy rysować, wykonujemy instrukcję

```
glViewport ( xi_l, eta_b, w, h );
```

albo

```
glViewport ( xiprime_l, H-h-etaprime_t, w, h );
```

zależnie od tego, czy chcemy podać współrzędne ξ_l, η_b dolnego lewego narożnika klatki w układzie współrzędnych OpenGL-a, czy współrzędne ξ'_l, η'_t górnego lewego narożnika w układzie okna. Jeśli klatka ma zajmować całe okno o szerokości W i wysokości H, to podajemy parametry `( 0, 0, W, H )`.

Ważnym parametrem każdego urządzenia rastrowego jest **aspekt**: określa on kształt piksela, którego wysokość nie musi być równa szerokości. Większość ekranów obecnie produkowanych monitorów ma współczynnik aspektu a, będący ilorazem szerokości i wysokości piksela, równy 1 (co oznacza, że piksele są kwadratowe), lub na tyle bliski 1, że można się nie przejmować błędem i przyjąć $a = 1$.[2] Jeśli klatka ma wysokość h pikseli i szerokość w pikseli, to jej fizyczne wymiary na ekranie pozostają w proporcji $aw : h$.

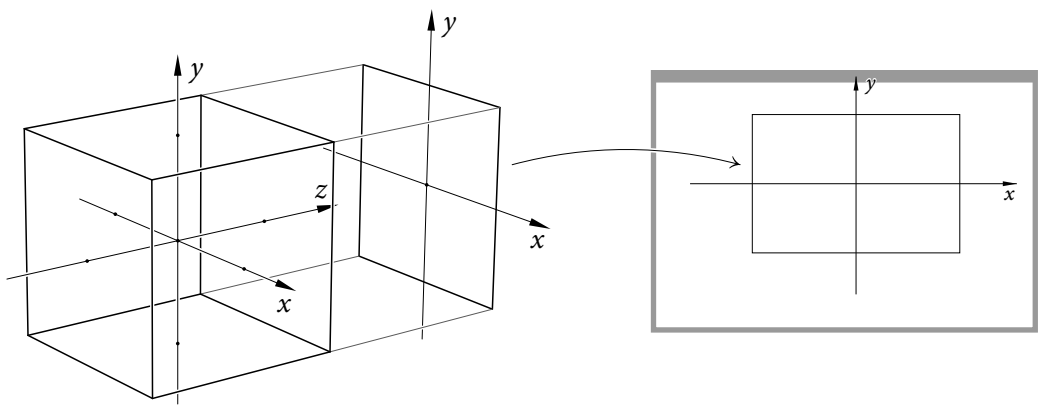
Kostka standardowa jest to sześcian o boku o długości 2; składa się z punktów, których wszystkie trzy współrzędne kartezjańskie mają wartości między −1 a 1. Zgodnie z opisem w poprzednim rozdziale współrzędne kartezjańskie punktu w układzie, w którym jest określona kostka, otrzymuje się ze współrzędnych jednorodnych X, Y, Z, W przez podzielenie pierwszych trzech przez ostatnią (wzór 5.2), przy czym obliczenia podczas obcinania prymitywów i rasteryzacja wykonywane są przy użyciu współrzędnych jednorodnych.

Właściwe rzutowanie punktów polega na odrzuceniu współrzędnej z (ale jest ona wykorzystywana w testach widoczności) oraz przeskalowaniu i przesunięciu współrzędnych x i y tak, aby ich przedziały zmienności ($[-1,1]$ dla obu współrzędnych) przeszły odpowiednio na przedziały $[\xi_l, \xi_l + w]$ i $[\eta_b, \eta_b + h]$ (zobacz rys. 6.2). Jeśli więc pewien punkt ma w układzie kostki standardowej współrzędne jednorodne (X, Y, Z, W), to jego obraz w oknie ma współrzędne

$$\xi = \xi_l + w(X/W + 1)/2, \quad \eta = \eta_b + h(Y/W + 1)/2. \tag{6.1}$$

Bryła widzenia, czyli obszar w przestrzeni, w którym zawarte są obiekty do narysowania na obrazie, musi być przekształcona (przez jeden z szaderów części przedniej potoku

[2] Według informacji podanych przez system okien (zobacz s. 83), jeden z używanych przeze mnie monitorów ma rozdzielczość 162 piksele na cal w poziomie i 161 pikseli na cal w pionie, a dwa inne odpowiednio 118 i 93 piksele na cal w poziomie oraz 117 i 95 w pionie. Stąd aspekt pierwszego monitora $a = \frac{161}{162} = 0.99382\ldots$, drugiego $a = \frac{117}{118} = 0.99152\ldots$, a trzeciego $a = \frac{95}{93} = 1.0215\ldots$. We wszystkich przypadkach błąd spowodowany przez przyjęcie $a = 1$ jest naprawdę mały.



Rysunek 6.2. Kostka standardowa i jej odwzorowanie na klatkę w oknie

przetwarzania grafiki: szadera wierzchołków, rozdrabniania lub geometrii) na kostkę standardową. Zatem aby uniknąć zniekształceń w postaci niejednakowego skalowania wymiarów poziomych i pionowych na obrazie, musimy podczas określania kształtu bryły widzenia uwzględnić proporcję fizycznych wymiarów klatki, określoną przez jej wymiary w pikselach i aspekt ekranu.

6.2. Rzutowanie perspektywiczne

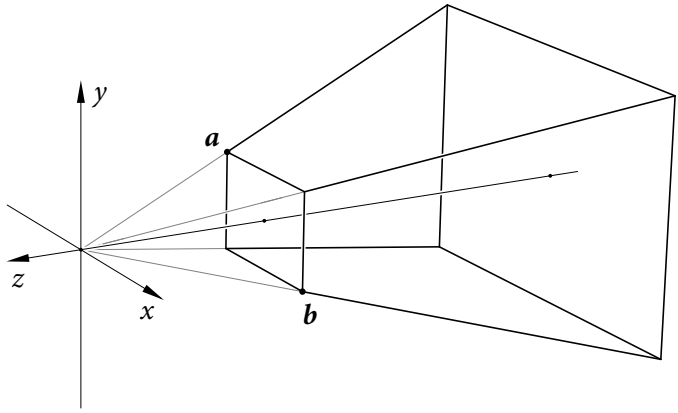
> *Spory naukowe i artystyczne* [*Platon*] *chciał rozstrzygać na drodze kodeksu karnego — w* Państwie *pisze, iż są tacy, którzy przedmioty odległe malują jako mniejsze, choć wiedzą, że są one normalnej wielkości: jako kłamcy zasługują zatem na chłostę.*
>
> Marek Kordos: *Wykłady z historii matematyki*

Bryła widzenia dla rzutowania perspektywicznego jest ściętym ostrosłupem o podstawie prostokątnej. Na rysunku 6.3 jest pokazana bryła zdefiniowana przy użyciu sześciu parametrów, które są używane jako standardowy opis w starym OpenGL-u i są na tyle wygodne, że warto tego opisu używać także w aplikacjach nowego OpenGL-a[3]. Trzeba tylko go oprogramować.

Bryła (ostrosłup) widzenia jest opisany w **układzie obserwatora**, którego początek jest środkiem rzutowania. Rzutnia jest płaszczyzną równoległą do płaszczyzny xy tego układu.

Parametry opisujące bryłę widzenia mają nazwy `left`, `right`, `bottom`, `top`, `near` i `far`; we wzorach podanych niżej są oznaczane symbolami l, r, b, t, n i f. Podstawy ostrosłupa, **przednia** i **tylna**, są położone w płaszczyznach $z = -n$ i $z = -f$; liczby n i f są dodatnie, a zatem wszystkie punkty bryły widzenia mają współrzędną z *ujemną*. Zauważmy, że dzięki temu układ xyz z osiami x i y zorientowanymi w prawo i do góry jest *prawoskrętny*.

[3] *Matematyka* używana w starym OpenGL-u nie zestarzała się nic a nic.



Rysunek 6.3. Bryła widzenia dla rzutowania perspektywicznego

Zaznaczone na rysunku wierzchołki przedniej ściany $\boldsymbol{a} = (l, t, -n)$ i $\boldsymbol{b} = (r, b, -n)$ wyjaśniają znaczenie pozostałych parametrów; ponieważ boki tej (oraz tylnej) ściany bryły widzenia są równoległe do osi x i y, a ponadto ściana tylna jest obrazem ściany przedniej w jednokładności o środku w początku układu i o skali f/n, bryła widzenia jest określona jednoznacznie.

Macierz P, która pomnożona przez wektor współrzędnych jednorodnych reprezentujący dowolny punkt z opisanego wyżej ostrosłupa widzenia da w wyniku wektor współrzędnych jednorodnych punktu z kostki standardowej[4], oraz odwrotność tej macierzy są opisane wzorami

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \qquad P^{-1} = \begin{bmatrix} \frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n} \\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{n-f}{2fn} & \frac{n+f}{2fn} \end{bmatrix}.$$

Można to wykazać, biorąc wektory współrzędnych jednorodnych wierzchołków odpowiedniej bryły, na przykład $\boldsymbol{A} = (l, t, -n, 1)$ dla punktu $\boldsymbol{a}$, i obliczając iloczyny macierzy P i tych wektorów[5]: w szczególności, $P\boldsymbol{A} = (-n, n, -n, n)$. Po podzieleniu pierwszych trzech przez czwartą współrzędną każdego wektora otrzymanego w wyniku tego mnożenia dostajemy współrzędne kartezjańskie odpowiedniego wierzchołka kostki standardowej (dla punktu $\boldsymbol{a}$ to jest wierzchołek $(-1, 1, -1)$). Odwrotność macierzy P może się przydać podczas odtwarzania położenia punktu w przestrzeni na podstawie współrzędnych odpowiedniego punktu na obrazie (w podrozdz. 27.1 jest opisany sposób i przykład zastosowania).

[4] Macierz przekształcenia, które przeprowadza bryłę widzenia na kostkę standardową, jest często nazywana **macierzą rzutowania** (*projection matrix*); jest to użyteczny skrót myślowy, ale w tym przypadku poprawniej byłoby mówić o **macierzy przekształcenia perspektywicznego**.

[5] Wektory współrzędnych zapisywane w tekście w postaci wierszowej, na przykład $(l, t, -n, 1)$, w wyrażeniach, w których są mnożone przez macierze, są macierzami kolumnowymi.

Listing 6.1. Procedura `M4x4Frustumf`

C

```
void M4x4Frustumf ( GLfloat a[16], GLfloat ai[16],
                    float left, float right, float bottom,
                    float top, float near, float far )
{
  float rl, tb, nf, nn;

  rl = right-left;  tb = top-bottom;
  nf = near-far;    nn = near+near;
  if ( a ) {
    memset ( a, 0, 16*sizeof(GLfloat) );
    a[0] = nn/rl;           a[8] = (right+left)/rl;
    a[5] = nn/tb;           a[9] = (top+bottom)/tb;
    a[10] = (far+near)/nf;  a[14] = far*nn/nf;
    a[11] = -1.0;
  }
  if ( ai ) {
    memset ( ai, 0, 16*sizeof(GLfloat) );
    ai[0] = rl/nn;         ai[12] = (right+left)/nn;
    ai[5] = tb/nn;         ai[13] = (top+bottom)/nn;
    ai[14] = -1.0;
    ai[11] = nf/(far*nn);  ai[15] = (far+near)/(far*nn);
  }
} /*M4x4Frustumf*/
```

Procedura `M4x4Frustumf` przedstawiona na listingu 6.1 wpisuje współczyniki tych macierzy do tablic wskazywanych przez parametry. Jeśli któraś macierz jest niepotrzebna, to można zamiast tablicy podać parametr `NULL`.

Zwróćmy uwagę, że przekształcenie współrzędnej *kartezjańskiej* z w odwzorowaniu ostrosłupa widzenia na kostkę standardową jest nieliniowe; w istocie jest to funkcja homograficzna. W przedziale $[-f, -n]$ funkcja ta jest monotonicznie malejąca, dzięki czemu testy widoczności wykonywane przez porównywanie współrzędnych z punktów w kostce standardowej (bliżej obserwatora jest ten punkt, którego współrzędna z jest mniejsza) są poprawne.

Ustalając wartości parametrów n i f, należy zachować umiar. Oczywiście, obiekty, które chcemy przedstawić na obrazie, muszą się mieścić w bryle widzenia, a więc zakres odległości określony przez te parametry musi być dostatecznie duży, ale im jest on większy, tym *mniejszą* dokładność będą mieć testy widoczności. Iloraz f/n powinien zatem być możliwie mały. Najlepiej, aby nie przekraczał rzędu kilkanaście, w ostateczności kilkadziesiąt[6].

Aby zapewnić jednakowe skalowanie wymiarów poziomych i pionowych na obrazie w klatce o szerokości w i wysokości h pikseli utworzonym na ekranie, który ma współczynnik aspektu a, należy przyjąć parametry spełniające warunek $r - l : t - b = aw : h$. Liczba

$$d = \sqrt{(r-l)^2 + (t-b)^2}$$

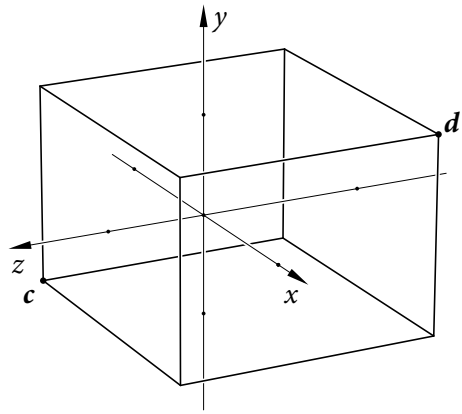
[6] Brakiem umiaru zajmujemy się w podrozdziale 13.6

jest długością przekątnej przedniej ściany ostrosłupa widzenia. Iloraz n/d określa rodzaj „obiektywu" realizującego rzutowanie: jeśli jest mały, to mamy obiektyw szerokokątny, a ze wzrostem tego ilorazu otrzymujemy coraz większe zbliżenia obiektów na obrazie. W klasycznej fotografii małoobrazkowej klatka na błonie światłoczułej ma wymiary 36 mm × 24 mm, zatem jej przekątna ma długość ok. 43 mm. Obiektywy o długości ogniskowej 50 mm, nazywane **standardowymi**, były chyba najczęściej używane w fotografii amatorskiej; dla takich obiektywów $n/d \approx 1.15$.[7] W grafice komputerowej lepiej jest przyjmować parametry, dla których iloraz n/d jest większy (orientacyjnie od 2 do ok. 5) i oddalić się od przedmiotów[8], aby osłabić efekt przerysowania.

Przyjęcie parametrów $l = -r$ i $b = -t$ daje symetryczny ostrosłup widzenia; punkty na jego osi symetrii są rzutowane na środek klatki. Zdarzają się sytuacje, gdy potrzebne są niesymetryczne bryły widzenia: najczęściej wtedy, gdy chcemy wykonać parę obrazów do stereoskopii.

6.3. Rzutowanie równoległe

Rzutowanie równoległe jest podstawą rysunku technicznego; obejmuje ono rzutowanie prostopadłe i opisaną dalej aksonometrię. Bryła widzenia dla takiego rzutowania jest prostopadłościanem. Sześć parametrów opisujących tę bryłę w układzie obserwatora ma te same nazwy co parametry dla rzutowania perspektywicznego. Przekształcenie bryły widzenia na kostkę standardową jest afiniczne, a dokładniej, każda ze współrzędnych x, y, z jest poddawana niezależnie przekształceniu afinicznemu (tj. skalowaniu z przesunięciem).



Rysunek 6.4. Bryła widzenia dla rzutowania równoległego

Na rysunku 6.4 mamy przykładową bryłę widzenia dla rzutowania równoległego z zaznaczonymi punktami $\boldsymbol{c} = (l, b, -n)$ i $\boldsymbol{d} = (r, t, -f)$. Macierz określająca przekształcenie tej

[7] Tradycyjnie w fotografii symbol f jest używany do oznaczenia długości ogniskowej obiektywu, a n oznacza względny otwór przysłony. Usiłujemy się tym nie przejmować.

[8] O ile jest to możliwe; uczestnik gry komputerowej zwykle musi widzieć scenę „fotografowaną" przez kamerę znajdującą się w pomieszczeniu z przedmiotami do narysowania.

bryły na kostkę standardową i odwrotność tej macierzy są opisane wzorami

$$P = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{l+r}{l-r} \\ 0 & \frac{2}{t-b} & 0 & \frac{b+t}{b-t} \\ 0 & 0 & \frac{2}{n-f} & \frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad P^{-1} = \begin{bmatrix} \frac{r-l}{2} & 0 & 0 & \frac{l+r}{2} \\ 0 & \frac{t-b}{2} & 0 & \frac{b+t}{2} \\ 0 & 0 & \frac{n-f}{2} & \frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Listing 6.2 przedstawia procedurę, która konstruuje macierz przekształcenia takiej bryły na kostkę standardową, a także macierz przekształcenia odwrotnego.

Listing 6.2. Procedura `M4x4Orthof`

C

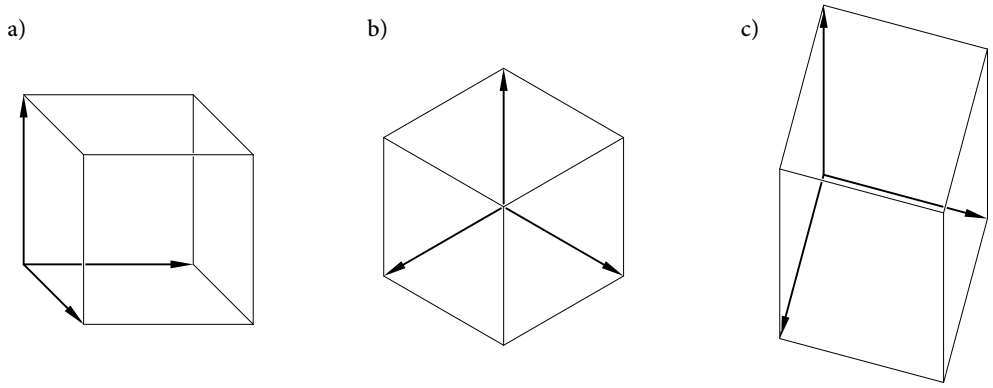
```
void M4x4Orthof ( GLfloat a[16], GLfloat ai[16],
                  float left, float right, float bottom,
                  float top, float near, float far )
{
  float rl, tb, nf;

  rl = right-left;  tb = top-bottom;  nf = near-far;
  if ( a ) {
    memset ( a, 0, 16*sizeof(GLfloat) );
    a[0] = 2.0/rl;   a[12] = -(right+left)/rl;
    a[5] = 2.0/tb;   a[13] = -(top+bottom)/tb;
    a[10] = 2.0/nf;  a[14] = (far+near)/nf;
    a[15] = 1.0;
  }
  if ( ai ) {
    memset ( ai, 0, 16*sizeof(GLfloat) );
    ai[0] = 0.5*rl;   ai[12] = 0.5*(right+left);
    ai[5] = 0.5*tb;   ai[13] = 0.5*(top+bottom);
    ai[10] = 0.5*nf;  ai[14] = 0.5*(far+near);
    ai[15] = 1.0;
  }
} /*M4x4Orthof*/
```

Dla $n < f$ współczynnik $\frac{2}{n-f}$ w trzecim wierszu i kolumnie jest ujemny, a ostatni współczynnik na diagonali macierzy P jest dodatni. W konsekwencji przekształcenie współrzędnej z jest funkcją malejącą. Jeśli dwa punkty po zrzutowaniu mają ten sam obraz, to punkt, którego współrzędna *z w układzie kostki standardowej* jest mniejsza, zasłania drugi punkt[9]. Ponieważ odwzorowanie przedziału zmienności współrzędnej z jest afiniczne, „rozdzielczość" głębokości (tj. informacji o głębokości punktów) w buforze głębokości OpenGL-a jest stała w całej bryle widzenia.

[9] Zauważmy, że tu nie można mówić o odległości tych punktów od obserwatora.

6.4. Aksonometria

Podstawą **aksonometrii** jest twierdzenie Pohlkego, zgodnie z którym dla dowolnej czwórki punktów nieleżących w jednej płaszczyźnie i dowolnej położonej na płaszczyźnie czwórki punktów, z których żadne trzy nie są współliniowe, istnieje taki rzut równoległy, w którym obraz pierwszej z tych czwórek jest figurą podobną[10] do drugiej czwórki. Możemy zatem dowolnie wybrać na płaszczyźnie obraz początku układu współrzędnych kartezjańskich w przestrzeni i obrazy wersorów osi, które muszą tylko mieć różne kierunki. Z twierdzenia wynika, że jest to poprawna konstrukcja pewnego rzutu równoległego i podobieństwa. Rysunek 6.5 przedstawia rzuty aksonometryczne najczęściej stosowane w rysunku technicznym.



Rysunek 6.5. Obrazy sześcianu w aksonometrii: a) kawalerskiej, b) izometrycznej, c) wojskowej

Aksonometrię w OpenGL-u otrzymamy, wybierając macierz V (macierz przejścia od układu współrzędnych świata do układu obserwatora) opisującą dowolne różnowartościowe przekształcenie afiniczne oraz macierz P zgodnie z opisem w podrozdziale 6.3. Napisanie procedury konstruującej odpowiednie macierze na podstawie wymiarów klatki oraz długości (zwanych **skróceniami aksonometrycznymi**) i kątów nachylenia obrazów wersorów osi układu współrzędnych świata jest prostym i pożytecznym ćwiczeniem.

6.5. Rzutowanie dla grafiki dwuwymiarowej

W pewnych sytuacjach występuje potrzeba określenia takiego rzutowania, aby współrzędne x, y w układzie obserwatora (albo modelu — patrz podrozdz. 6.6) były tożsame ze współrzędnymi ξ', η' w oknie (początek tego układu, przypomnijmy, jest w lewym górnym narożniku okna, współrzędna η' rośnie do dołu, a jednostki osi mają długości równe szerokości i wysokości jednego piksela). Najczęściej jest to potrzebne wtedy, gdy chcemy wykorzystać OpenGL-a do wykonania grafiki dwuwymiarowej, na przykład narysować menu z wihajstrami w oknie. Aby określić takie przekształcenie dla klatki, której górny lewy narożnik ma

[10] Chodzi o podobieństwo geometryczne, czyli złożenie izometrii z jednokładnością.

współrzędne ξ', η' i która ma szerokość w i wysokość h, położonej w oknie o wysokości H, możemy wykonać następujące instrukcje:

```
glViewport ( xiprime, H-h-etaprime, w, h );
M4x4Orthof ( a, NULL, (float)xiprime-0.5, (float)(xiprime+w)-0.5,
             (float)(H-h-etaprime)-0.5, (float)(H-1-etaprime)-0.5,
             -1.0, 1.0 );
```

a następnie przesłać współczynniki macierzy P przekształcenia z tablicy a do odpowiedniej zmiennej jednolitej. Jeśli chcemy, aby układ miał początek w górnym lewym narożniku *klatki* (która nie musi być całym oknem), to odpowiednią macierz przekształcenia otrzymamy, wykonując instrukcję

```
M4x4Orthof ( a, NULL, -0.5, (float)w-0.5,
               (float)h-0.5, -0.5, -1.0, 1.0 );
```

Zauważmy korekty przesunięcia w pionie i poziomie o pół piksela; one są wprowadzone dlatego, że linie $\eta' = 0$ i $\xi' = 0$ przechodzą przez *środki* pikseli odpowiednio w pierwszym wierszu i w pierwszej kolumnie pikseli w oknie lub klatce. Chcemy, aby liczby *całkowite* x, y, będące współrzędnymi punktu w bryle widzenia, po przekształceniu odpowiadały prostym przechodzącym przez środki pikseli w odpowiedniej kolumnie i wierszu rastra w oknie.

W tym miejscu (cokolwiek przedwcześnie) wspomnę, że jest też inna możliwość wprowadzenia tej korekty. Polega ona na przedeklarowaniu zmiennej wejściowej `gl_FragCoord` szadera fragmentów z kwalifikatorem układu `layout(pixel_center_integer)`, zobacz s. 199.

6.6. Przekształcenia rzutowanych wierzchołków

Położenie wierzchołka wprowadzanego do potoku przetwarzania grafiki przez etap pobierania wierzchołków jest zwykle reprezentowane przez wektor o czterech współrzędnych, X, Y, Z, W. To są zazwyczaj opisane w rozdziale 5 współrzędne jednorodne. W procesie tworzenia obrazu używamy wielu układów współrzędnych (kartezjańskich i związanych z nimi współrzędnych jednorodnych) w przestrzeni, w której znajduje się rysowana scena. Wyróżnimy cztery układy[11]:

Układ współrzędnych modelu — w tym układzie podajemy współrzędne wierzchołków obiektu; to one są przechowywane w pamięci GPU w odpowiednim buforze i to te współrzędne etap pobierania wierzchołków podaje na wejście szadera wierzchołków.

[11] Ściśle biorąc, wyróżniamy cztery *klasy* układów współrzędnych. Każdy obiekt (model) może być zdefiniowany w swoim układzie, innym niż pozostałe obiekty. Możemy mieć też więcej niż jednego obserwatora (np. w stereoskopii, podczas tworzenia obrazów obiektów odbitych w lustrze lub obserwatorów związanych ze źródłami światła w algorytmie cieni) i każdy obserwator będzie miał inną bryłę widzenia, a więc też inaczej określony w świecie układ współrzędnych kostki standardowej. Tylko układ świata jest tylko jeden.

Układ współrzędnych świata — wyróżniony układ, w którym poszczególne obiekty są odpowiednio rozmieszczone względem siebie. W tym układzie trzeba określić położenie obserwatora i w tym układzie jest obliczane oświetlenie obiektów.

Układ współrzędnych obserwatora — początek tego układu (w przypadku rzutowania perspektywicznego) jest położeniem obserwatora, który patrzy w kierunku ujemnej półosi z. W tym układzie opisujemy bryłę widzenia.

Układ kostki standardowej — współrzędne wierzchołków w tym układzie są przekazywane z części przedniej potoku przetwarzania grafiki do etapu obcinania, po którym dokonywana jest rasteryzacja, przeprowadzana także w tym układzie współrzędnych. Współrzędne w układzie kostki standardowej są oznaczane skrótem NDC (*normalized device coordinates*).

Uwaga: Jeśli stosowane rzutowanie nie jest równoległe, to w przestrzeni, w której wcześniej wymienione układy współrzędnych są kartezjańskie, układ kostki standardowej nie jest układem współrzędnych kartezjańskich.

W części przedniej potoku przetwarzania grafiki szadery muszą dokonać przejścia od współrzędnych podanych w układzie modelu do układu kostki standardowej. Obliczenie to może być wykonane przez *dowolny* szader części przedniej — wierzchołków, sterowania rozdrabnianiem, rozdrabniania lub geometrii. Potrzebne przekształcenie jest złożeniem trzech przekształceń opisanych przez macierze 4×4: macierz M opisuje przejście od układu modelu do układu świata, macierz V opisuje przejście od układu świata do obserwatora, macierz P opisuje przejście od układu obserwatora do układu kostki standardowej. Jeśli symbolem $\boldsymbol{A}$ oznaczymy wektor współrzędnych jednorodnych wierzchołka w układzie modelu, to któryś szader części przedniej ma obliczyć wektor

$$\boldsymbol{A}' = PVM\boldsymbol{A}$$

i przekazać go dalej. Macierze M, V i P będziemy nazywać odpowienio **macierzą modelu**, **macierzą widoku** i **macierzą rzutowania**. Zazwyczaj są one przechowywane w odpowiednich zmiennych jednolitych (pierwszy przykład zobaczymy już wkrótce).

Uwaga: Jeśli szader fragmentów ma wykonać obliczenie oświetlenia, to trzeba też przekazać dalej wektor $M\boldsymbol{A}$ zawierający współrzędne wierzchołka w układzie świata.

Uwaga: Można utożsamić układy modelu i świata lub świata i obserwatora. W tym celu wystarczy przyjąć macierz M lub V jednostkową, można też napisać szader, który nie uwzględnia odpowiedniego przejścia między układami, tj. nie wykonuje mnożenia wektora przez macierz tego przejścia, co jest równoważne pomnożeniu tego wektora przez macierz jednostkową.

Dodatkowo szadery mogą przeprowadzać obliczenia w układach współrzędnych (kartezjańskich lub barycentrycznych) określonych w dziedzinie rozdrabnianego płata lub w dziedzinie tekstury; tym będziemy się zajmować w rozdziałach 12, 15, i 19–22.

7

Pierwsza aplikacja

Aplikacja opisana w tym rozdziale powstała przez dołączenie do szkieletu aplikacji FreeGLUT-a z rozdziału 3 części graficznej, która tworzy obraz dwudziestościanu foremnego. Najpierw przedstawię szadery, nieformalnie omawiając użyte w nich konstrukcje języka GLSL, a następnie procedury w C, które robią wszystko, co trzeba, aby powstał obraz.

7.1. Szadery

Program, który zostanie użyty w tej aplikacji, składa się z tylko dwóch szaderów, wierzchołków i fragmentów, przedstawionych na listingach 7.1 i 7.2.

Pierwsza linia każdego szadera[1] musi zawierać informację o użytej wersji języka GLSL, w postaci liczby całkowitej podanej po symbolu preprocesora `#version`. W naszym przykładzie jest to wersja 4.2.

W liniach 3 i 4 na listingu 7.1 są opisane atrybuty wierzchołka podanego na wejście szadera wierzchołków; dane wejściowe są zadeklarowane z użyciem słowa kluczowego `in`. Słowo kluczowe `layout` z zawartością podanego za nim nawiasu jest **kwalifikatorem** (*qualifier*) zmiennej. Informacje podane w kwalifikatorach w liniach 3 i 4 definiują **numery miejsc** atrybutów wierzchołka na wejściu szadera: zmienna `in_Position`, która zawiera wektor współrzędnych jednorodnych położenia wierzchołka, ma numer miejsca `0`, a zmienna `in_Colour` zawierająca wektor współrzędnych *r*, *g*, *b* koloru wierzchołka ma numer miejsca `1`. Aplikacja, tworząc obiekt tablicy wierzchołków (*vertex array object*, *VAO*), podaje te numery razem z informacją, gdzie w tablicy wierzchołków należy odnaleźć odpowiednie atrybuty, co jest opisane dalej.

Szader wierzchołków powinien nadać wartość zmiennej `gl_Position`, która jest częścią struktury wyjściowej (zadeklarowanej „odgórnie"). Dodatkowy atrybut wierzchołka przetworzonego, w tym przykładzie kolor, jest w linii 17 przypisywany zmiennej wyjściowej `Colour`.

[1] która nie jest pusta lub nie zawiera tylko spacji i komentarzy

Listing 7.1. Szader wierzchołków pierwszej aplikacji

GLSL

```
1: #version 420
2:
3: layout(location=0) in vec4 in_Position;
4: layout(location=1) in vec3 in_Colour;
5:
6: out vec3 Colour;
7:
8: uniform TransBlock {
9:     mat4 mm;
10:    mat4 vm;
11:    mat4 pm;
12:  } trb;
13:
14: void main ( void )
15: {
16:   gl_Position = trb.pm * (trb.vm * (trb.mm * in_Position));
17:   Colour = in_Colour;
18: } /*main*/
```

W liniach 8–12 jest zadeklarowany blok zmiennych jednolitych, który zawiera trzy macierze; pierwsza (`trb.mm`) to macierz modelu M, która opisuje przejście od układu współrzędnych, w którym jest zdefiniowany obiekt (tj. w którym są podane współrzędne jego wierzchołków) do układu świata, druga to macierz widoku V (`trb.vm`) opisująca przejście od układu świata do układu obserwatora, a trzecia macierz rzutowania P (`trb.pm`) reprezentuje przekształcenie bryły widzenia na kostkę standardową skonstruowane zgodnie z opisem w poprzednim rozdziale. Blok zmiennych jednolitych ma dwie nazwy. **Nazwa zewnętrzna** `TransBlock` jest używana przez aplikację w celu uzyskania dostępu do tych zmiennych jednolitych.

Dygresja: W starym OpenGL-u przekształcenie wierzchołków jest wykonywane przy użyciu dwóch macierzy, o nazwach *ModelView* i *Projection*. Pierwsza z nich jest iloczynem macierzy modelu i widoku; druga z nich to macierz rzutowania. Z powodów praktycznych (z którymi spotkamy się dalej) chyba lepiej jest rozdzielić przekształcenia, których złożenie opisuje pierwsza z tych macierzy, tak jak to tu zrobiłem.

Obliczenie realizuje procedura `main` (linie 14–18), która musi mieć pustą listę parametrów i pusty typ wyniku. W linii 16 obliczany jest wektor współrzędnych położenia wierzchołka w kostce standardowej; zwracam uwagę na nawiasy, których rozmieszczenie z pięciu możliwych sposobów obliczania iloczynu trzech macierzy i wektora wybiera sposób najtańszy. Atrybut koloru w linii 17 jest po prostu kopiowany do zmiennej wyjściowej.

Szader fragmentów na listingu 7.2 spełnia bardzo proste zadanie: kopiuje kolor otrzymany na wejściu na wynik, który zgodnie z deklaracją w linii 5 ma być przypisany do zmiennej `out_Colour`. Do współrzędnych r, g, b koloru szader dołącza czwartą współrzędną, tzw. **składową alfa**, której zastosowania zbadamy w dalszych rozdziałach.

Listing 7.2. Szader fragmentów pierwszej aplikacji

GLSL

```
#version 420

in vec3 Colour;

out vec4 out_Colour;

void main ( void )
{
  out_Colour = vec4 ( Colour, 1.0 );
} /*main*/
```

Należy zwrócić uwagę, że wyjście szadera wierzchołków jest zbudowane identycznie jak wejście szadera fragmentów — jest tu jedna zmienna tego samego typu i o identycznej nazwie. Między szaderami można przekazać więcej danych w osobnych zmiennych, które muszą sobie odpowiadać tak samo dokładnie jak w tym przykładzie. Wygodniej jest jednak zdefiniować struktury złożone z pewnej liczby pól. Z tego rozwiązania będziemy korzystać w bardziej skomplikowanych aplikacjach.

Deklaracje zmiennych takich jak `Colour` można poprzedzić **kwalifikatorem miejsca**, czyli napisać na przykład `layout(location=0)` przed początkiem deklaracji (słowem kluczowym `out` oraz `in`). Poszczególne zmienne muszą mieć przydzielone różne miejsca[2].

Szader fragmentów wykonuje obliczenie dla pojedynczego piksela obrazu odpowiedniego prymitywu (punktu, odcinka lub trójkąta) i ma za zadanie tylko podać kolor lub ewentualnie podjąć decyzję o zaniechaniu rysowania tego piksela. Ale jaki jest kolor podany w zmiennej `Colour`, jeśli wierzchołki trójkąta mają różne kolory, a piksel odpowiada punktowi innemu niż wierzchołek? Otóż, wszelkie atrybuty[3] są **interpolowane** między wartościami podanymi dla wierzchołków. W ten sposób realizowane jest **cieniowanie** (*shading*, stąd nazwa szadery) i na przykład piksel odpowiadający środkowi ciężkości trójkąta będzie miał kolor otrzymany jako średnia z kolorów wierzchołków trójkąta. Kolor jest atrybutem wektorowym; interpolowane są wszystkie jego współrzędne.

7.2. Przygotowanie obiektów w programie

Procedura `InitMyWorld` (wywołana w linii 70 programu na listingu 3.1) ma za zadanie przygotowanie programu szaderów, skonstruowanie przekształceń, tzn. obliczenie współczynników macierzy przekształcenia modelu, przejścia do układu obserwatora oraz przejścia do układu kostki standardwej i umieszczenie ich w bloku zmiennych jednolitych, a także przygotowanie reprezentacji obiektu do narysowania. Podprogramy wywoływane przez tę procedurę są opisane niżej.

[2] „Ręczne" przydzielanie miejsc (zamiast zostawienia tego do zrobienia kompilatorowi) jest konieczne w szaderach, które mają pracować w potokach programów (zobacz p. 4.5.2) oraz w aplikacjach Vulkana.

[3] Z wyjątkiem tych, które mają kwalifikator `flat`, zobacz p. 12.4.5. Sposób interpolacji jest opisany w p. 12.4.4.

Listing 7.3. Procedura `InitMyWorld`

```C
1: void InitMyWorld ( int argc, char *argv[], int width, int height )
2: {
3:   LoadMyShaders ();
4:   InitModelMatrix ();
5:   InitViewMatrix ();
6:   ConstructIcosahedronVAO ();
7:   ResizeMyWorld ( width, height );
8: } /*InitMyWorld*/
```

7.2.1. Przygotowanie programu szaderów

Listing 7.4 przedstawia procedurę `LoadMyShaders` oraz zmienne globalne aplikacji, w których będzie zapamiętany identyfikator programu szaderów i inne informacje konieczne do współpracy części aplikacji działających na CPU i GPU. Zakładam, że przedstawione na listingach 7.1 i 7.2 szadery wierzchołków i fragmentów są zapisane (w całości) w plikach o nazwach `app1.vert.glsl` i `app1.frag.glsl`.

W liniach 14–16 procedura `LoadMyShaders` wywołuje opisane w rozdziale 4 procedury, które czytają z pliku i kompilują szadery (nazwy plików są podane w linii 8), a następnie łączą je w program. Identyfikatory szaderów są zapisywane w tablicy `shader_id`, a identyfikator programu w zmiennej globalnej `program_id`.

Program szaderów korzysta z bloku zmiennych jednolitych, który zawiera trzy macierze 4×4 o współczynnikach rzeczywistych (zmiennopozycyjnych pojedynczej precyzji). Instrukcje w liniach 17–22 odczytują z programu szaderów niezbędne informacje o tym bloku zmiennych jednolitych. W liniach 23–25 jest tworzony i przygotowywany do pracy obiekt bufora zmiennych jednolitych (*uniform buffer object*, *UBO*), w którym będą przechowywane macierze przekształceń.

Procedura `glGetUniformBlockIndex` w linii 17 zwraca **indeks bloku zmiennych jednolitych** — jest to identyfikator bloku w utworzonym programie szaderów[4]. Parametry tej procedury to identyfikator programu i napis ASCIIZ, który jest *zewnętrzną* nazwą struktury zadeklarowanej w programie szaderów (zobacz linię 10 na listingu 7.4 i linię 8 na listingu 7.1).

Procedura `glGetActiveUniformBlockiv` w linii 18–19 przypisuje zmiennej `trbsize` wielkość (w bajtach) bloku; bufor, który utworzymy dla tego bloku musi mieć co najmniej taką wielkość. Następnie procedura `glGetUniformIndices` wyciąga z programu informacje o poszczególnych polach struktury będącej zmienną jednolitą. Mamy tu trzy pola (stąd drugi parametr ma wartość 3), których nazwy (napisy ASCIIZ) są podane w tablicy napisów przekazanej jako parametr trzeci. Parametr czwarty jest tablicą, do której zostają wpisane **indeksy** (identyfikatory) tych pól.

[4] W istocie jest to indeks do utworzonej przez procedurę `glLinkProgram` tablicy opisów bloków zmiennych jednolitych. Więcej informacji na ten temat jest w p. 11.5.3.

Listing 7.4. Procedura `LoadMyShaders`

C

```
1: GLuint program_id;
2: GLuint trbi, trbuf;
3: GLint  trbsize, trbofs[3];
4:
5: void LoadMyShaders ( void )
6: {
7:   static const char *filename[] =
8:     { "app1.vert.glsl", "app1.frag.glsl" };
9:   static const GLchar *UTBNames[] =
10:    { "TransBlock", "TransBlock.mm", "TransBlock.vm", "TransBlock.pm" };
11:  GLuint shader_id[2], ind[3];
12:  int i;
13:
14:  shader_id[0] = CompileShaderFiles ( GL_VERTEX_SHADER, 1, &filename[0] );
15:  shader_id[1] = CompileShaderFiles ( GL_FRAGMENT_SHADER, 1, &filename[1]);
16:  program_id = LinkShaderProgram ( 2, shader_id, NULL );
17:  trbi = glGetUniformBlockIndex ( program_id, UTBNames[0] );
18:  glGetActiveUniformBlockiv ( program_id, trbi,
19:                                GL_UNIFORM_BLOCK_DATA_SIZE, &trbsize );
20:  glGetUniformIndices ( program_id, 3, &UTBNames[1], ind );
21:  glGetActiveUniformsiv ( program_id, 3, ind, GL_UNIFORM_OFFSET, trbofs );
22:  glUniformBlockBinding ( program_id, trbi, 0 );
23:  glGenBuffers ( 1, &trbuf );
24:  glBindBufferBase ( GL_UNIFORM_BUFFER, 0, trbuf );
25:  glBufferData ( GL_UNIFORM_BUFFER, trbsize, NULL, GL_DYNAMIC_DRAW );
26:  for ( i = 0; i < 2; i++ )
27:    glDeleteShader ( shader_id[i] );
28:  ExitIfGLError ( "LoadMyShaders" );
29: } /*LoadMyShaders*/
```

Wywołana w linii 21 procedura `glGetActiveUniformsiv` wyciąga informacje o przesunięciach (w bajtach) poszczególnych pól w strukturze zmiennej jednolitej względem jej początku. Zwróćmy uwagę na sposób używania indeksów pól struktury otrzymanych w poprzedniej instrukcji.

W linii 22 jest wywołana procedura `glUniformBlockBinding`. Jej zadaniem jest ustalenie, że blok zmiennych jednolitych `TransBlock` ma być skojarzony z punktem dowiązania o numerze `0` w celu `GL_UNIFORM_BUFFER` w kontekście OpenGL-a[5]. Warto tu odwołać się do zamieszczonego w pierwszym rozdziale opisu kojarzenia UBO z blokami zmiennych jednolitych (zobacz rys. 1.3). Utworzony przez dalsze instrukcje bufor, do którego aplikacja będzie wpisywać wartości zmiennych jednolitych w bloku `TransBlock` szadera, będzie przywiązany do punktu `0`.

[5]Numer punktu dowiązania bloku zmiennych jednolitych można jawnie zapisać w treści szadera, poprzedzając blok, tj. słowo kluczowe `uniform`, kwalifikatorem na przykład `layout(binding=1)`. Procedura `glUniformBlockBinding` jest władna zmienić ten numer.

Procedura `glGenBuffers` w linii 23 tworzy (w tym przypadku jeden) obiekt bufora — w tym momencie jeszcze nie wiadomo, jaki długi to będzie bufor, jakie w nim będą dane ani jak będzie używany. Jego identyfikator ląduje w zmiennej `trbuf`, przerobionej *ad hoc* na tablicę o długości 1, ponieważ ostatni parametr procedury `glGenBuffers` ma być tablicą.

Podstawowa procedura, która przywiązuje bufor do wskazanego celu, ma nazwę `glBindBuffer`; będziemy jej wielokrotnie używać. W kontekście OpenGL-a jest kilkanaście celów, z których każdy ma swoje zastosowania. Pewne cele, w tym właśnie `GL_UNIFORM_BUFFER`, są indeksowane, tj. zawierają tablice punktów dowiązania. Procedura `glBindBufferBase` może być wywołana dla celu z taką tablicą; oprócz przywiązania bufora do samego celu, procedura przywiązuje go również do wskazanego przez drugi parametr punktu dowiązania w tym celu (nie zmieniając, co ważne, pozostałych punktów dowiązania w tablicy). Jeśli program szaderów zawiera wiele bloków zmiennych jednolitych, to każdy UBO będzie przywiązany do innego, (mam nadzieję) właściwego punktu dowiązania. Zatem, procedura `glBindBufferBase` w linii 24 przywiązuje nowo utworzony bufor do celu `GL_UNIFORM_BUFFER` i do punktu dowiązania 0 w tym celu; dalsze działania na buforze odbywają się za pośrednictwem tego celu.

Procedura `glBufferData` w zasadzie służy do przesłania danych z pamięci CPU do bufora, ale w linii 25 jako adres tablicy z danymi jest podany parametr `NULL`. Rzecz w tym, że jest to sposób na ustalenie długości bufora (bo to jest dokonywane w momencie pierwszego wywołania procedury `glBufferData` dla nowo utworzonego bufora) oraz postulowanego sposobu wykorzystywania bufora — stała `GL_DYNAMIC_DRAW` wskazuje, że będziemy chcieli dane do bufora wpisywać wielokrotnie. Zmienna `trbsize`, przechowująca potrzebną wielkość bufora, otrzymała wartość w linii 19.

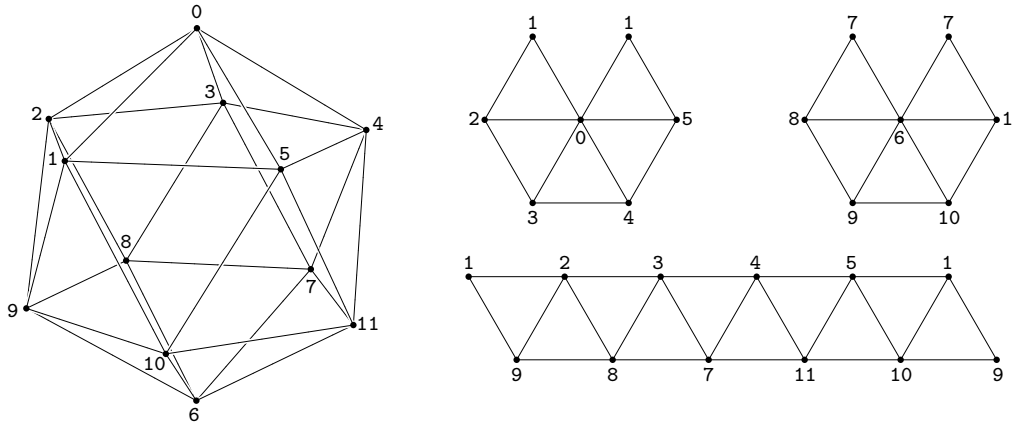
Z chwilą połączenia w program szadery przestają być potrzebne i dlatego w liniach 26–27 są kasowane, co zwalnia miejsce w pamięci. Bardziej skomplikowane aplikacje mogą zachować szadery, aby w trakcie działania (a nie tylko bezpośrednio po uruchomieniu) łączyć nowe programy.

7.2.2. Tworzenie obiektu tablicy wierzchołków

Rysunek 7.1 przedstawia wierzchołki i krawędzie dwudziestościanu; liczby na rysunku oznaczają numery wierzchołków. W aplikacji będziemy chcieli rysować dwudziestościan na trzy sposoby: jako zbiór wierzchołków, szkielet zbudowany z krawędzi albo jako wielościan (czyli zbiór nieprzezroczystych ścian).

Każdy z dwunastu wierzchołków dwudziestościanu jest końcem pięciu krawędzi i wierzchołkiem pięciu ścian. Procedura `glDrawArrays` może rysować odcinki i trójkąty, biorąc pary lub trójki kolejnych punktów z tablicy wierzchołków. Ale wtedy każdy wierzchołek musiałby wystąpić w tablicy pięciokrotnie, czego wolimy uniknąć. W tym celu do rysowania użyjemy procedury `glDrawElements`. Procedura ta posługuje się dodatkową tablicą liczb całkowitych — indeksów do tablicy wierzchołków; indeksy zajmują znacznie mniej miejsca, a ponadto w ten sposób eliminowane są błędy w kopiowaniu współrzędnych wierzchołka[6].

[6] Okazji do popełniania błędów i tak nie zabraknie.



Rysunek 7.1. Model dwudziestościanu foremnego: wachlarze i taśma trójkątowa

Tablica ta będzie przechowywana w osobnym buforze, który bywa określany skrótem IBO (od *index buffer object*) lub EBO (*element buffer object*).

Aby skrócić tablicę indeksów, zamiast oddzielnych odcinków lub trójkątów będziemy rysować łamane lub taśmy albo wachlarze trójkątów. Dla każdego odcinka lub trójkąta, z wyjątkiem pierwszego, trzeba podać tylko jeden indeks. Taśma i wachlarze trójkątów są narysowane schematycznie na rysunku 7.1; procedura konstruująca VAO z modelem (tj. atrybutami wierzchołków) dwudziestościanu i tablicę indeksów wierzchołków potrzebną do rysowania jego ścian i krawędzi jest podana na listingu 7.5.

Liczby

$$A = \sqrt{\frac{5-\sqrt{5}}{10}} \quad \text{i} \quad B = \sqrt{\frac{5+\sqrt{5}}{10}}$$

podane w liniach 5 i 6 spełniają następujące równania: $A^2 + B^2 = 1$ i $\frac{B}{A} = \frac{A}{B-A}$; pierwsze z nich oznacza, że wierzchołki podane w tablicy `vertpos` leżą na sferze jednostkowej. Liczby B i A spełniające drugie równanie pozostają w złotej proporcji — punkty w tablicy `vertpos`, które mają takie współrzędne, są wierzchołkami dwudziestościanu foremnego.

Procedura `glGenVertexArrays` w linii 23 tworzy nowy, początkowo pusty VAO. Jak zwykle w procedurach tworzących obiekty OpenGL-a (szadery, programy, bufory, tekstury itp.), procedura może utworzyć wiele VAO jednocześnie (tyle, ile określa pierwszy parametr). Ich identyfikatory są wpisywane do tablicy, która jest drugim parametrem (tu zmienna `icos_vao` jest potraktowana jak tablica jednoelementowa).

Nowo utworzony VAO został **przywiązany**, tj. uczyniony aktywnym w kontekście OpenGL-a, przez wywołanie procedury `glBindVertexArray` w linii 24. W linii 25 tworzymy trzy bufory na dane opisujące dwudziestościan; ich identyfikatory są wpisywane do tablicy `icos_vbo`; bufory te wypełniamy kolejno.

W liniach 26 i 32 pierwszy i drugi bufor są przez procedurę `glBindBuffer` kolejno przywiązywane do celu `GL_ARRAY_BUFFER`. W liniach 27–28 oraz 33–34 do bufora przywiązanego

do tego celu przesyłamy dane z zadeklarowanych w liniach 7–10 oraz 11–13 tablic. Ostatni parametr procedury `glBufferData` w obu tych przypadkach deklaruje, że zawartość tych buforów nie zmieni się przez cały czas działania aplikacji[7].

Uwaga: W OpenGL-u 4.5 można przesyłanie danych do buforów wykonać prościej, używając procedur `glNamedBufferData` i `glNamedBufferSubData`, których pierwszy parametr jest identyfikatorem *bufora*, a nie *celu*; pozostałe parametry są takie same jak parametry procedur `glBufferData` i `glBufferSubData`. Wywołań procedur `glNamedBufferData` i `glNamedBufferSubData` nie trzeba poprzedzać wywołaniem procedury `glBindBuffer`. Nadal jednak bufor musi być przywiązany do określonego celu, gdy wymagają tego pewne procedury, na przykład opisane dalej `glVertexAttribPointer` i `glDrawElements`.

Procedura `glEnableVertexAttribArray` zapisuje w bieżącym VAO informację o tym, że atrybut o numerze miejsca podanym jako parametr znajduje się w tablicy, skąd ma być pobierany dla kolejnych wierzchołków. Numery miejsc atrybutów podane w liniach 29 i 35 to są te same numery miejsc, które były zapisane w kwalifikatorze `layout` w liniach 3 i 4 na listingu 7.1. Alternatywne rozwiązanie polega na umieszczeniu wartości atrybutu (jednakowej dla wszystkich wierzchołków) w tzw. **zmiennej statycznej**. Ma to sens, jeśli wszystkie wierzchołki mają na przykład ten sam kolor i chcemy go podać tylko raz. Wartość atrybutu przypisuje się zmiennej statycznej, wywołując jedną z procedur z rodziny `glVertexAttrib*`[8]. Pobieranie dowolnego atrybutu z tablicy „włączone" przez `glEnableVertexAttribArray` można „wyłączyć", wywołując procedurę `glDisableVertexAttribArray` z odpowiednim parametrem.

Wreszcie, procedura `glVertexAttribPointer` przekazuje do VAO informację na temat miejsc w tablicy umieszczonej w buforze (przywiązanym w danym momencie do celu `GL_ARRAY_BUFFER`), w których znajdują się wartości atrybutu dla kolejnych wierzchołków. Atrybuty wprowadzone za pomocą tej procedury na wejściu szadera wierzchołków mają współrzędne zmiennopozycyjne reprezentowane w pojedynczej precyzji[9].

Uwaga: Każdy atrybut zadeklarowany na wejściu szadera wierzchołków musi być pobierany z bufora, którego zawartość trzeba opisać za pomocą procedury `glVertexAttribPointer`, albo ze zmiennej statycznej i wtedy należy go „wyłączyć". Zaniedbanie tego (nawet jeśli program szaderów ma ten atrybut zignorować) prowadzi do tyleż spektakularnych, co niepożądanych (i niepojętych, jeśli autor aplikacji nie wie, o co chodzi) efektów na obrazie.

Pierwszy parametr procedury `glVertexAttribPointer` jest numerem atrybutu. Drugi parametr określa liczbę podanych współrzędnych, a trzeci i czwarty opisują sposób ich reprezentowania. Przypomnijmy, że szader wierzchołków (na listingu 7.1) otrzymuje oba atrybuty

[7] Parametr ten daje *wskazówkę* OpenGL-owi, który może wybrać sposób utworzenia bufora optymalny ze względu na czas dostępu procedur, które będą najczęściej wykonywać działania na tym buforze; konkretna implementacja może też tę wskazówkę zignorować.

[8] Czyli na przykład `glVertexAttrib3fv` albo `glVertexAttrib4ub` — zobacz opis przyrostków nazw procedur OpenGL-a w rozdziale 2; pierwszy parametr każdej procedury z tej rodziny jest numerem atrybutu, a pozostałe parametry, których typ jest określony przez przyrostek, podają współrzędne wektora atrybutu.

[9] Do wprowadzania atrybutów, które na wejście szadera wierzchołków mają być podane jako liczby całkowite służy procedura `glVertexAttribIPointer`, a atrybuty w podwójnej precyzji (która ma być utrzymana w potoku przetwarzania grafiki) wprowadza się, wywołując procedurę `glVertexAttribLPointer`.

Listing 7.5. Procedura `ConstructIcosahedronVAO`

C

```
GLuint icos_vao, icos_vbo[3];

void ConstructIcosahedronVAO ( void )
{
#define A 0.52573115
#define B 0.85065085
  static const GLfloat vertpos[12][3] =
    {{ -A,0.0, -B},{  A,0.0, -B},{0.0, -B, -A},{ -B, -A,0.0},
     { -B,  A,0.0},{0.0,  B, -A},{  A,0.0,  B},{ -A,0.0,  B},
     {0.0, -B,  A},{  B, -A,0.0},{  B,  A,0.0},{0.0,  B,  A}};
  static const GLubyte vertcol[12][3] =
    {{255,0,0},{255,127,0},{255,255,0},{127,255,0},{0,255,0},{0,255,127},
     {0,255,255},{0,127,255},{0,0,255},{127,0,255},{255,0,255},{255,0,127}};
  static const GLubyte vertind[62] =
     { 0, 1, 2, 0, 3, 4, 0, 5, 1, 9, 2, 8, 3, /* łamana, od 0 */
       7, 4, 11, 5, 10, 9, 6, 8, 7, 6, 11, 7,
       1, 10, 6,                              /* łamana, od 25 */
       2, 3, 4, 5, 8, 9, 10, 11,              /* 4 odcinki, od 28 */
       0, 1, 2, 3, 4, 5, 1,                   /* wachlarz, od 36 */
       6, 7, 8, 9, 10, 11, 7,                 /* wachlarz, od 43  */
       1, 9, 2, 8, 3, 7, 4, 11, 5, 10, 1, 9}; /* taśma, od 50 */

  glGenVertexArrays ( 1, &icos_vao );
  glBindVertexArray ( icos_vao );
  glGenBuffers ( 3, icos_vbo );
  glBindBuffer ( GL_ARRAY_BUFFER, icos_vbo[0] );
  glBufferData ( GL_ARRAY_BUFFER,
                 12*3*sizeof(GLfloat), vertpos, GL_STATIC_DRAW );
  glEnableVertexAttribArray ( 0 );
  glVertexAttribPointer ( 0, 3, GL_FLOAT, GL_FALSE,
                          3*sizeof(GLfloat), (GLvoid*)0 );
  glBindBuffer ( GL_ARRAY_BUFFER, icos_vbo[1] );
  glBufferData ( GL_ARRAY_BUFFER,
                 12*3*sizeof(GLubyte), vertcol, GL_STATIC_DRAW );
  glEnableVertexAttribArray ( 1 );
  glVertexAttribPointer ( 1, 3, GL_UNSIGNED_BYTE, GL_TRUE,
                          3*sizeof(GLubyte), (GLvoid*)0 );
  glBindBuffer ( GL_ELEMENT_ARRAY_BUFFER, icos_vbo[2] );
  glBufferData ( GL_ELEMENT_ARRAY_BUFFER,
                 62*sizeof(GLubyte), vertind, GL_STATIC_DRAW );
  glBindVertexArray ( 0 );
  ExitIfGLError ( "ConstructIcosahedronVAO" );
} /*ConstructIcosahedronVAO*/
```

typu vec4; tymczasem nasza procedura, zarówno dla położenia wierzchołka, jak i koloru, przesyła do bufora tylko 3 początkowe współrzędne. Współrzędne niepodane otrzymają

(w etapie pobierania wierzchołków w potoku przetwarzania grafiki) wartości domyślne — druga i trzecia współrzędna, w razie niepodania, byłyby równe 0, a czwarta 1. W ten sposób ze współrzędnych kartezjańskich punktu na płaszczyźnie lub w przestrzeni trójwymiarowej etap pobierania wierzchołków skonstruuje wektor współrzędnych jednorodnych, wymagany przez dalsze etapy potoku.

Trzeciemu parametrowi procedury `glVertexAttribPointer` w tej aplikacji nadajemy wartości `GL_FLOAT` i `GL_UNSIGNED_BYTE`, oznaczające odpowiednio liczby typu `GLfloat` i `GLubyte`. Czwarty parametr jest boolowski; jego wartość `GL_FALSE` oznacza, że podane wartości współrzędnych atrybutu należy przyjąć bezpośrednio. Podanie `GL_TRUE` (dozwolone tylko dla typów całkowitych) spowoduje **normalizację**, czyli podzielenie przez ustaloną liczbę; dla typu `GLubyte` dzielnik jest równy 255. Normalizacja wytwarza liczby rzeczywiste z przedziału $[0,1]$ (dla typów liczbowych bez znaku) albo $[-1,1]$ (dla typów ze znakiem).

Do bufora koloru, w linii 33–34, wpisujemy tylko 3 współrzędne koloru, opisujące składowe *r*, *g*, *b*. Gdyby zmienna `in_Colour` szadera wierzchołków była typu `vec4`, to czwarta współrzędna, czyli składowa alfa (*A*), otrzymałaby wartość 1, której domyślna interpretacja jest taka, że „farba" o podanym kolorze jest całkowicie nieprzezroczysta[10].

Piąty parametr procedury `glVertexAttribPointer` określa **krok** (*stride*) w tablicy; jest to odległość w bajtach początków danych dla kolejnych wierzchołków. Można podać wartość 0, jeśli dane są w buforze upakowane bez przerw (tak jak w naszej aplikacji), albo faktyczną odległość (tak jak w naszej aplikacji).

Ostatni parametr, będący z powodów historycznych wskaźnikiem[11], jest w istocie liczbą całkowitą — odległością w bajtach początku danych dla pierwszego wierzchołka od początku bufora. W programie w C możemy zdefiniować dowolny typ strukturalny z polami struktury opisującymi kilka (lub nawet wszystkie) atrybuty wierzchołka. Wtedy do VAO wpiszemy informacje o różnych atrybutach znajdujących się w jednym buforze; dla poszczególnych atrybutów, w kolejnych wywołaniach procedury `glVertexAttribPointer`, podamy odległości pól z tymi atrybutami od początku struktury.

W liniach 38–40 trzeci z utworzonych przez `glGenBuffers` w linii 25 buforów jest przywiązywany do celu `GL_ELEMENT_ARRAY_BUFFER`, po czym następuje przesyłanie do tego bufora tablicy indeksów. Indeksy muszą być liczbami całkowitymi bez znaku: 8-bitowymi (`GLubyte`), 16-bitowymi (`GLushort`) albo 32-bitowymi (`GLuint`). Ponieważ dwudziestościan ma 12 wierzchołków, ich indeksy są małymi liczbami, stąd najlepiej jest użyć dla nich typu `GLubyte`. Kolejne fragmenty ciągu liczb umieszczonego w tym buforze opisują dwie łamane i cztery oddzielne odcinki, dające w sumie wszystkie 30 krawędzi dwudziestościanu, oraz pokazane na rysunku 7.1 dwa wachlarze i jedną taśmę trójkątową, które zawierają wszystkie 20 ścian (zobacz też komentarze w liniach 15–21 na listingu 7.5).

W aktywnym VAO jest zapamiętywany także identyfikator bufora przywiązanego do celu `GL_ELEMENT_ARRAY_BUFFER`. W chwili ponownego uaktywnienia tego VAO (przed rysowa-

[10] Ale w tej aplikacji wartość składowej *A* jest ignorowana; aby ją wykorzystać, należy wybrać odpowiednią **funkcję mieszającą** (*blending function*), „działającą" w etapie końcowych operacji na buforze obrazu. Funkcja domyślna, tj. aktywna zanim się wybierze inną, powoduje przypisanie pikselowi wartości przekazanej na wyjście szadera fragmentów.

[11] W starym OpenGL-u parametr ten był adresem w pamięci RAM CPU.

niem) następuje przywiązanie tego bufora do tego celu, dzięki czemu nie trzeba tego robić osobną instrukcją. Ale jeśli jakiś bufor zostanie przywiązany do tego celu w trakcie działania programu, bez związku z umieszczaniem w pamięci GPU danych związanych z tym obiektem, to identyfikator tego bufora zostanie zapamiętany w aktywnym VAO, co może oznaczać zepsucie reprezentacji obiektu. Aby temu zapobiec, warto po przesłaniu wszystkich danych do buforów i uaktywnieniu atrybutów wierzchołków odwiązać VAO. Dokonuje tego procedura `glBindVertexArray` wywołana z parametrem `0` (linia `41`). VAO ponownie uaktywnimy przed i odwiążemy zaraz po narysowaniu obiektu.

Uwaga: W zasadzie *można* uniknąć korzystania z obiektu tablicy wierzchołków w aplikacji[12]. Mając umieszczone w buforach dane opisujące atrybuty wierzchołków, wywołanie procedury `glBindVertexArray` przed rysowaniem (listing 7.8, linia `4`) należałoby zastąpić wszystkimi kolejnymi wywołaniami procedur `glBindBuffer`, `glEnableVertexAttribArray` i `glVertexAttribPointer` zapisanymi w liniach `26`–`38` na listingu 7.5. Jak widać, aplikacja korzystająca z VAO jest prostsza.

7.2.3. Przekształcenia współrzędnych

Macierz przekształcenia modelu w pierwszej aplikacji będzie jednostkowa; to oznacza, że układ, w którym są podane współrzędne wierzchołków (układ współrzędnych modelu) jest układem świata; później to zmienimy. Procedura `InitModelMatrix` na listingu 7.6 wpisuje współczynniki macierzy jednostkowej do UBO, w miejscu odpowiadającym zmiennej jednolitej `TransBlock.mm`. Miejsce to jest określone przez podanie przesunięcia pola `mm` struktury `TransBlock` względem początku tej struktury — wielkość (w bajtach) tego przesunięcia (i przesunięć pozostałych dwóch pól) uzyskaliśmy, wykonując instrukcje w liniach `20`–`21` na listingu 7.4.

Procedura `glBindBuffer` w liniach `6` i `16` przywiązuje UBO (czyli bufor o identyfikatorze `trbuf`) do celu `GL_UNIFORM_BUFFER`. Szczerze mówiąc, w tej aplikacji jest to zbędne (bo nie ma w niej innych buforów przywiązywanych do tego celu, więc wystarczyło zrobić to tylko raz, po utworzeniu bufora), ale w ten sposób oszczędzimy sobie niespodzianek podczas przerabiania aplikacji. Procedura `glBufferSubData` wpisuje do bufora podaną w trzecim parametrze liczbę bajtów, zaczynając od miejsca (przesunięcia) określonego przez drugi parametr. Procedury `glBufferSubData` można użyć *po ustaleniu wielkości* bufora — to nastąpiło podczas pierwszego wywołania procedury `glBufferData` dla tego bufora.

Dwudziestościan, który będziemy rysować, jest wpisany w kulę o promieniu 1 i środku $(0, 0, 0)$ (tj. w początku układu świata). Na listingu 7.6 jest pokazana procedura inicjalizacji macierzy przejścia od układu świata do układu obserwatora, którego początek znajduje się w punkcie $(0, 0, 10)$ — przekształcenie jest przesunięciem o wektor $(0, 0, -10)$.

Macierz przekształcenia ostrosłupa widzenia na kostkę standardową musi być wyznaczona po nadaniu oknu wymiarów początkowych i po każdej zmianie jego wymiarów. Dla-

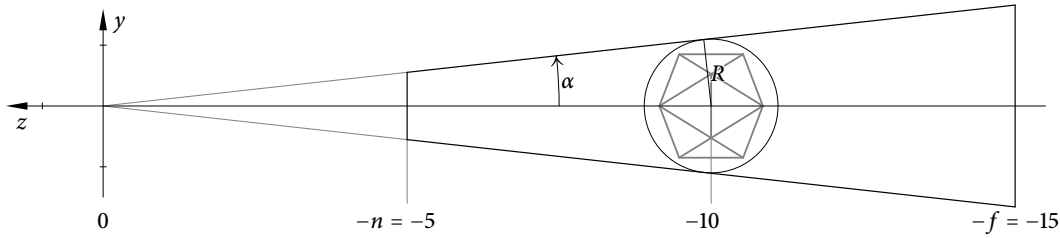
[12] To jednak może nie zadziałać; brak aktywnego VAO podczas rysowania w pewnych przypadkach powoduje błąd OpenGL-a, przy czym nie udało mi się dociec, dlaczego tak się dzieje. Dlatego lepiej jest używać VAO *zawsze*. Jeśli wierzchołki nie mają żadnych atrybutów podanych w buforach wierzchołków (VBO), to rozwiązaniem jest używanie pustego VAO. Przyda się to w drugiej i trzeciej aplikacji.

tego to obliczenie jest przeprowadzane w procedurze `ResizeMyWorld`, wywoływanej przez procedurę `ReshapeFunc` zarejestrowaneą przez `glutReshapeFunc`, zamiast w procedurze inicjalizacji.

Listing 7.6. Procedury `InitModelMatrix` i `InitViewMatrix`

C

```
1: void InitModelMatrix ( void )
2: {
3:   GLfloat m[16];
4:
5:   M4x4Identf ( m );
6:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
7:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[0], 16*sizeof(GLfloat), m );
8:   ExitIfGLError ( "InitModelMatrix" );
9: } /*InitModelMatrix*/
10:
11: void InitViewMatrix ( void )
12: {
13:   GLfloat m[16];
14:
15:   M4x4Translatef ( m, 0.0, 0.0, -10.0 );
16:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
17:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1], 16*sizeof(GLfloat), m );
18:   ExitIfGLError ( "InitViewMatrix" );
19: } /*InitViewMatrix*/
```



Rysunek 7.2. Bryła widzenia w układzie obserwatora

Umieściliśmy obserwatora w odległości 10 od środka obiektu. Chcemy dobrać rzutowanie tak, aby kula o tym samym środku i promieniu $R = 1.1$ miała obraz o wysokości klatki zajmującej całe okno. Aby tak było, kąt dwuścienny[13] między płaszczyznami dolnej i górnej ściany ostrosłupa widzenia musi być równy

$$2\alpha = 2 \arcsin \frac{1.1}{10} \approx 0.22045.$$

[13] Miary kątów obliczam konsekwentnie w radianach.

Przyjąłem (arbitralnie) parametry ostrosłupa widzenia $n = 5$, $f = 15$, co daje zupełnie wystarczający zakres odległości dla naszego obiektu. Wtedy musi być[14]

$$t = -b = n \operatorname{tg} \alpha \approx 0.5533.$$

Parametry l i r należy obliczyć na podstawie wymiarów okna. Listing 7.7 przedstawia procedurę wykonującą obliczenia na podstawie powyższych rachunków; przyjęty współczynnik aspektu ekranu jest równy 1.[15]

Listing 7.7. Procedura `ResizeMyWorld`

C

```
1: void ResizeMyWorld ( int width, int height )
2: {
3:   GLfloat m[16];
4:   float lr;
5:
6:   glViewport ( 0, 0, width, height );      /* klatka jest całym oknem */
7:   lr = 0.5533*(float)width/(float)height;  /* przyjmujemy aspekt równy 1 */
8:   M4x4Frustumf ( m, NULL, -lr, lr, -0.5533, 0.5533, 5.0, 15.0 );
9:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
10:  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[2], 16*sizeof(GLfloat), m );
11:  ExitIfGLError ( "ResizeMyWorld" );
12: } /*ResizeMyWorld*/
```

Ustawianie klatki i macierzy rzutowania w procedurze `ResizeMyWorld` jest poprawnym rozwiązaniem, jeśli zawsze klatka jest całym oknem, w którym obraz jest wykonywany przy użyciu jednego tylko rzutu. Gdyby tak nie było (w bardziej skomplikowanej aplikacji), to odpowiednie dane trzeba umieszczać w kontekście OpenGL-a w procedurze `RedrawMyWorld` bezpośrednio przed rysowaniem obiektów w poszczególnych klatkach w oknie.

7.3. Rysowanie

Rysowanie całej sceny wykonuje procedura `RedrawMyWorld`, która wywołuje procedurę `DrawIcosahedron`, aby narysować dwudziestościan. Procedury te są przedstawione na listingach 7.8 i 7.9.

Program szaderów, skompilowany i złączony na początku działania aplikacji, procedura `glUseProgram` w linii 3 uaktywnia, tj. przygotowuje do pracy potok przetwarzania grafiki z jego szaderami. Procedura `glBindVertexArray` w linii 4 uaktywnia VAO z wierzchołkami dwudziestościanu, również przygotowany na początku działania aplikacji.

Dalsze działania zależą od wartości parametru `opt`, który wybiera rodzaj rysunku. Jeśli ma on wartość 0, to mają być narysowane wierzchołki, a dokładniej kropki w miejscach obrazów wierzchołków. Procedura `glPointSize` określa wielkość kwadratowej kropki (jej

[14] W tym obliczeniu dokładność do czterech cyfr dziesiętnych jest aż nadto wystarczająca.

[15] Dla ciekawości: gdyby okno, czyli klatka, miało proporcje szerokości i wysokości jak $3 : 2$, to przekątna przedniej ściany ostrosłupa widzenia miałaby długość $d \approx 1.995$. Wtedy $n/d \approx 2.506$, co odpowiada obiektywowi o długości ogniskowej 125 mm. W fotografii to jest obiektyw długoogniskowy, ale jeszcze nie teleobiektyw.

szerokość i wysokość w pikselach)[16]. Następnie jest wywołana procedura `glDrawArrays`; jej pierwszy parametr określa, że chcemy narysować kropki. Drugi parametr jest indeksem (w buforach zarejestrowanych w VAO) pierwszego wierzchołka do narysowania, a parametr trzeci określa liczbę tych wierzchołków. W tym przykładzie rysujemy wszystkie wierzchołki.

Listing 7.8. Procedura `DrawIcosahedron`

C

```
1: void DrawIcosahedron ( int opt )
2: {
3:   glUseProgram ( program_id );
4:   glBindVertexArray ( icos_vao );
5:   switch ( opt ) {
6: case 0:    /* wierzchołki */
7:     glPointSize ( 5.0 );
8:     glDrawArrays ( GL_POINTS, 0, 12 );
9:     break;
10: case 1:    /* krawędzie */
11:     glDrawElements ( GL_LINE_STRIP, 25,
12:                      GL_UNSIGNED_BYTE, (GLvoid*)0 );
13:     glDrawElements ( GL_LINE_STRIP, 3,
14:                      GL_UNSIGNED_BYTE, (GLvoid*)(25*sizeof(GLubyte)) );
15:     glDrawElements ( GL_LINES, 8,
16:                      GL_UNSIGNED_BYTE, (GLvoid*)(28*sizeof(GLubyte)) );
17:     break;
18: default:   /* ściany */
19:     glDrawElements ( GL_TRIANGLE_FAN, 7,
20:                      GL_UNSIGNED_BYTE, (GLvoid*)(36*sizeof(GLubyte)) );
21:     glDrawElements ( GL_TRIANGLE_FAN, 7,
22:                      GL_UNSIGNED_BYTE, (GLvoid*)(43*sizeof(GLubyte)) );
23:     glDrawElements ( GL_TRIANGLE_STRIP, 12,
24:                      GL_UNSIGNED_BYTE, (GLvoid*)(50*sizeof(GLubyte)) );
25:     break;
26:   }
27:   glBindVertexArray ( 0 );
28:   ExitIfGLError ( "DrawIcosahedron" );
29: } /*DrawIcosahedron*/
```

Jeśli `(opt == 1)`, to procedura ma narysować krawędzie dwudziestościanu. Indeksy końców tych krawędzi znajdują się w tablicy umieszczonej w buforze, którego identyfikator jest zapamiętany w zmiennej `icos_vbo[2]`; ponieważ ten bufor (IBO) został przywiązany do celu w czasie, gdy używany teraz VAO był aktywny, wywołanie procedury w linii 4 spowodowało ponowne przywiązanie go do celu celu `GL_ELEMENT_ARRAY_BUFFER` i można przystąpić do rysowania za pomocą procedury `glDrawElements`. Pierwsze dwa wywołania, z parametrem `GL_LINE_STRIP`, powodują rysowanie łamanych (odpowiednio z 25 i 3 wierzchołkami, czyli złożonych z 24 i 2 odcinków).

[16] Szader fragmentów może nadać kropce dowolny kształt, zakazując przypisywania nowego koloru wybranym pikselom tego kwadratu — służy do tego instrukcja `discard`.

Trzeci parametr określa typ zmiennych użyty do reprezentowania indeksów w tablicy. Parametr ten musi być równy GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT albo GL_UNSIGNED_INT, przy czym w naszym przypadku indeksy są reprezentowane przez liczby ośmiobitowe, czyli bajty. Parametr czwarty, zadeklarowany w nagłówku procedury jako wskaźnik z tych samych historycznych powodów, które wcześniej były podane dla procedury glVertexAttribPointer, jest przesunięciem (w bajtach) od początku tablicy do miejsca, z którego etap pobierania wierzchołków ma wziąć pierwszy indeks.

Procedura glDrawElements wywołana w liniach 15–16 z parametrem GL_LINES rysuje 4 oddzielne odcinki. Liczba tych odcinków jest wartością drugiego parametru podzieloną przez liczbę końców jednego odcinka.

Jeśli parametr opt ma wartość inną niż 0 lub 1, to mają być narysowane obrazy ścian trójkątnych, wypełnione odpowiednimi kolorami. W tym celu w kolejnych wywołaniach procedury glDrawElements są rysowane dwa wachlarze trójkątów (za to odpowiada parametr GL_TRIANGLE_FAN) oraz taśma trójkątowa (GL_TRIANGLE_STRIP). W każdym przypadku drugi parametr określa liczbę wierzchołków rysowanego prymitywu; w przypadku wachlarza lub taśmy liczba trójkątów jest o 2 mniejsza od wartości tego parametru.

Procedura RedrawMyWorld przedstawiona na listingu 7.9 jest wywoływana za każdym razem, gdy należy wykonać obraz w oknie aplikacji. Jej zadaniem jest przygotowanie tła, ustawienie opcji rysowania (w szczególności włączenie algorytmu widoczności), wywołanie procedur rysujących obiekty (w naszym przypadku jednej procedury) i pociągnięcie gotowego obrazu werniksem.

Listing 7.9. Procedura RedrawMyWorld

C

```
1: int option = 2;
2:
3: void RedrawMyWorld ( void )
4: {
5:   glClearColor ( 1.0, 1.0, 1.0, 1.0 );
6:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
7:   glEnable ( GL_DEPTH_TEST );
8:   DrawIcosahedron ( option );
9:   glFlush ();
10: } /*RedrawMyWorld*/
```

Procedura glClearColor ustawia kolor tła, na którym będzie wykonywany rysunek. Ponieważ na rysunku wydrukowanym na papierze jasne kropki na czarnym tle są słabo widoczne [17], parametry podane w linii 5 ustawiają białe tło.

Wszystkie piksele w buforze obrazu otrzymują kolor tła[18] przez wywołanie procedury glClear w linii 6. Jej parametr określa, że jednocześnie z buforem obrazu ma być skasowany bufor głębokości używany przez algorytm widoczności. Procedura glEnable w linii 7

[17] Poza tym, gdy pisałem skrypt, szkoda mi było tonera, a i farbą drukarską nie należy szafować.

[18] To jest chyba jedyna operacja rysowania, jaką w nowym OpenGL-u da się wykonać bez programu szaderów.

włącza algorytm widoczności, dzięki czemu obraz sceny składającej się z nieprzezroczystych obiektów da się oglądać.

Po narysowaniu wszystkich obiektów (czyli na razie jednego), należy wywołać procedurę `glFlush`. Informuje ona OpenGL-a, że na tym rysunku wszelkie obiekty już zostały przekazane do narysowania i nie należy czekać na dalsze. Powrót z procedury `glFlush` może nastąpić przed zakończeniem rysowania (pamiętajmy, że CPU i GPU działają równolegle), ale GPU dokończy rysowanie najszybciej, jak potrafi.

Istnieją sytuacje, w których należy poczekać na dokończenie rysowania, na przykład wtedy, gdy chcemy odczytać obraz z bufora obrazu i zapisać go w pliku. Wywołanie procedury `glFlush` wtedy nie wystarczy; zamiast niej trzeba wywołać procedurę `glFinish`, z której powrót nastąpi po zakończeniu rysowania.

Ostatnią instrukcją procedury `DisplayFunc` (listing 3.1) jest wywołanie procedury `glutSwapBuffers`. Powoduje ona, że po zakończeniu rysowania (nie wcześniej!) bufory obrazu zostaną zamienione. Na początku działania aplikacji zadeklarowane zostało podwójne buforowanie obrazu (listing 3.1, linia 56, parametr z maską bitową `GLUT_DOUBLE`): rysowanie odbywa się w buforze niewidocznym, a w tym czasie w oknie jest widoczna zawartość drugiego bufora. Po zakończeniu rysowania bufory są zamieniane, dzięki czemu w oknie nie bywa widoczny obraz niedokończony. W animacji bez podwójnego buforowania niedokończone obrazy objawiają się jako silne i nieprzyjemne dla użytkownika migotanie obrazu.

7.4. Interakcja

W pierwszej aplikacji pozwolimy użytkownikowi tylko na zatrzymanie programu i na przełączanie trzech rodzajów obrazów; wszystkie te polecenia można wydawać za pomocą klawiatury. Listing 7.10 przedstawia procedurę `KeyboardFunc` z listingu 3.1 uzupełnioną o wywołanie procedury `ProcessCharCommand` z części „graficznej” aplikacji — procedura ta zmienia sposób rysowania dwudziestościanu po napisaniu liter `S`, `K` i `W`.

Listing 7.10. Procedura `KeyboardFunc`

C

```
1: void KeyboardFunc ( unsigned char charcode, int x, int y )
2: {
3:   switch ( charcode ) {
4: case 0x1B:            /* klawisz Esc - zatrzymanie programu */
5:     Cleanup ();
6:     glutLeaveMainLoop ();
7:     break;
8: default:
9:     if ( ProcessCharCommand ( charcode ) )
10:      glutPostWindowRedisplay ( WindowHandle );
11:    break;
12:  }
13: } /*KeyboardFunc*/
```

Jeśli dana litera spowodowała zmianę sposobu rysowania, to procedura `ProcessCharCommand` przekazuje wartość niezerową. Wtedy procedura `KeyboardFunc` wywołuje procedurę `glutPostWindowRedisplay`, która spowoduje wywołanie procedury `DisplayFunc` i w oknie ukaże się nowy obraz. Procedura `ProcessCharCommand` tak jak pozostałe procedury „graficzne" „nie wie", że jest częścią aplikacji FreeGLUT-a.

Listing 7.11. Procedura `ProcessCharCommand`

C

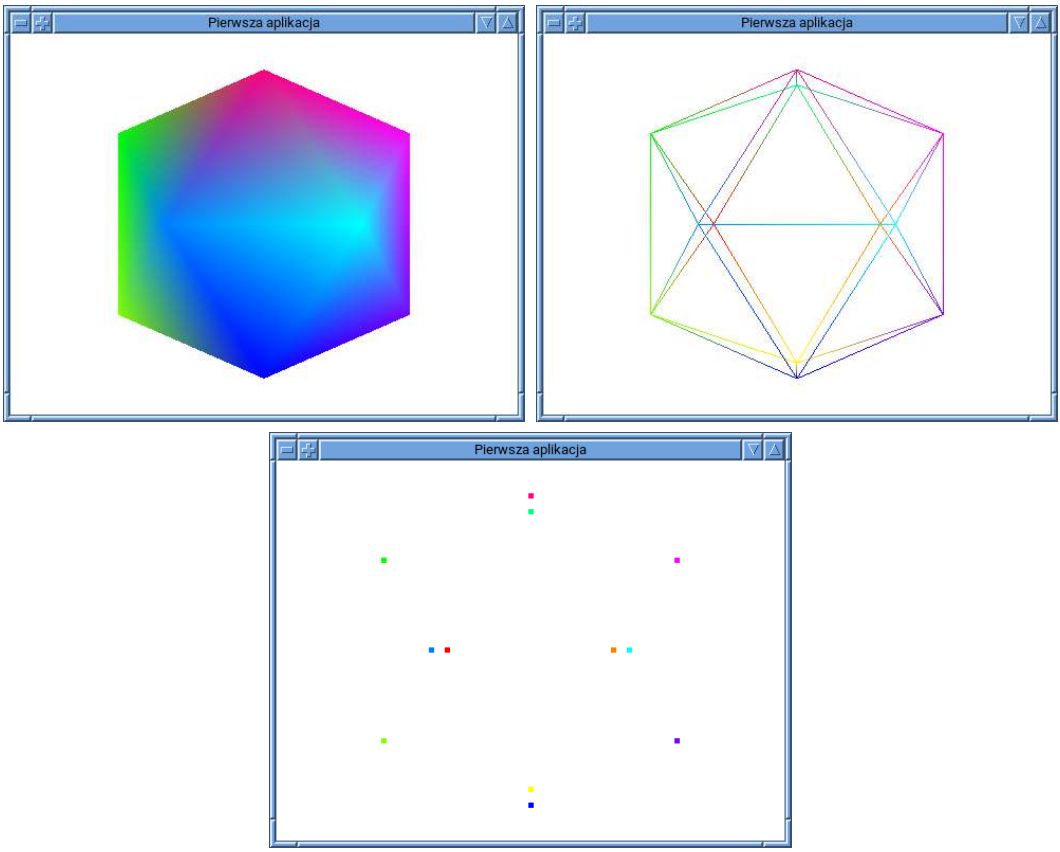
```
1: char ProcessCharCommand ( char charcode )
2: {
3:   int oldoption;
4:
5:   oldoption = option;
6:   switch ( toupper ( charcode ) ) {
7: case 'W': option = 0;  break;  /* przełączamy na wierzchołki */
8: case 'K': option = 1;  break;  /* przełączamy na krawędzie */
9: case 'S': option = 2;  break;  /* przełączamy na ściany */
10:  default: return false;       /* ignorujemy wszystkie inne znaki */
11:   }
12:   return option != oldoption;
13: } /*ProcessCharCommand*/
```

7.5. Sprzątanie

Aplikacja, którą napisaliśmy z wykorzystaniem szkieletu z listingu 3.1, przed wyjściem z pętli komunikatów FreeGLUT-a wywoła jeszcze procedurę `Cleanup`, a ta wywoła procedurę `DeleteMyWorld`. W zasadzie można by nie zawracać sobie głowy jej pisaniem; system operacyjny w komputerze zawsze dokładnie sprząta i szoruje podłogę po procesie, który zakończył działanie. Ale dobre wychowanie wymaga, aby samemu sprzątać po sobie, co *nie jest* tylko pustym gestem. Bardziej skomplikowane aplikacje, takie na miarę naszych ambicji, będą tworzyć wiele obiektów zajmujących pamięć CPU i GPU. Wiele z nich będzie likwidowanych w trakcie działania aplikacji po to, aby zrobić miejsce dla następnych obiektów. Dlatego warto już od małego[19] przyzwyczajać się do takiego programowania, aby *wszystkie* zasoby zajmowane przez program były zwalniane natychmiast, gdy przestają być potrzebne; jeśli nie wcześniej, to w chwili zatrzymania programu.

Procedura sprzątająca pierwszej aplikacji jest pokazana na listingu 7.12. Wywołanie procedury `glUseProgram` z parametrem `0` odaktywnia program (liczba `0` pełni obowiązki identyfikatora programu pustego). W linii `4` program szaderów jest kasowany. Następnie zwalniany jest UBO, tj. bufor, w którym przechowywane były zmienne jednolite (macierze przekształceń) oraz VAO i bufory z tablicami wierzchołków i indeksów do wierzchołków.

[19] projektu



Rysunek 7.3. Okno pierwszej aplikacji z obrazami dwudziestościanu

Listing 7.12. Procedura `DeleteMyWorld`

C

```
1: void DeleteMyWorld ( void )
2: {
3:   glUseProgram ( 0 );
4:   glDeleteProgram ( program_id );
5:   glDeleteBuffers ( 1, &trbuf );
6:   glDeleteVertexArrays ( 1, &icos_vao );
7:   glDeleteBuffers ( 3, icos_vbo );
8:   ExitIfGLError ( "DeleteMyWorld" );
9: } /*DeleteMyWorld*/
```

Sprzątanie kontekstu OpenGL-a, tj. likwidacja szaderów i buforów ma sens *przed* usunięciem okna, ponieważ procedura `glutDestroyWindow` unicestwia kontekst stworzony dla tego okna (razem z ewentualnym śmietnikiem w nim zawartym[20]). Dlatego ostatnie sprawdzenie, czy wystąpił błąd OpenGL-a, następuje, gdy kontekst jeszcze istnieje. Podobnie, pro-

[20] Proszę jednak nie traktować tego jako wymówki od sprzątania.

cedura `glutLeaveMainLoop` powoduje powrót z procedury `glutMainLoop` poprzedzony likwidacją wszystkich okien (razem z ich kontekstami), zatem po zakończeniu jej działania nie ma już czego sprzątać. Aplikacja *może* wznowić działanie, ale musi w tym celu zacząć wszystko od nowa, tj. od ponownego wywołania procedury `glutInit`.

7.6. Uzupełnienia

7.6.1. Tryby pracy potoku przetwarzania grafiki

Pierwszy parametr procedur `glDrawArrays` i `glDrawElements` określa **tryb** pracy potoku przetwarzania grafiki. W pierwszej aplikacji wykorzystane są tryby rysowania punktów (`GL_POINTS`), osobnych odcinków (`GL_LINES`), łamanej (`GL_LINE_STRIP`), taśmy trójkątowej (`GL_TRIANGLE_STRIP`) i wachlarza trójkątów (`GL_TRIANGLE_FAN`). Pozostałe dopuszczalne wartości tego parametru wybierają następujące tryby:

`GL_LINE_LOOP` — rysowanie łamanej zamkniętej; w dodatku do odcinków łączących kolejne wierzchołki jest rysowany odcinek między wierzchołkiem ostatnim a pierwszym.

`GL_TRIANGLES` — rysowanie osobnych trójkątów; ciąg wierzchołków jest dzielony na trójki, każda z nich określa jeden trójkąt.

`GL_PATCHES` — rysowanie tzw. **płatów** z wykorzystaniem szaderów rozdrabniania, przedstawione w rozdziałach 12 i 15.

`GL_LINES_ADJACENCY`, `GL_LINE_STRIP_ADJACENCY`, `GL_TRIANGLES_ADJACENCY`, `GL_TRIANGLE_STRIP_ADJACENCY` — rysowanie odcinków lub trójkątów, przy czym szader geometrii otrzymuje dane opisujące odcinki lub trójkąty sąsiadujące z przetwarzanym obiektem. O tych trybach będzie mowa w p. 12.4.3.

7.6.2. Pomijanie ścian odwróconych tyłem

Dla **obiektów z zamkniętą objętością**, czyli w OpenGL-u dla zbudowanych z trójkątów powierzchni brył, istotne znaczenie ma **orientacja** brzegu. Jedna strona trójkąta może być widziana przez obserwatora *znajdującego się na zewnątrz* bryły, a druga strona jest „wewnętrzna". W szczególności trójkąty „odwrócone tyłem" do obserwatora można podczas rysowania pominąć, ponieważ są one zasłonięte przez inne trójkąty, „odwrócone przodem" do obserwatora. Dla skomplikowanych scen daje to znaczną oszczędność czasu rysowania.

Trójkąt, którego obraz nie jest odcinkiem, może być zorientowany względem obserwatora na dwa sposoby: przechodząc przez obrazy jego wierzchołków zgodnie z podaną kolejnością obchodzimy obraz jego środka ciężkości w kierunku zgodnym z ruchem wskazówek zegara (*clockwise*) albo przeciwnym (*counterclockwise*).

Warto zadbać o to, aby wszystkie trójkąty powierzchni bryły miały **zgodną orientację**. Niech $\boldsymbol{p}_i, \boldsymbol{p}_j, \boldsymbol{p}_k$ oznacza wierzchołki trójkąta (w $\mathbb{R}^3$), podane w tej kolejności, i niech $\boldsymbol{v}_{ij} = \boldsymbol{p}_j - \boldsymbol{p}_i$, $\boldsymbol{v}_{ik} = \boldsymbol{p}_k - \boldsymbol{p}_i$ oraz $\boldsymbol{n}_{ijk} = \boldsymbol{v}_{ij} \wedge \boldsymbol{v}_{ik}$. Wektor $\boldsymbol{n}_{ijk}$ jest prostopadły do płaszczyzny trójkąta. Orientacja trójkątów jest zgodna, jeśli dla wszystkich trójkątów tak obliczone wektory są zwrócone na zewnątrz bryły albo dla wszystkich trójkątów są zwrócone do wewnątrz.

Aby pominąć ściany odwrócone tyłem, należy poprzedzić rysowanie ścian bryły wywołaniem procedur

```
glEnable ( GL_CULL_FACE );
glCullFace ( GL_FRONT );
glFrontFace ( GL_CW );
```

przy czym procedura `glCullFace` może mieć też parametr `GL_BACK`, jeśli chcemy rysować tylko ściany „odwrócone tyłem", bo na przykład obserwator znalazł się wewnątrz bryły. Parametr `GL_CW` procedury `glFrontFace` określa, że przodem są odwrócone ściany, których obrazy są zorientowane zgodnie z ruchem wskazówek zegara, a `GL_CCW` deklaruje orientację przeciwną. Pamiętajmy, aby rysując następny obiekt, jeśli jego brzeg nie jest odpowiednio zorientowany, wyłączyć ten mechanizm za pomocą instrukcji

```
glDisable ( GL_CULL_FACE );
```

OpenGL zapewnia, że wszystkie trójkąty w wachlarzu lub taśmie są zgodnie zorientowane, wystarczy zatem zadbać tylko o poprawną orientację ich pierwszego trójkąta — reprezentacja dwudziestościanu w pierwszej aplikacji ten warunek spełnia.

7.7. Ćwiczenia

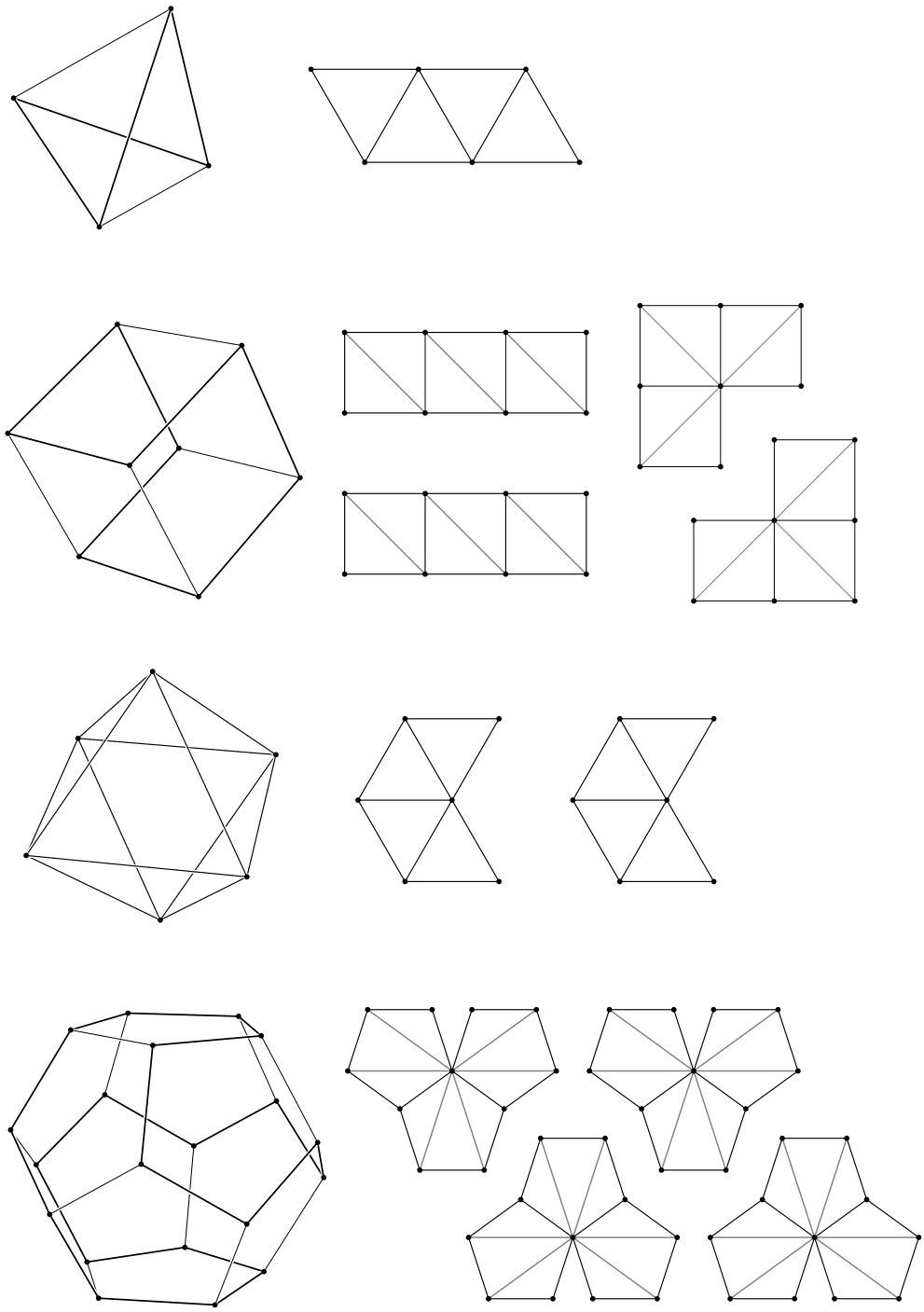
1. Dopisz do aplikacji (w procedurze `DrawIcosahedron`) instrukcje pomijania ścian o jednej, a potem o drugiej orientacji, skompiluj i obejrzyj skutki. Skonfrontuj wyniki obserwacji z wiadomościami podanymi w podrozdziale 5.6.
2. Napisz procedury, które tworzą w pamięci GPU reprezentacje czworościanu foremnego, sześcianu i ośmiościanu foremnego i procedury rysujące obrazy tych brył i dołącz je do aplikacji. Kolory wierzchołków wybierz dowolnie. Rozbuduj procedurę `KeyboardFunc`, aby móc przełączać wyświetlane bryły.

   Krawędzie czworościanu można reprezentować jako łamaną zamkniętą i dwa osobne odcinki, a z jego ścian można zrobić jedną taśmę trójkątową.

   Z krawędzi sześcianu można zrobić jedną łamaną zamkniętą i cztery osobne odcinki, a ze ścian, po podzieleniu ich na trójkąty, można zrobić dwie taśmy trójkątowe albo dwa wachlarze.

   Krawędzie ośmiościanu można ustawić w takiej kolejności, aby powstała jedna łamana zamknięta; ściany można narysować jako dwa wachlarze trójkątów. Wykonując to ćwiczenie, postaraj się o zapewnienie zgodnej orientacji wszystkich ścian wielościanów.

   Wskazówka: Weź najpierw kolorowe kredki i ponumeruj (dowolnie) wierzchołki brył na rysunku 7.4, a potem wierzchołki ścian w taśmach i wachlarzach narysowanych obok.
3. Napisz i uruchom procedury tworzenia reprezentacji i rysowania dwunastościanu foremnego. Z jego krawędzi można utworzyć łamaną zamkniętą przechodzącą przez wszyst-

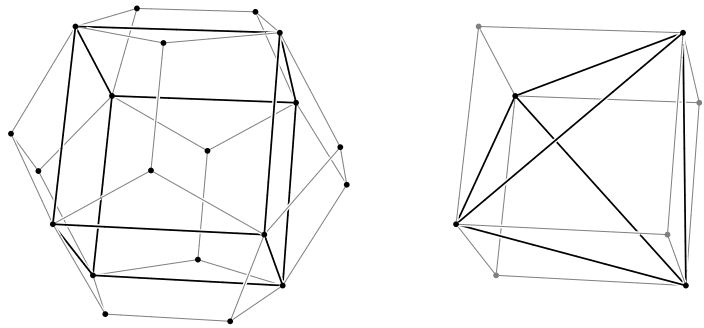


Rysunek 7.4. Bryły foremne: wierzchołki, krawędzie i ściany

kie wierzchołki (cykl Hamiltona) i wtedy pozostaje 10 krawędzi do narysowania osobno. Ściany pięciokątne trzeba podzielić na trójkąty (każdą na 3). Można zgrupować po trzy ściany i każdą taką trójkę reprezentować w postaci wachlarza z 9 trójkątów — wtedy narysowanie ścian dwunastościanu wymaga wyświetlenia czterech takich wachlarzy.

Wskazówka: Dwunastościan jest bryłą dualną do dwudziestościanu. Jego wierzchołki można znaleźć w środkach ciężkości trójkątnych ścian dwudziestościanu — do obliczenia ich współrzędnych można wykorzystać liczby A, B, użyte w procedurze na listingu 7.5.

4. Narysuj szkielet krawędziowy dwudziestościanu (lub innej bryły) i obrazy wierzchołków w postaci kropek. Dla wszystkich krawędzi użyj *jednego* koloru, natomiast wierzchołki narysuj w różnych kolorach. Kolor krawędzi podaj w zmiennej statycznej i przed ich rysowaniem wyłącz tablicę kolorów w VAO za pomocą `glDisableVertexAttribArray` (a przed rysowaniem wierzchołków ją włącz).
5. Korzystając z tablicy zawierającej wierzchołki sześcianu, narysuj dwa czworościany wpisane w ten sześcian — ich suma jest bryłą wielościenną zwaną *stella octangula*.



Rysunek 7.5. Sześcian wpisany w dwunastościan i czworościan wpisany w sześcian

6. W dwunastościan foremny można wpisać sześcian tak, aby jego wierzchołki były niektórymi wierzchołkami dwunastościanu (rys. 7.5). Narysuj na jednym obrazie wszystkie takie sześciany (jest ich pięć) oraz (na innym obrazie) wszystkie czworościany wpisane w te sześciany.

   Wskazówka: Druga część tego ćwiczenia jest znacznie bardziej zaawansowana niż się to początkowo wydaje — zobacz podrozdział 7.8.

7. Utwórz bufor z tablicą z liczbami

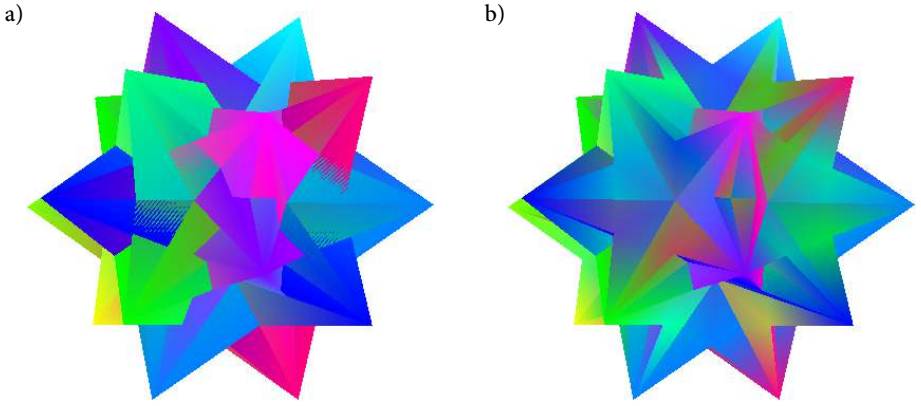
   ```
   0, 1, 9, 8, 3, 7, 11, 5, 10, 9, 2, 8, 7, 4, 11, 10, 1, 6, 7, 4, 5, 10,
   1, 2, 8, 3, 4, 11, 5, 1, 9, 2, 3, 7, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11.
   ```

   Używając go jako tablicy indeksów wierzchołków razem z VAO zawierającym wierzchołki dwudziestościanu, narysuj dwa wachlarze trójkątów z 17 wierzchołkami (od miejsc 0 i 17) i dwa wachlarze trójkątów z 5 wierzchołkami (od miejsc 34 i 39). Otrzymasz obraz bryły zwanej **dwunastościanem wielkim**.

8. Jeśli użytkownik, zmieniając wymiary okna, sprawi, że Jego Wysokość będzie znacznie większa niż szerokość, to nie cały obiekt zmieści się na obrazie. Napraw to, modyfikując procedurę `ReshapeFunc` (listing 7.7): jeśli wysokość jest większa niż szerokość, to kąt dwuścienny między płaszczyznami lewej i prawej ściany bryły widzenia ma być równy $2\alpha = 2 \arcsin \frac{1.1}{10}$, a kąt dwuścienny między płaszczyznami górnej i dolnej ściany odpowiednio większy.

7.8. *Uzupełnienia — błędy rozstrzygania widoczności

Kto wykonał ćwiczenie 6, mógł być niemile zaskoczony wyglądem obrazu wszystkich czworościanów wpisanych w dwunastościan foremny (rys. 7.6a). Każda ściana tych czworościanów ma (sześciokątną) część wspólną ze ścianą innego czworościanu, wskutek czego piksele mające kolor jednej ze ścian są brzydko „wymieszane" z pikselami w kolorze drugiej ściany. Co więcej, zmieniając położenie obserwatora (co oprogramujemy w następnym rozdziale), zobaczymy nieprzyjemne migotanie pikseli. Mniejsza o wrażenia estetyczne; mamy tu okazję do wglądu w działanie algorytmu widoczności i skwapliwie z niej skorzystamy.



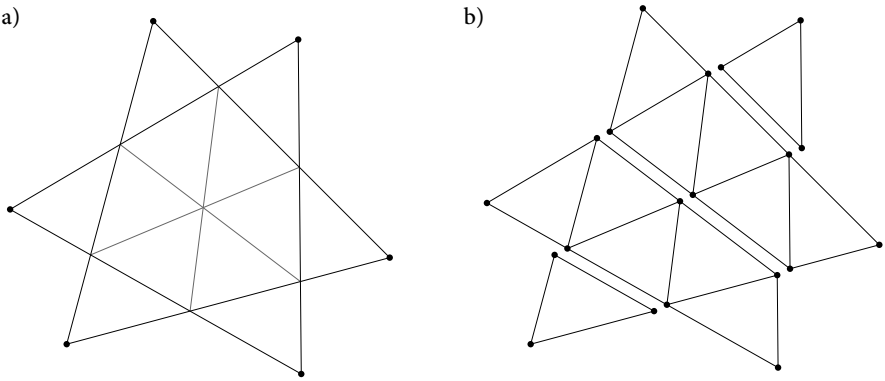
Rysunek 7.6. Obrazy czworościanów wpisanych w dwunastościan

Rozważany tu efekt jest skutkiem błędów reprezentacji wierzchołków trójkątów i błędów zaokrągleń powstających podczas rasteryzacji trójkątów. Współrzędne punktów są reprezentowane w postaci zmiennopozycyjnej, a więc muszą być odpowiednio zaokrąglone. Dwie różne trójki punktów w przestrzeni po takim zaokrągleniu zazwyczaj wyznaczają dwie *różne* płaszczyzny, równoległe albo przecinające się. Gdyby przekształcanie do układu kostki standardowej i rasteryzacja odbywały się bez dalszych błędów zaokrągleń, to w pierwszym przypadku jeden z trójkątów zasłaniałby cały sześciokątny fragment drugiego, a w drugim przypadku moglibyśmy zobaczyć odcinek prostej wspólnej płaszczyzn — granicę obszarów wypełnionych kolorami jednego i drugiego trójkąta[21].

[21] co wyglądałoby lepiej, ale też niezbyt dobrze

Obliczona podczas rasteryzacji trójkąta głębokość punktu odpowiadającego pikselowi (tj. liczba $\zeta = \frac{1}{2}(1+z)$ otrzymana na podstawie współrzędnej z punktu w układzie kostki standardowej) jest obarczona błędem wynikającym z zaokrągleń, którego wielkość (w tym znak) bywa różna dla różnych pikseli. Po narysowaniu pierwszego trójkąta głębokości jego widocznych pikseli są zapamiętane w buforze głębokości; rysując drugi trójkąt, GPU porównuje głębokości pikseli z tym, co w buforze znalazło się wcześniej i nadaje nowy kolor pikselowi (oraz uaktualnia zawartość bufora), jeśli nowa głębokość jest mniejsza. Skutki błędów zaokrągleń są widoczne na rysunku.

Jeśli rysowane trójkąty, mające wielokątną część wspólną, mają różne kolory, to nie ma żadnego łatwego sposobu naprawienia tej sytuacji. Jedyne, co można wtedy zrobić, to podzielić trójkąty na ich część wspólną i pozostałości, dzięki czemu kolor każdego punktu części wspólnej trójkątów da się określić jednoznacznie.



Rysunek 7.7. Dwa trójkąty i ich podział

Rysunek 7.7 przedstawia dwie ściany czworościanów mające wspólny sześciokąt oraz schemat podziału tych ścian na taśmy trójkątowe i osobne trójkąty, zastosowany w programie, który wykonał obraz pokazany na rysunku 7.6b. Każda krawędź czworościanu (a jest ich 60) przecina się z czterema innymi krawędziami, zatem jest do znalezienia 120 punktów przecięcia krawędzi. Wprowadziłem też dodatkowy wierzchołek we wspólnym środku ciężkości każdych dwóch nakładających się trójkątów, wskutek czego tablica wierzchołków wydłużyła się z 20 do 160 wierzchołków. Oczywiście, reprezentacji obiektu nie kodowałem ręcznie. Zamiast tego napisałem program, który na podstawie tablicy wierzchołków dwunastościanu skonstruował dodatkowe wierzchołki i zapisał w języku C deklaracje odpowiednich tablic, zawierających współrzędne położenia i koloru wierzchołków oraz indeksy wierzchołków trójkątów (pojedynczych i w taśmach), do pliku, który następnie dołączyłem do aplikacji. Kolory dodatkowych wierzchołków zostały otrzymane przez interpolację i uśrednianie danych kolorów wierzchołków dwunastościanu.

Choć to w zasadzie nie jest ćwiczenie z OpenGL-a, tylko z programowania prostych konstrukcji geometrycznych, polecam napisanie takiego programu w ramach indywidualnych rozszerzeń kursu. Zrobiwszy to samemu, zapewniam, że było warto.

8

Aplikacja pierwsza A

Do pierwszej aplikacji dodamy większe możliwości interakcji. Będziemy chcieli obracać obserwatora wokół wyświetlanej bryły i będziemy chcieli wprawiać bryłę w „samoczynny" ruch obrotowy. Zatem, nie będzie tu nowych konstrukcji OpenGL-a ani GLSL-a, ale wykorzystamy więcej możliwości FreeGLUT-a.

8.1. Składanie obrotów

Dowolny obrót w $\mathbb{R}^3$ może być reprezentowany za pomocą wektora jednostkowego $\boldsymbol{\nu}$ wyznaczającego kierunek osi obrotu i liczby φ, która jest miarą kąta obrotu. Trzeba będzie tak reprezentowane obroty składać — mając dane reprezentacje dwóch kolejno wykonanych obrotów w postaci par $(\boldsymbol{\nu}_1, \varphi_1)$ i $(\boldsymbol{\nu}_2, \varphi_2)$, potrzebujemy obliczyć parę $(\boldsymbol{\nu}, \varphi)$, która reprezentuje złożenie tych obrotów[1]. Do tego obliczenia potrzebna będzie procedura `V3CompRotationsf` pokazana na listingu 8.1, którą najwygodniej jest dodać do pliku `utilities.c` (a jej prototyp do `utilities.h`). Macierz obrotu na podstawie wektora i kąta skonstruuje procedura `M4x4RotateVf` podana na listingu 5.1.

Należy znaleźć parę $(\boldsymbol{\nu}, \varphi)$ złożoną z wektora jednostkowego $\boldsymbol{\nu}$ i liczby φ, reprezentującą obrót przestrzeni $\mathbb{R}^3$ będący złożeniem dwóch obrotów reprezentowanych w ten sam sposób; obrotu wokół osi o kierunku $\boldsymbol{\nu}_1$ o kąt φ_1, *po którym* następuje obrót o kąt φ_2 wokół osi o kierunku $\boldsymbol{\nu}_2$. Wektory $\boldsymbol{\nu}_1, \boldsymbol{\nu}_2 \in \mathbb{R}^3$, których współrzędne są podane w tablicach `v1` i `v2`, muszą być jednostkowe. Aby znaleźć wektor $\boldsymbol{\nu}$ i liczbę φ, należy najpierw obliczyć wektor

$$\boldsymbol{w} = \boldsymbol{\nu}_2 \sin\frac{\varphi_2}{2}\cos\frac{\varphi_1}{2} + \boldsymbol{\nu}_1 \sin\frac{\varphi_1}{2}\cos\frac{\varphi_2}{2} + \boldsymbol{\nu}_2 \wedge \boldsymbol{\nu}_1 \sin\frac{\varphi_1}{2}\sin\frac{\varphi_2}{2},$$

[1] Obroty można reprezentować za pomocą macierzy ortogonalnych i wtedy złożenie obrotów jest reprezentowane przez iloczyn tych macierzy. Ale iloczyn *wielu* takich macierzy obliczony przy użyciu arytmetyki zmiennopozycyjnej może być (wskutek kumulacji błędów zaokrągleń) złym przybliżeniem macierzy ortogonalnej, a to spowodowałoby zniekształcenia obiektów. Reprezentacja obrotu za pomocą wektora $\boldsymbol{\nu}$ i kąta φ jest pod tym względem lepsza, ponieważ kumulacja błędów powoduje niedokładne wyznaczenie wektora i kąta, które *zawsze* określają *jakiś* obrót — macierz tego obrotu będzie obliczona na ich podstawie całkiem dokładnie.

a następnie użyć wzorów

$$\boldsymbol{v} = \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}, \quad \cos\frac{\varphi}{2} = \cos\frac{\varphi_1}{2}\cos\frac{\varphi_2}{2} - \langle \boldsymbol{v}_1, \boldsymbol{v}_2\rangle \sin\frac{\varphi_1}{2}\sin\frac{\varphi_2}{2}, \quad \sin\frac{\varphi}{2} = \|\boldsymbol{w}\|.$$

Wzory te natychmiast wynikają z przedstawionej w dodatku A kwaternionowej reprezentacji obrotów w przestrzeni trójwymiarowej.

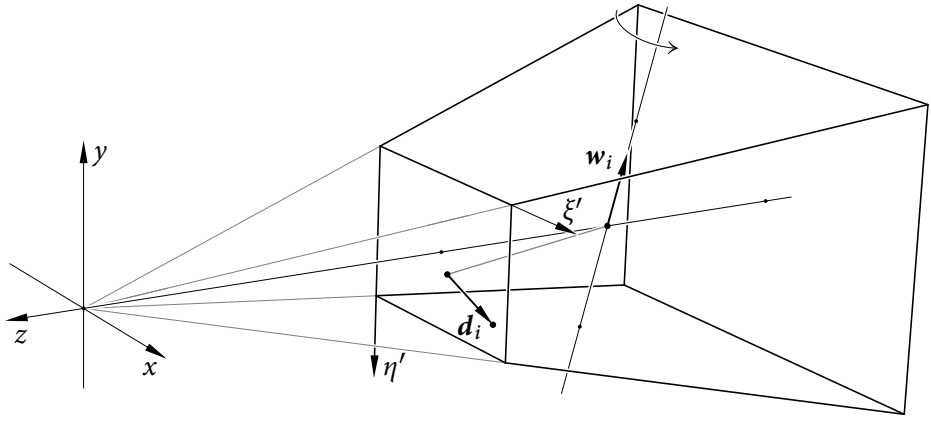
Listing 8.1. Procedura `V3CompRotationsf`

C

```
void V3CompRotationsf ( GLfloat v[3], double *phi,
                        const GLfloat v2[3], double phi2,
                        const GLfloat v1[3], double phi1 )
{
  float s2, c2, s1, c1, s, c, s2c1, s1c2, s2s1, v2v1, v2xv1[3];

  s2 = sin ( 0.5*phi2 );  c2 = cos ( 0.5*phi2 );
  s1 = sin ( 0.5*phi1 );  c1 = cos ( 0.5*phi1 );
  s2c1 = s2*c1;  s1c2 = s1*c2;  s2s1 = s2*s1;
  v2v1 = V3DotProductf ( v2, v1 );
  V3CrossProductf ( v2xv1, v2, v1 );
  c = c2*c1 - v2v1*s2s1;
  v[0] = v2[0]*s2c1 + v1[0]*s1c2 + v2xv1[0]*s2s1;
  v[1] = v2[1]*s2c1 + v1[1]*s1c2 + v2xv1[1]*s2s1;
  v[2] = v2[2]*s2c1 + v1[2]*s1c2 + v2xv1[2]*s2s1;
  s = sqrt ( V3DotProductf ( v, v ) );
  if ( s > 0.0 ) {
    v[0] /= s, v[1] /= s, v[2] /= s;
    *phi = 2.0*atan2 ( s, c );
  }
  else
    v[0] = 1.0, v[1] = v[2] = *phi = 0.0;
} /*V3CompRotationsf*/
```

8.2. Obracanie obserwatora wokół obiektu

Chcemy zrealizować następujący scenariusz interakcji: użytkownik naciska przycisk (np. lewy) myszy, a następnie przesuwa mysz, powodując przemieszczanie kursora w oknie. Spowoduje to wysłanie do aplikacji serii komunikatów o przesunięciu kursora. W aplikacji FreeGLUT-a przekazanie każdego takiego komunikatu polega na wywołaniu procedury zarejestrowanej przez `glutMotionFunc` (procedury `MotionFunc` na listingu 3.1). Przemieszczenie kursora od poprzedniej pozycji jest opisane przez pewien wektor równoległy do płaszczyzny obrazu, czyli rzutni. Chcemy, aby aplikacja obróciła obserwatora o pewien kąt wokół osi równoległej do rzutni i prostopadłej do tego wektora, po czym wykonała obraz dla nowego położenia obserwatora. Po zwolnieniu przycisku obracanie ma się zakończyć.



Rysunek 8.1. Przemieszczenie kursora i obrót w układzie obserwatora

Procedura `MouseFunc` wywołana po naciśnięciu przycisku musi zapamiętać współrzędne ξ'_0, η'_0 położenia kursora w oknie[2] i wprowadzić aplikację w tryb obracania obserwatora; procedura `MotionFunc` wywołana z parametrami x = ξ'_i i y = η'_i, korzystając z zapamiętanych współrzędnych ξ'_{i-1}, η'_{i-1} poprzedniego położenia kursora, musi wyznaczyć odpowiedni obrót, a potem zapamiętać liczby ξ'_i, η'_i na użytek obsługi następnego komunikatu.

Wektor przemieszczenia kursora w oknie (w układzie okna OpenGL-a) ma współrzędne $(\xi'_i - \xi'_{i-1})$ i $(\eta'_{i-1} - \eta'_i)$. W układzie współrzędnych obserwatora (tym, w którym jest opisana bryła widzenia) wektor ten ma kierunek i zwrot wektora

$$\boldsymbol{d}_i = c \begin{bmatrix} (\xi'_i - \xi'_{i-1})(r-l)/w \\ (\eta'_{i-1} - \eta'_i)(t-b)/h \\ 0 \end{bmatrix},$$

gdzie l, r, b, t to parametry określające bryłę widzenia (zobacz rozdział 6), liczby w i h to szerokość i wysokość klatki (okna) w pikselach, a c jest pewną liczbą dodatnią; ponieważ tu interesuje nas tylko kierunek i zwrot wektora $\boldsymbol{d}_i$ (który jest potrzebny do wyznaczenia kierunku osi obrotu), wartość bezwzględna c jest nieistotna[3].

Oś obrotu ma być równoległa do rzutni i prostopadła do wektora $\boldsymbol{d}_i$ (rys. 8.1); stąd możemy przyjąć, że ma ona kierunek wektora

$$\boldsymbol{w}_i = \begin{bmatrix} (\eta'_i - \eta'_{i-1})(t-b)/h \\ (\xi'_i - \xi'_{i-1})(r-l)/w \\ 0 \end{bmatrix},$$

który natychmiast podzielimy przez jego długość, aby otrzymać wektor jednostkowy $\boldsymbol{v}_i$.

[2] Biblioteki FreeGLUT i GLFW oraz systemy X Window i Windows używają układu z osią η' skierowaną do dołu. Oś η w układzie OpenGL-a jest skierowana do góry, zobacz rozdział 6.

[3] Nie jest zresztą oczywiste, jak należałoby ją zdefiniować dla rzutowania perspektywicznego.

Obrót układu obserwatora od położenia wyjściowego (w którym układ ten jest tylko przesunięty, ale nie obrócony względem układu świata) do położenia bieżącego będziemy reprezentować za pomocą pary $(\boldsymbol{z}_{i-1}, \varphi_{i-1})$. Obrót do nowego położenia jest złożeniem tego obrotu z obrotem reprezentowanym przez parę $(\boldsymbol{v}_i, \alpha)$, przy czym wbrew pozorom nie jest złym pomysłem, aby miara kąta tego obrotu była stała w każdym kroku; z mojego doświadczenia wynika, że dobrym wyborem jest przyjęcie $\alpha = 0.052359878 \approx 3^\circ$ (kto zechce, zmieni to sobie). Należy zatem obliczyć i zapamiętać odpowiedni wektor $\boldsymbol{z}_i$ i kąt φ_i.

Listing 8.2 przedstawia deklaracje zmiennych i procedury „graficzne”[4], które trzeba dodać do aplikacji opisanej w poprzednim rozdziale albo zmodyfikować, aby można było obracać obserwatora względem obiektu. Procedura `ResizeMyWorld` ma dodatkowe zadanie zapamiętania w zmiennych `win_width` i `win_height` wymiarów okna oraz przypisania zmiennym `left`, `right`, `bottom`, `top`, `near` i `far` obliczonych parametrów bryły widzenia.

W zmiennych `viewer_rvec` i `viewer_rangle` jest przechowywana reprezentacja obrotu obserwatora wokół obiektu, składająca się z wektora $\boldsymbol{z}_i$ i kąta φ_i. Wartości początkowe reprezentują obrót o kąt 0, czyli przekształcenie tożsamościowe[5].

Wreszcie, w zmiennej `viewer_pos0` mamy położenie obserwatora w obróconym układzie współrzędnych, niezmienne w tej aplikacji. Przejście od układu świata do układu obserwatora jest złożeniem obrotu reprezentowanego przez parę $(\boldsymbol{z}_i, \varphi_i)$ i przesunięcia. Jednak lepiej, aby współrzędne położenia obserwatora były przechowywane w zadeklarowanej zmiennej, a nie twardo zakodowane w treści procedury obliczającej macierz tego przejścia.

Procedura `InitViewMatrix` została zmieniona w ten sposób, że początkowe przekształcenie jest konstruowane na podstawie wartości zmiennej `viewer_pos0` (porównaj z listingiem 7.6); ta zmiana jest kosmetyczna.

Nowa jest procedura `RotateViewer`, której parametry `delta_xi` $= \xi'_i - \xi'_{i-1}$ oraz `delta_eta` $= \eta'_i - \eta'_{i-1}$ podają współrzędne wektora przemieszczenia kursora w oknie[6]. Działanie tej procedury szczegółowo wyjaśnię, bo jeszcze ważniejsze od tego, aby program działał, jest to, aby jego autor[7] rozumiał, co się tam dzieje. Przede wszystkim każda macierz nieosobliwa reprezentuje co najmniej *dwa* przekształcenia, albo też wynik mnożenia wektora przez macierz można interpretować na co najmniej dwa sposoby[8]. Pierwsza interpretacja jest taka, że mnożąc macierz przez wektor współrzędnych (jednorodnych) punktu, otrzymujemy współrzędne *nowego punktu*. Jeśli na przykład przekształcenie jest przesunięciem, to nowy punkt jest przesunięty o ustalony wektor $\boldsymbol{t}$ względem punktu danego.

W drugiej (dualnej) interpretacji wynik mnożenia macierzy przez wektor jest wektorem współrzędnych tego samego punktu *w nowym układzie współrzędnych*. Przesunięciu punktu o wektor $\boldsymbol{t}$ odpowiada przejście do układu przesuniętego względem układu wyjściowego

[4] tj. niezależne od środowiska FreeGLUT

[5] Oś takiego obrotu ma nieokreślony kierunek, zatem liczby obecne początkowo w tablicy `viewer_rvec` mogą być dowolne, byleby suma ich kwadratów była równa 1.

[6] Parametry tej procedury są typu `double`, bo ten typ jest używany do podawania współrzędnych położenia kursora w bibliotece GLFW; koszt tego rozwiązania jest znikomy, więc można go ponieść.

[7] czyli niestety ja

[8] Była o tym mowa w rozdziale 5. Trzecia interpretacja, tu nieistotna, to przejścia między współrzędnymi kartezjańskimi a barycentrycznymi.

Listing 8.2. Dodatkowe zmienne i procedury w części „graficznej” aplikacji pierwszej A

C

```
int          win_width, win_height;
float        left, right, bottom, top, near, far;
float        viewer_rvec[3] = {1.0,0.0,0.0};
double       viewer_rangle = 0.0;
const float viewer_pos0[4] = {0.0,0.0,10.0,1.0};

void ResizeMyWorld ( int width, int height )
{
  GLfloat m[16];
  float   lr;

  glViewport ( 0, 0, win_width = width, win_height = height );
  lr = 0.5533*(float)width/(float)height;
  M4x4Frustumf ( m, NULL, left = -lr, right = lr,
                  bottom = -0.5533, top = 0.5533, near = 5.0, far = 15.0 );
  glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[2], 16*sizeof(GLfloat), m );
  ExitIfGLError ( "ResizeMyWorld" );
} /*ResizeMyWorld*/

void InitViewMatrix ( void )
{
  GLfloat m[16];

  M4x4Translatef ( m, -viewer_pos0[0], -viewer_pos0[1], -viewer_pos0[2] );
  glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1], 16*sizeof(GLfloat), m );
  ExitIfGLError ( "InitViewMatrix" );
} /*InitViewMatrix*/

void RotateViewer ( double delta_xi, double delta_eta )
{
  float   vi[3], lgt, vk[3];
  double angi, angk;
  GLfloat vm[16];

  if ( delta_xi == 0.0 && delta_eta == 0.0 )
    return;  /* natychmiast uciekamy - nie chcemy dzielić przez zero */
  vi[0] = (float)delta_eta*(right-left)/(float)win_height;
  vi[1] = (float)delta_xi*(top-bottom)/(float)win_width;
  vi[2] = 0.0;
  lgt = sqrt ( V3DotProductf ( vi, vi ) );
  vi[0] /= lgt;  vi[1] /= lgt;
  angi = -0.052359878;  /* -3 stopnie */
  V3CompRotationsf ( vk, &angk, viewer_rvec, viewer_rangle, vi, angi );
```

```
46:   memcpy ( viewer_rvec, vk, 3*sizeof(float) );
47:   viewer_rangle = angk;
48:   M4x4RotateVfv ( vm, viewer_rvec, -viewer_rangle );
49:   M4x4InvTranslateMfv ( vm, viewer_pos0 );
50:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
51:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1], 16*sizeof(GLfloat), vm );
52:   ExitIfGLError ( "RotateViewer" );
53: } /*RotateViewer*/
```

o wektor $-\boldsymbol{t}$. Podobnie obrót wokół ustalonej osi o kąt α można zinterpretować jako przejście do układu współrzędnych obróconego wokół tej samej osi o kąt $-\alpha$. Zawsze układ odniesienia (składający się z początku układu i wersorów osi) w dualnej interpretacji jest poddawany przekształceniu odwrotnemu do przekształcenia punktu w interpretacji pierwotnej[9].

Naciskając przycisk i przesuwając mysz, intuicyjnie zamierzamy „chwycić" pewien wyobrażony punkt $\boldsymbol{p}$ znajdujący się *przed osią obrotu* (zobacz rys. 8.1) i spowodować jego przemieszczenie w kierunku ruchu myszy. Postrzegamy to jako polecenie obrócenia oglądanego obiektu; w rzeczywistości (dualnej) jako obserwator „odpychamy się" od „chwyconego" punktu i obracamy *się* (tj. obserwatora z jego układem współrzędnych) w przeciwną stronę względem nieruchomego obiektu. Obrócenie obiektu o 3° wokół osi o kierunku wektora $\boldsymbol{v}_i$ jest równoznaczne z obróceniem obserwatora o kąt −3° wokół tej osi.

Ponieważ obrót w i-tym kroku jest określony w układzie *obserwatora*, złożenie obrotów: dotychczas wykonanego, reprezentowanego przez wartości zmienych `viewer_rvec` i `viewer_rangle` oraz bieżącego, reprezentowanego w zmiennych `vi` i `angi` (lokalnych w procedurze `RotateViewer`), trzeba wykonać tak, aby współrzędne punktu były pomnożone *najpierw* przez macierz bieżącego obrotu, a potem wynik tego mnożenia przez macierz reprezentującą złożenie wszystkich obrotów wykonanych wcześniej. Tę kolejność realizują parametry procedury `V3CompRotationsf` wywołanej w linii 45.

Zatem, po wykonaniu procedury `V3CompRotationsf` mamy reprezentację *nowego obrotu układu* obserwatora względem układu świata, przypisaną (w liniach 46 i 47) zmiennym `viewer_rvec` i `viewer_rangle`. Ale obrót układu obserwatora o kąt φ_i oznacza, że *przejście do układu* współrzędnych obserwatora jest obrotem wokół tej samej osi o kąt $-\varphi_i$. Stąd ostatni parametr procedury `M4x4MRotateVf` w linii 48 został opatrzony znakiem „−".

Osie wszystkich rozpatrywanych tu obrotów przechodzą przez początek układu świata. Po obróceniu obserwator musi się jeszcze przesunąć do punktu, który w układzie obróconym ma współrzędne $(0, 0, 10)$; a więc obliczenie współrzędnych punktów w układzie przesuniętym jest przesunięciem o wektor $(0, 0, -10)$. Przesunięcie to jest dane w układzie obróconym, zatem właściwa kolejność mnożonych macierzy jest taka, jakby najpierw był wykonany obrót, a potem przesunięcie. Dlatego jest tu użyta procedura `M4x4InvTranslateMfv`, która

[9] To samo dotyczy nieobecnych w tej aplikacji skalowań. Pomnożenie współrzędnych kartezjańskich przez 2.54 można zinterpretować jako jednokładność *powiększającą obiekt* w tej skali, lub obliczenie współrzędnych punktu w centymetrach na podstawie jego współrzędnych podanych w calach — czyli przejście do układu, którego jednostki są 2.54 razy *krótsze*. Aby zmienić centymetry na cale (czyli *powiększyć jednostkę długości*), trzeba współrzędne przez 2.54 *podzielić*.

macierz obrotu, daną w tablicy przekazanej jako pierwszy parametr, przez macierz przesunięcia (o wektor przeciwny do danego w drugim parametrze) mnoży z *lewej* strony.

W liniach 50 i 51 przywiązujemy UBO, tj. bufor z macierzami przekształceń do celu GL_UNIFORM_BUFFER i przesyłamy do tego bufora współczynniki nowej macierzy przejścia do układu obserwatora.

Listing 8.3. Dodatkowe zmienne i procedury w części „okienkowej" aplikacji pierwszej A

C

```
#define STATE_NOTHING 0
#define STATE_TURNING 1

int last_xi, last_eta;
int app_state = STATE_NOTHING;

void MouseFunc ( int button, int state, int x, int y )
{
  switch ( app_state ) {
case STATE_NOTHING:
    if ( button == GLUT_LEFT_BUTTON && state == GLUT_DOWN ) {
      last_xi = x,  last_eta = y;
      app_state = STATE_TURNING;
    }
    break;
case STATE_TURNING:
    if ( button == GLUT_LEFT_BUTTON && state != GLUT_DOWN )
      app_state = STATE_NOTHING;
    break;
default:
    break;
  }
} /*MouseFunc*/

void MotionFunc ( int x, int y )
{
  switch ( app_state ) {
case STATE_TURNING:
    if ( x != last_xi || y != last_eta ) {
      RotateViewer ( (double)(x-last_xi), (double)(y-last_eta) );
      last_xi = x,  last_eta = y;
      glutPostWindowRedisplay ( WindowHandle );
    }
    break;
default:
    break;
  }
} /*MotionFunc*/
```

Interakcję z użytkownikiem zapewniają przedstawione na listingu 8.3 procedury `MouseFunc` i `MotionFunc`, które trzeba wstawić zamiast pustych procedur na listingu 3.1. Wartość zmiennej globalnej `app_state` określa stan (albo tryb pracy) aplikacji. W tej aplikacji są zdefiniowane dwie wartości tej zmiennej, liczby nazwane (za pomocą makrodefinicji) `STATE_NOTHING` i `STATE_TURNING`[10]. Wartość początkowa to `STATE_NOTHING`. Przemieszczenie kursora w trybie obracania (tj. gdy zmienna `app_state` ma wartość `STATE_TURNING`) powoduje obracanie obserwatora. Procedura `MouseFunc` jest wywoływana za każdym razem, gdy któryś przycisk myszy został naciśnięty lub zwolniony. Jeśli jest to lewy przycisk, to jego naciśnięcie spowoduje przejście aplikacji do trybu obracania obserwatora (zmienna `app_state` otrzymuje wartość `STATE_TURNING`) i zapamiętanie w zmiennych `last_xi` i `last_eta` położenia kursora w oknie. Jeśli lewy przycisk został zwolniony, to aplikacja wraca do stanu początkowego.

Wywołanie procedury `MotionFunc` w trybie obracania obserwatora powoduje obliczenie przemieszczenia kursora; jeśli przemieszczenie jest niezerowe (a może być zerowe), to wywoływana jest procedura `RotateViewer`, która oblicza nową macierz przejścia do układu obserwatora i przypisuje współczynniki tej macierzy odpowiedniej zmiennej jednolitej. Nowe położenie kursora jest zapamiętywane i na koniec FreeGLUT zostaje poinformowany o tym, że poprzedni obraz w oknie jest już nieaktualny i trzeba narysować nowy.

8.3. Animacja

Niech po naciśnięciu klawisza spacji obiekt sam się obraca, albo niech się przestanie obracać. Będziemy animować macierz przekształcenia modelu, tj. powodować obracanie modelu w układzie świata, niezależnie od ruchu obserwatora względem układu świata. Oś obrotu obiektu będzie tu ustalona; jest to oś y układu świata, będzie też stała prędkość obrotowa, $\pi/4$ na sekundę (czyli będzie jeden pełny obrót na 8 sekund). Część aplikacji, która to realizuje, korzysta z procedur opisanych w p. 3.5.1.

Listing 8.4 przedstawia zmiany w części okienkowej aplikacji. Do procedury `KeyboardFunc` została dodana reakcja na naciśnięcie klawisza spacji, które uruchamia lub zatrzymuje animację. Wywoływana wtedy procedura `ToggleAnimation`, zależnie od wartości nowej zmiennej `animate`, rejestruje za pomocą procedury `glutIdleFunc` procedurę `IdleFunc` i uruchamia stoper (zobacz p. 3.5.1), albo wyrejestrowuje tę procedurę, wywołując `glutIdleFunc` z parametrem `NULL`. Po zarejestrowaniu procedura `IdleFunc` jest wywoływana co chwila; wywołuje ona procedurę `MoveOn` z części „graficznej" aplikacji i przedstawia FreeGLUT-owi życzenie wykonania nowego obrazu.

Makrodefinicje, zmienne i procedury dodane do części graficznej aplikacji są przedstawione na listingu 8.5. Makrodefinicja `ANGULAR_VELOCITY` określa prędkość kątową obrotów obiektu, równą $\pi/4$. W zmiennych `model_rot_axis` i `model_rot_angle` są zapisane (ustalone) współrzędne wektora osi obrotu oraz miara kąta obrotu obiektu wokół tej osi w danej chwili.

[10] Nieważne, jakie to są liczby, ważne, że są różne.

Listing 8.4. Procedury „okienkowe” animacji

C

```
char animate = false;

void IdleFunc ( void )
{
  if ( MoveOn () )
    glutPostWindowRedisplay ( WindowHandle );
} /*IdleFunc*/

void ToggleAnimation ( void )
{
  if ( (animate = !animate) ) {  /* zaczynamy obracanie */
    TimerTic ();
    glutIdleFunc ( IdleFunc );
  }
  else                           /* kończymy obracanie */
    glutIdleFunc ( NULL );
} /*ToggleAnimation*/

void KeyboardFunc ( unsigned char key, int x, int y )
{
  switch ( key ) {
case 0x1B:             /* klawisz Esc - zatrzymanie programu */
    Cleanup ();
    glutLeaveMainLoop ();
    break;
case ' ':              /* włączamy albo wyłączamy animacje */
    ToggleAnimation ();
    break;
default:
    if ( ProcessChar ( key ) )
      glutPostWindowRedisplay ( WindowHandle );
    break;
  }
} /*KeyboardFunc*/
```

Nazwę procedury `InitModelMatrix` zmieniłem na `SetupModelMatrix`, co jest motywowane tym, że procedura ta nie służy już tylko do inicjalizacji macierzy przekształcenia modelu, ale do konstruowania tej macierzy dla wszelkich kątów obrotu podawanych w trakcie animacji. Parametry procedury (wektor i kąt) reprezentują obrót modelu. Procedura `InitMyWorld` wywołuje tę procedurę, podając jako parametry zmienne o wartościach nadanych w deklaracji — obrót o kąt 0 jest przekształceniem tożsamościowym, a zatem na początku działania aplikacji macierz modelu jest jednostkowa. Ponadto procedura `InitMyWorld` wywołuje procedurę `TimerInit`, która przygotowuje stoper do działania.

Listing 8.5. Procedury „graficzne” animacji

C

```
1: #define ANGULAR_VELOCITY (0.25*PI)
2:
3: float  model_rot_axis[3] = {0.0,1.0,0.0};
4: double model_rot_angle = 0.0;
5:
6: void SetupModelMatrix ( float axis[3], double angle )
7: {
8:   GLfloat m[16];
9:
10:  M4x4RotateVf ( m, axis[0], axis[1], axis[2], angle );
11:  glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
12:  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[0], 16*sizeof(GLfloat), m );
13:  ExitIfGLError ( "SetupModelMatrix" );
14: } /*SetupModelMatrix*/
15:
16: char MoveOn ( void )
17: {
18:  if ( (model_rot_angle += ANGULAR_VELOCITY * TimerTocTic ()) >= PI )
19:    model_rot_angle -= 2.0*PI;
20:  SetupModelMatrix ( model_rot_axis, model_rot_angle );
21:  return true;
22: } /*MoveOn*/
23:
24: void InitMyWorld ( int argc, char *argv[], int width, int height )
25: {
26:  LoadMyShaders ();
27:  TimerInit ();
28:  SetupModelMatrix ( model_rot_axis, model_rot_angle );
29:  InitViewMatrix ();
30:  ConstructIcosahedronVAO ();
31:  ResizeMyWorld ( width, height );
32: } /*InitMyWorld*/
```

Procedura `MoveOn`, wywoływana przez `IdleFunc`, w linii 18 dodaje do bieżącego kąta obrotu przyrost obliczony na podstawie odczytu stopera. Procedura `TimerTocTic`, podając czas, kasuje stoper, a zatem podana przez nią wartość jest przyrostem czasu od poprzedniego wywołania (lub za pierwszym razem od wywołania procedury `TimerTic` przez `ToggleAnimation`). Przyrost kąta obrotu jest iloczynem prędkości kątowej i przyrostu czasu podanego przez stoper. Instrukcja w linii 19 powoduje zmniejszenie kąta o 2π, aby wartość zmiennej `model_rot_angle` zawsze była liczbą z przedziału $[-\pi, \pi)$, co gwarantuje najmniejszy błąd względny zmiennopozycyjnej reprezentacji tego kąta. Po wyłączeniu animacji wartość zmiennej `model_rot_angle` pozostaje niezmieniona, dzięki czemu po ponownym włączeniu animacji obiekt rusza z położenia, w którym się zatrzymał.

Procedura `MoveOn` należy do interfejsu między częścią okienkową a graficzną aplikacji (w dodatku do procedur `InitMyWorld`, `ResizeMyWorld`, `RedrawMyWorld`, `DeleteMyWorld`, `ProcessCharCommand` i `RotateViewer`). Przekazuje ona zawsze wartość `true`, bo za każdym razem wskutek upływu czasu kąt obrotu obiektu się zmienił. Procedura `MoveOn` w pewnych zastosowaniach może podjąć decyzję o pozostawieniu dotychczasowego obrazu[11]. Wtedy powinna podać wartość `false`, aby nie spowodować wywołania `glutPostWindowRedisplay` przez procedurę `IdleFunc`.

8.4. Ćwiczenia

1. Rozbuduj aplikację o możliwość powiększania i zmniejszania obrazu w oknie. W tym celu należy wprowadzić dodatkową zmienną, reprezentującą „długość ogniskową obiektywu", na podstawie której ma być obliczany kąt α między płaszczyną xy układu obserwatora a górną i dolną ścianą bryły widzenia (zobacz rachunki na s. 152). Zmiana „długości ogniskowej", polegająca na pomnożeniu lub podzieleniu przez czynnik bliski jedynki, na przykład 1.05, może następować po użyciu rolki myszy — aplikacja biblioteki FreeGLUT po obróceniu rolki otrzymuje parę komunikatów o naciśnięciu i zwolnieniu przycisku o numerze 3 lub 4.[12]

   Wykonaj to ćwiczenie przed przeczytaniem rozdziału 13.

2. Czy potrafisz tak zmienić aplikację, aby po naciśnięciu spacji obiekt zaczynał i przestawał się obracać płynnie, tj. stopniowo zwiększając i zmniejszając swoją prędkość obrotową między zerem a prędkością maksymalną? W okresach „rozpędzania" i „hamowania" można przyjąć stałe przyspieszenia kątowe.

   Wskazówka: Zadeklaruj zmienną, której wartość jest bieżącą prędkością obrotową. Po naciśnięciu klawisza spacji nie można natychmiast wyłączyć animacji. Powinna to zrobić procedura `IdleFunc`, którą dodatkowa funkcja dodana do części „graficznej" zawiadomi, że już pora (gdy prędkość kątowa zmaleje do zera).

3. Rozszerzając poprzednie ćwiczenie, spróbuj zmienić aplikację tak, aby w reakcji na naciskanie dwóch klawiszy, na przykład + i –, prędkość obrotowa płynnie się zwiększała albo zmniejszała, co umożliwiałoby obserwowanie bryły obracającej się w różnym tempie.

4. *Połącz część graficzną aplikacji 1A z częścią okienkową realizowaną przez szkielet aplikacji biblioteki GLFW, X Window lub Windows opisany w podrozdziale 3.2, 3.3 lub 3.4 i uruchom aplikację. Wybierz środowisko najwygodniejsze dla siebie.

[11] Wywołanie procedury `MoveOn` może być spowodowane na przykład zmianą stanu dżojstika, pojawieniem się danych z sieci lub zakończeniem obliczeń wykonywanych w tle przez inny wątek aplikacji. Obsługa tych zdarzeń należy do części „okienkowej". Nie zawsze po takim zdarzeniu obraz musi być zmieniony.

[12] „Przyciski", które w bibliotece FreeGLUT służą do sygnalizowania obrotów rolki myszy, nie mają nazw symbolicznych takich jak `GLUT_LEFT_BUTTON`.

8.5. Uzupełnienia — powiększenie swobody ruchu obserwatora

Zrealizowany w aplikacji 1A ruch obserwatora jest bardzo ograniczony — obserwator pozostaje w niezmiennej odległości 10 od początku układu współrzędnych świata i jest zawsze obrócony tak, aby początek ten leżał na osi z układu obserwatora (wskutek czego obraz tego punktu jest zawsze środkiem klatki). Zobaczmy sposób realizacji ruchów obserwatora umożliwiający umieszczenie go w dowolnym punkcie przestrzeni i skierowanie „kamery" w dowolną stronę. Do obracania obserwatora wokół początku układu świata dodamy możliwość obracania go wokół jego bieżącego położenia (czyli wokół osi przechodzących przez początek układu obserwatora) oraz możliwość przesuwania obserwatora „do przodu" i „do tyłu", tj. wzdłuż osi z układu obserwatora.

W nowej zmiennej `viewer_pos` będziemy przechowywać wektor $\boldsymbol{p}$ współrzędnych położenia obserwatora w układzie świata. Kąt φ i wektor kierunkowy $\boldsymbol{z}$ osi obrotu przeprowadzającego wersory osi układu świata na wersory osi układu obserwatora są, jak wcześniej, pamiętane w zmiennych `viewer_rangle` i `viewer_rvec`. Macierz V przejścia od układu świata do obserwatora będzie potrzebna nie tylko do obliczeń w potoku przetwarzania grafiki, więc będziemy ją przechowywać w globalnej zmiennej `vm`.

Procedura `InitViewMatrix` nadaje obserwatorowi domyślne położenie początkowe.

Nowa procedura `RotateViewer` ma dodatkowy parametr `origin`, który wybiera punkt, wokół którego ma być obracany obserwator. Jeśli ten parametr ma wartość `true`, to punkt ten jest początkiem układu świata, a jeśli `false`, to obserwator ma się obrócić wokół *swojego* położenia. W części okienkowej należy wprowadzić nowy stan obracania obserwatora, w który aplikacja wchodzi na przykład po naciśnięciu prawego przycisku myszy.

Nowy sposób obliczania macierzy V jest zrealizowany w liniach 41–42. Macierz ta jest iloczynem $V = RT$ macierzy R obrotu o kąt $-\varphi$ wokół osi o kierunku $\boldsymbol{z}$ i macierzy T przesunięcia o wektor $-\boldsymbol{p}$. Teraz położenie obserwatora jest dane w układzie współrzędnych świata, dlatego w linii 42 jest wywołana procedura `M4x4MInvTranslatefv`, która macierz obrotu, przekazaną jako pierwszy parametr, mnoży przez macierz przesunięcia z *prawej* strony. Składanie obrotów wykonanych wcześniej z obrotem bieżącym jest wykonywane tak jak poprzednio. Jeśli oś bieżącego obrotu przechodzi przez położenie obserwatora, to oczywiście ono się nie zmienia. Jeśli obserwator ma się obrócić wokół osi przechodzącej przez początek układu świata, to należy obliczyć jego (obserwatora) nowe położenie.

Robi się to tak: w linii 36 wektor $\boldsymbol{v}_i$ osi obrotu bieżącego, dany w układzie obserwatora, jest przekształcany do układu świata; w tym celu jest on mnożony przez transpozycję górnego lewego bloku 3×3 macierzy V. Przypomnijmy, że przejście od układu świata do obserwatora jest izometrią. Wspomniany blok jest więc macierzą ortogonalną (reprezentuje obrót), a zatem jego transpozycja jest jego odwrotnością i opisuje część liniową przejścia od układu obserwatora do układu świata. Instrukcja w linii 37 konstruuje macierz tego obrotu (w układzie świata), a w linii 38 położenie obserwatora jest poddawane temu obrotowi. Dobór kąta obrotu pozostawiam Czytelnikom, w tym przypadku 3° to (moim zdaniem) za dużo.

Procedurę `MoveViewer` część okienkowa aplikacji może wywoływać po obróceniu rolki myszy — do przodu lub do tyłu. Polecenie przemieszczenia obserwatora do przodu powo-

Listing 8.6. Implementacja rozszerzenia swobody ruchu obserwatora

C

```
float   viewer_rvec[3], viewer_pos[4];
double  viewer_rangle;
GLfloat vm[16];

void InitViewMatrix ( void )
{
  const float viewer_pos0[4] = {0.0,0.0,10.0,1.0};
  const float viewer_rvec0[3] = {1.0,0.0,0.0};

  memcpy ( viewer_pos, viewer_pos0, 4*sizeof(float) );
  memcpy ( viewer_rvec, viewer_rvec0, 3*sizeof(float) );
  viewer_rangle = 0.0;
  M4x4InvTranslatefv ( vm, viewer_pos );
  glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1], 16*sizeof(GLfloat), vm );
  ExitIfGLError ( "InitViewMatrix" );
} /*InitViewMatrix*/

void RotateViewer ( double delta_xi, double delta_eta, char origin )
{
  float  vi[3], lgt, vk[3], vpos[4];
  double angi, angk;

  if ( delta_xi == 0.0 && delta_eta == 0.0 )
    return;
  vi[0] = (float)delta_eta*(right-left)/(float)win_height;
  vi[1] = (float)delta_xi*(top-bottom)/(float)win_width;
  vi[2] = 0.0;
  lgt = sqrt ( V3DotProductf ( vi, vi ) );
  vi[0] /= lgt;  vi[1] /= lgt;
  angi = ....;   /* do wyboru */
  V3CompRotationsf ( vk, &angk, viewer_rvec, viewer_rangle, vi, angi );
  memcpy ( viewer_rvec, vk, 3*sizeof(float) );
  viewer_rangle = angk;
  if ( origin ) {
    M4x4MultMTV3f ( vk, vm, vi );
    M4x4RotateVfv ( vm, vk, angi );
    M4x4MultMVf ( vpos, vm, viewer_pos );
    memcpy ( viewer_pos, vpos, 4*sizeof(float) );
  }
  M4x4RotateVfv ( vm, viewer_rvec, -viewer_rangle );
  M4x4MInvTranslatefv ( vm, viewer_pos );
  glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1], 16*sizeof(GLfloat), vm );
  ExitIfGLError ( "RotateViewer" );
```

```
46: } /*RotateViewer*/
47:
48: void MoveViewer ( char forward )
49: {
50:   const float stepf[3] = {0.0,0.0,-0.2}, stepb[3] = {0.0,0.0,0.2};
51:   float v[3];
52:   int   i;
53:
54:   M4x4MultMTV3f ( v, vm, forward ? stepf : stepb );
55:   for ( i = 0; i < 3; i++ )
56:     viewer_pos[i] += v[i];
57:   M4x4RotateVfv ( vm, viewer_rvec, -viewer_rangle );
58:   M4x4MInvTranslatefv ( vm, viewer_pos );
59:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
60:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1], 16*sizeof(GLfloat), vm );
61:   ExitIfGLError ( "MoveViewer" );
62: } /*MoveViewer*/
```

duje przesunięcie go o wektor $\boldsymbol{t} = (0, 0, -1/5)$, a krok do tyłu opisuje wektor $-\boldsymbol{t}$. Te wektory są dane w układzie współrzędnych *obserwatora*, dlatego w linii 54 następuje obliczenie współrzędnych wektora przemieszczenia obserwatora w układzie świata. Pętla w liniach 55–56 dodaje przemieszczenie do dotychczasowego położenia obserwatora, po czym nowa macierz V jest konstruowana tak samo, jak po obróceniu go. Do procedury można dodać wykonywanie przesunięć na boki oraz do góry i do dołu, odpowiednio rozszerzając interpretację parametru, i umożliwić przesuwanie obserwatora w dodanych kierunkach, na przykład za pomocą klawiszy strzałek. Długości wektorów przesunięć należy dobrać do aplikacji.

Opisane wyżej rozszerzenie możliwości przemieszczania obserwatora stwarza dodatkowe problemy. Jeśli mamy tylko jeden niewielki obiekt, to łatwo można go „zgubić", a gdy już wypadnie poza bryłę widzenia, trudno zgadnąć, dokąd się zwrócić, aby ponownie ukazał się w oknie. Warto więc dodać polecenie przywracające domyślne położenie obserwatora. Trzeba unikać kolizji obserwatora z obiektami, na przykład przez wprowadzenie ograniczeń przemieszczania go zależnych od ich rozmieszczenia. Zbytnie zbliżenie się do lub oddalenie od obiektu też spowoduje jego wyjście poza bryłę widzenia, a wcześniej jego fragmenty zostaną poobcinane przez płaszczyznę przedniej lub tylnej ściany bryły widzenia, trzeba zatem dostosowywać jej parametry do odległości obserwatora od obiektu. Jeśli obserwator znajdzie się wewnątrz obiektu, którego tylne ściany są pomijane (zobacz p. 7.6.2), to podczas rysowania tego obiektu trzeba wybrać pomijanie ścian o przeciwnej orientacji.

Z tych powodów aplikacje opisane dalej realizują tylko sposób obracania obserwatora opisany w podrozdziale 8.2. Do zilustrowania działania procedur OpenGL-a to wystarczy.

9

Podstawy języka GLSL

Język GLSL, w którym mają być napisane szadery dla aplikacji OpenGL-a, różni się od języka używanego dla aplikacji Vulkana. Specyfikacja GLSL 4.6 [4] zawiera opis tych różnic. Ten rozdział jest oparty na specyfikacji GLSL 4.5 [3], uwzględniającej tylko standard OpenGL.

Poszczególne rodzaje szaderów mają istotnie różne role do spełnienia, w związku z czym zestaw dopuszczalnych konstrukcji jest w każdym przypadku inny. Dlatego specyfikacja [3] oficjalnie stwierdza, że GLSL to w istocie *sześć różnych* języków, każdy dla szaderów określonego typu. Ale większość elementów języki te mają wspólne.

Język GLSL jest dosyć podobny do języka C; wychodząc z założenia, że ten ostatni jest Czytelnikowi znany, przedstawię go przez wskazanie różnic między językami GLSL a C.

9.1. Symbole leksykalne

Tekst źródłowy programu w GLSL-u (szadera) składa się z jednego lub wielu napisów (łańcuchów ASCII lub ASCIIZ), których tablicę rejestruje się w obiekcie szadera (zobacz rozdział 4) przed kompilacją. W napisach tych wyróżniane są **separatory** (spacje, znaki końca linii i komentarze), **identyfikatory** (ciągi liter i cyfr zaczynające się od litery), przy czym pewne identyfikatory są **słowami kluczowymi** (w tej książce będą podkreślane), **literały** (czyli liczby, nie ma znanych w C znaków ani napisów) i **operatory** zbudowane ze znaków specjalnych.

Tak jak w C, w GLSL-u rozróżnia się wielkie i małe litery w identyfikatorach, zatem identyfikator `Out` jest czymś innym niż słowo kluczowe `out`. Znak podkreślenia `_` jest też literą, ale nie wolno w identyfikatorze napisać dwóch takich znaków obok siebie. Ponadto identyfikatory zaczynające się od przedrostka `gl_` są zarezerwowane.

Komentarze mają dwie dopuszczalne formy, tak jak w języku C++. Pierwsza forma, znana z C, to

```
/* dowolny tekst */,
```

wewnątrz którego nie może wystąpić para znaków `*/`. Druga forma to

```
// dowolny tekst do końca linii.
```

Kompilator GLSL-a komentarze pomija, ale trzeba je pisać; programy piszemy dla swojej przyjemności, nie kompilatora.

9.2. Preprocesor

Kompilator GLSL-a jest wyposażony w preprocesor podobny do tego znanego z C; **dyrektywa preprocesora** składa się ze słowa kluczowego preprocesora z doklejonym z przodu znakiem #, a po nim następuje treść dyrektywy. W linii przed słowem kluczowym dyrektywy i po jej treści mogą być tylko spacje i tabulatory.

`#version` — dyrektywa deklarująca wersję języka, w której szader jest napisany; *musi* wystąpić na początku jego tekstu[1]. Użycie konstrukcji językowej z wersji nowszej niż podana w tej dyrektywie powoduje błąd kompilacji. Numer wersji podaje się jako liczbę trzycyfrową, 100 razy większą niż numer wersji traktowany jak ułamek. Po tym numerze *można* podać słowo `core`, `compatibility` albo `es`, które deklaruje używany profil: odpowiednio **profil podstawowy**, **profil zgodności** (ze starym OpenGL-em) lub **profil dla systemów wbudowanych** (*embedded systems*), czyli po ludzku dla telefonów komórkowych[2].

Zaczynając od wersji 3.3 OpenGL-a, numeracja wersji GLSL i OpenGL jest zsynchronizowana. Zatem, pisząc program w OpenGL-u 4.5, powinno się deklarować wersję GLSL 4.5, choć można używać też starszych wersji języka. Przykładowa deklaracja wersji języka może wyglądać tak:

```
#version 450 core
```

Przypominam, że konieczne jest zakończenie tej deklaracji znakiem końca linii (zobacz linię 2 na listingu 4.4).

`#define`, `#undef` — utworzenie i likwidacja makrodefinicji. Te dyrektywy działają tak samo jak w C, w szczególności makra mogą być sparametryzowane. Jeśli makrodefinicja jest długa, to można ją zapisać w większej liczbie linii, stawiając na końcu każdej linii makra oprócz ostatniej znak \.

`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif` — dyrektywy kompilacji warunkowej, które działają tak samo jak w języku C, przy czym wyrażenia opisujące warunki kompilacji mają ograniczenia wynikające z dostępnych konstrukcji w języku GLSL (zobacz [3]), nieco innych niż w C.

`#pragma` — opcja dla kompilatora, na przykład `optimize(off)` wyłączająca optymalizację, albo `debug(on)` ułatwiająca uruchamianie szadera. Te pragmy nie mogą wystąpić wewnątrz podprogramu.

[1] Jeśli dyrektywa `#version` jest nieobecna, to kompilator uznaje, że szader jest napisany zgodnie ze specyfikacją 1.10 i jest przeznaczony do współpracy z aplikacją starego OpenGL-a.

[2] Aplikacja na telefon powinna być napisana zgodnie ze standardem OpenGL ES, którym tu się nie zajmujemy.

`#extension` — dyrektywa określająca sposób postępowania w sytuacji, gdy pewne rozszerzenie języka jest albo nie jest dostępne.

`#error` — powoduje błąd kompilacji. Zazwyczaj umieszcza się ją w miejscu obłożonym jakąś dyrektywą kompilacji warunkowej, na przykład uzależniającej powodzenie kompilacji od obecności potrzebnego rozszerzenia.

`#line` — dla porządku odnotujmy tę dyrektywę służącą do zapisywania numerów linii tekstu w jakimś innym języku, na podstawie którego tekst w GLSL-u został wygenerowany automatycznie, i nie przejmujmy się tym zbytnio. Trochę bardziej przejmujemy się *brakiem* dyrektywy `#include`, ale bez przesady.

9.3. Podstawowe typy zmiennych

Typy proste to `void` (dla procedur, które nie przekazują wyniku przez nazwę), `bool` (typ boolowski, tj. logiczny), `uint` (liczby całkowite bez znaku), `int` (liczby ze znakiem), `float` (liczby zmiennopozycyjne pojedynczej precyzji) oraz `double` (liczby zmiennopozycyjne podwójnej precyzji). Zamiast `uint` można napisać `unsigned int`.

Zmienne typu `bool` zajmują w pamięci 32 bity, ale mogą przyjmować tylko dwie wartości: `false` i `true`[3]. Wartości te są reprezentowane przez takie same bity jak liczby 0 i 1 w typie `int`, ale nie mogą być argumentami działań arytmetycznych na liczbach.

Liczby całkowite typu `int` oraz `uint` są 32-bitowe. Literały tych typów mogą być podane w postaci ciągu cyfr dziesiętnych, ósemkowych lub szesnastkowych, zapisywanych tak samo jak w C. Może wystąpić przyrostek u, który oznacza, że podany literał jest typu `uint`.

Liczby zmiennopozycyjne są reprezentowane zgodnie ze standardem IEEE-754, ale działania na nich *nie muszą* w pełni realizować tego standardu — chodzi o możliwość wyboru sposobu zaokrąglania (której nie ma), postępowanie w przypadku niedomiaru i o dostępność reprezentacji nie-liczb (*not a number, NaN*) i nieskończoności.

Typy wektorowe składają się z dwóch, trzech lub czterech składowych prostego typu liczbowego lub boolowskiego. Cztero- lub pięcioznakowe nazwy tych typów składają się z opcjonalnego jednoliterowego przedrostka `b`, `i`, u lub `d`, rdzenia `vec` i przyrostka — cyfry 2, 3 lub 4 określającej liczbę składowych. Przedrostek oznacza, że składowe są odpowiednio typu `bool`, `int`, `uint` lub `double`, a brak przedrostka oznacza, że są typu `float`. Nazwy wszystkich tych typów (jest ich 15) są słowami kluczowymi. Proszę zatem samemu rozszyfrować, co oznaczają typy `ivec2`, `bvec3` i `vec4`.

W podobny sposób są utworzone 24 nazwy 18 **typów macierzowych**, przy czym składowe tych macierzy mogą być tylko typu `float` albo `double`, co jest określone odpowiednio brakiem przedrostka lub przedrostkiem `d` przed rdzeniem `mat`. Przyrostek jest jedno- lub trzyznakowy i albo jest to cyfra, albo dwie cyfry przedzielone literą `x`. Typy macierzy kwadratowych mają po *dwie nazwy*, na przykład `mat2` to jest to samo co `mat2x2`, natomiast pozostałe macierze prostokątne mają tylko przyrostek trzyznakowy, na przykład `dmat3x4`.

[3] Inaczej niż w C, choć tak samo jak w C++, ich identyfikatory są słowami kluczowymi.

Uwaga: Pierwsza cyfra w nazwie typu macierzowego oznacza liczbę kolumn, a druga (jeśli są dwie) wierszy. Zatem na przykład macierz 3×4 może być wartością zmiennej typu `mat4x3`.

Zmienne **typów zamkniętych** (*opaque types*) są w zasadzie reprezentowane jak liczby całkowite, ale służą tylko jako identyfikatory obiektów — tekstur, ewaluatorów tekstury i liczników niepodzielnych (*atomic counters*) — i nie mogą być argumentami działań arytmetycznych, ale mogą być parametrami podprogramów i mogą być składowymi zmiennych jednolitych i innych struktur. W specyfikacji GLSL 4.5 [3] naliczyłem w sumie 41 typów zamkniętych (i 75 słów kluczowych będących ich nazwami), ale tu ich na razie nie wypiszę.

Nie ma typów wskaźnikowych. Jedyne zmienne wskaźnikowe, jakie mogą wystąpić w programie w GLSL-u, to opisane dalej zmienne jednolite wskazujące podprogramy.

9.3.1. Typy wektorowe i macierzowe

Zmienna typu `vec4` składa się z czterech pól typu `float`. Pola te mają aż trzy zestawy nazw: `xyzw`, `rgba` i `stpq`, które można wybierać dowolnie. Intencją jest umożliwienie poprawienia czytelności tekstu źródłowego przez używanie nazw odpowiednich do zastosowania danej zmiennej. Pierwszy zestaw jest odpowiedni dla współrzędnych punktów i wektorów opisujących obiekty geometryczne, drugi zestaw sugeruje, że pola są współrzędnymi koloru (łącznie ze składową alfa), a trzeci zestaw jest przeznaczony do opisu współrzędnych tekstury.

Typy `vec2` i `vec3` mają odpowiednio tylko dwie albo trzy początkowe nazwy pól w każdym zestawie. Dostęp do poszczególnych pól odbywa się za pomocą operatora kropki, na przykład dla zmiennej `a` typu `vec4` możemy pisać `a.x`, `a.t` lub `a.b`, co wybiera odpowiednio pierwsze, drugie i trzecie pole. Nazwy tych pól (ale tylko wzięte z jednego, dowolnego zestawu) można sklejać, aby uzyskać dostęp do kilku pól naraz, w celu ich „wypreparowania", poprzestawiania lub nawet powielenia. Na przykład wyrażenia `a.xy`, `a.ywz` i `a.xxxx` mają odpowiednio typy `vec2`, `vec3` i `vec4`; pierwszy wektor ma pierwsze dwie współrzędne wektora `a`, drugi ma jego ostatnie trzy współrzędne, przy czym ostatnie dwie są przestawione, a wszystkie cztery współrzędne trzeciego wektora są równe pierwszej współrzędnej wektora `a`. Nie wolno w ten sposób „wyprodukować" wektora o większej niż 4 liczbie współrzędnych.

Wektory i wybrane ich pola można dodawać i odejmować (byleby składniki miały tyle samo współrzędnych), a także mnożyć i dzielić przez liczby. W szczególności wyrażenie `a.xyz/a.w` ma typ `vec3`; produkuje ono wektor współrzędnych kartezjańskich punktu, którego współrzędne jednorodne są wartościami pól wektora `a`.

Konstruktory wektorów mają nazwy takie jak typy tych wektorów. Jeśli zmienne `b`, `c` są typu `vec2`, to zmiennej `a` typu `vec4` możemy nadać wartość wyrażenia takiego jak `vec4(b,c)`, `vec4(b.xxy,1.0)` lub `vec4(1.0)`. W ostatnim przypadku wszystkie cztery pola otrzymują tę samą wartość.

Macierz typu `mat4` (mającego też nazwę `mat4x4`) ma cztery wiersze i tyleż kolumn; dostęp do poszczególnych współczynników uzyskuje się tak, jakby to była tablica, na przykład

`a[2][1]` jest to trzeci współczynnik w drugim wierszu[4]. Można odwoływać się do całych kolumn, pisząc na przykład `a[2]` — dla macierzy `a` typu `mat4` jest to wektor typu `vec4`.

Konstruktory macierzy służą do utworzenia macierzy z liczb, wektorów lub innych macierzy; po słowie kluczowym będącym nazwą typu konstruowanej macierzy podaje się w nawiasach okrągłych argumenty, rozdzielone przecinkami. Jeśli jest jeden argument — liczba — to powstaje macierz diagonalna z tą liczbą na wszystkich miejscach diagonali, na przykład `mat4(1.0)` tworzy macierz jednostkową 4×4. Dla macierzy $m \times n$ można podać mn liczb, które będą umieszczone w kolejnych kolumnach macierzy. Można też podać n wektorów o m współrzędnych; staną się one kolumnami. Konstruktorów można też użyć do „wycinania" bloków. Jeśli na przykład zmienna `a` jest typu `mat4`, to `mat3(a)` wybiera górny lewy blok 3×3 macierzy będącej wartością tej zmiennej.

Działania na wektorach i macierzach są tak intensywnie wykorzystywane, że zapisuje się je jak działania elementarne, bez potrzeby obliczania osobno wszystkich współrzędnych wyniku[5] (np. mnożenie macierzy `a` przez wektor `v` zapisujemy jako wyrażenie `a*v`). Istotne jest dopasowanie argumentów działań — dodawane i odejmowane mogą być wektory i macierze o tych samych wymiarach. W mnożeniu dwóch macierzy liczba kolumn pierwszej z nich musi być równa liczbie wierszy drugiej.

Powyższe uwagi stosują się też do wektorów i macierzy, których pola mają typ inny niż `float`.

9.3.2. Struktury

Definicja typu strukturalnego wykorzystuje słowo kluczowe `struct`[6] i może być jednocześnie deklaracją zmiennych tego typu. Na przykład

```
struct moje {
    float a, b, c;
    vec4  qq[3];
  } z;
```

definiuje nowy typ strukturalny o nazwie `moje` i jednocześnie deklaruje zmienną `z` tego typu. Struktura zadeklarowana wyżej ma cztery pola, z których trzy są typu `float`, a czwarte jest tablicą trzech wektorów, z których każdy ma cztery współrzędne. Po zdefiniowaniu typu strukturalnego można deklarować zmienne tego typu, pisząc na przykład

```
moje e, f, g;
```

Dostęp do pól struktury odbywa się za pomocą operatora kropki, na przykład `e.c`.

[4] Odwrotnie niż zazwyczaj w C — to ma związek z kolumnowym przechowywaniem współczynników macierzy typów `mat2x2`, …, `mat4x4`.

[5] Procesory graficzne mają specjalne rozkazy wykonujące działania na całych wektorach i macierzach (szczegóły znają tylko producenci sprzętu).

[6] Nie ma znanego z C słowa kluczowego `typedef`, ani powodu do używania go.

9.3.3. Tablice

Podanie po nazwie zmiennej nawiasów kwadratowych `[ ]` obejmujących stałe wyrażenie liczbowe czyni z tej zmiennej tablicę (o długości takiej jak wartość tego wyrażenia). Zgodnie ze specyfikacją [3] dopuszczalne jest puste określenie długości *ostatniej* tablicy zadeklarowanej w bloku magazynowym (czyli tam i tylko tam można napisać np. `moje a[];`). W szczególności wszystkie tablice w blokach zmiennych jednolitych oraz tablicowe parametry podprogramów muszą mieć jawnie podaną długość. Tak jak w C, indeksy tablicy o długości n mają zakres od 0 do $n-1$. Inaczej niż w C, typ tablicowy *może* być typem wyniku podprogramu — funkcji.

Deklaracja zmiennej tablicowej może określać wartości początkowe, podane w **konstruktorze tablicy**, na przykład

```
float a[4] = float[]{0.0, 1.0, 2.0, 3.0};
```

(można podać długość także w konstruktorze, obie długości muszą być wtedy jednakowe). Tablice są zasadniczo tylko jednowymiarowe, ale (tak jak w C) można deklarować tablice tablic, na przykład

```
float b[2][3];
```

Elementy takiej tablicy zajmują kolejne pozycje w pamięci, przy czym najszybciej (ze wzrostem adresów kolejnych elementów tablicy w pamięci) zmienia się *ostatni* indeks (ale między elementami tablicy mogą być odstępy, jeśli to wynika z reguł określonych przez układ (*layout*) przyjęty dla zmiennej tablicowej).

Tablica jest *obiektem* wyposażonym w *metodę* o nazwie `length`. Metoda ta podaje długość tablicy. Na przykład dla zmiennej `b` zadeklarowanej jak wyżej jest `b.length == 2`, `b[0].length == 3`.

9.4. Deklaracje zmiennych

Deklaracja zmiennej może być na zewnątrz podprogramów (globalna), wewnątrz podprogramu (lokalna), albo nawet wewnątrz instrukcji złożonej. W ostatnich dwóch przypadkach zmienna jest widoczna tylko między najciaśniej obejmującymi ją nawiasami klamrowymi, definiującymi **zakres widoczności**. Natomiast w pierwszym przypadku zakres widoczności zmiennej jest wszędzie tam, gdzie jej nazwa nie jest zasłonięta przez nazwę jakiegoś obiektu lokalnego. Można też zadeklarować zmienną sterującą pętli w jej nagłówku i wtedy zmienna jest widoczna wewnątrz pętli, na przykład

```
for ( int i = 0; i < n; i++ ) { if ( i > 0 ) ....  }
```

Uwaga: Wszystkie nazwy zmiennych, typów strukturalnych i podprogramów znajdują się w *tej samej* przestrzeni nazw w danym zakresie widoczności. Zatem, choć nazwy podprogramów są przeciążane (np. funkcje obliczające iloczyn skalarny wektorów o 2, 3 i 4 współ-

rzędnych mają tę samą nazwę `dot`, ale każda ma argumenty innego typu), nie można mieć jednocześnie zmiennej i podprogramu o tej samej nazwie.

Globalna deklaracja zmiennej może (*nie musi*) być poprzedzona **kwalifikatorem zmiennej** (*storage qualifier*). Używane w profilu podstawowym kwalifikatory zmiennych są takie:

`buffer` — blok magazynowy, którego zawartość mogą przypisywać i odczytywać zarówno szadery, jak i aplikacja działająca na CPU, wywołując odpowiednie procedury OpenGL-a.

`const` — zmienna, której wartości nie wolno zmieniać; jej wartość jest przypisana w deklaracji.

`in` — zmienna, której wartość została nadana przez wcześniejszy etap potoku przetwarzania grafiki.

`out` — zmienna, której szader ma nadać wartość przekazywaną następnie do kolejnego etapu potoku przetwarzania grafiki.

`shared` — zmienna współdzielona, do której dostęp mają (działające jednocześnie na wielu procesorach GPU) wątki szadera obliczeniowego, w ramach lokalnej grupy roboczej (podrozdz. 9.15). Przykład użycia takich zmiennych jest w p. 29.2.6.

`uniform` — zmienna jednolita lub blok zmiennych jednolitych.

Zmienne globalne (tj. zadeklarowane poza podprogramem) opatrzone kwalifikatorami `buffer`, `uniform`, `in`, `out` i `shared` są tzw. **zmiennymi interfejsu**; zgodnie z opisem wyżej, służą one do przekazywania danych między aplikacją działającą na CPU a szaderami, między poszczególnymi etapami potoku przetwarzania grafiki lub między wątkami szadera obliczeniowego w lokalnej grupie roboczej. Pozostałe zmienne globalne są widoczne tylko w obrębie jednego szadera, co więcej, tylko jednego wątku szadera przetwarzającego równolegle dane (np. wierzchołki lub fragmenty)[7].

Jeśli bezpośrednio po kwalifikatorze `uniform` zmiennej jest podany typ, a po nim nazwa, to zmienna jednolita o tej nazwie trafia do **domyślnego bloku zmiennych jednolitych** programu szaderów. Tylko ten program (tzn. wszystkie wchodzące w jego skład szadery zawierające deklarację tej zmiennej) ma dostęp do tej zmiennej. Jej wartość może przypisać aplikacja za pomocą odpowiedniej procedury z rodziny `glUniform*`, na przykład `glUniform1i`[8]. Procedury te przypisują wartości zmiennym jednolitym zadeklarowanym w bieżącym (wybranym za pomocą `glUseProgram`) programie szaderów. Pierwszy parametr każdej z tych procedur określa **położenie** (*location*) zmiennej, które jest liczbą całkowitą. Deklaracja zmiennej jednolitej postaci

`layout(location=`*n*`) uniform` <*typ*> <*nazwa*>;

wymusza położenie *n* dla tej zmiennej. Poszczególne zmienne jednolite muszą mieć różne położenia, za co odpowiedzialność, pisząc kwalifikatory położenia, autor szaderów bierze

[7] Zatem każdy wątek ma swój prywatny „egzemplarz" takiej zmiennej.

[8] Wartość takiej zmiennej aplikacja może odczytać za pomocą procedury z rodziny `glGetUniform*`, na przykład `glGetUniform1i`. Programy szaderów przeznaczone do pracy z aplikacjami Vulkana nie mogą mieć domyślnych bloków zmiennych jednolitych.

na siebie. Jeśli zmienna jednolita jest zadeklarowana bez kwalifikatora położenia, to jej położenie przydzieli kompilator, aplikacja zaś może je poznać za pomocą procedury `glGetUniformLocation`.

Jeśli zaraz po kwalifikatorze `uniform` jest podany identyfikator niebędący nazwą typu, a po nim w klamrach typy i nazwy pól (będących zmiennymi jednolitymi), to identyfikator ten jest nazwą zewnętrzną bloku zmiennych jednolitych. Dla takich zmiennych trzeba utworzyć bufor (UBO) i przywiązać go do odpowiedniego punktu dowiązania, a przypisanie wartości następuje za pomocą procedur `glBufferData` i `glBufferSubData`. Przykład zastosowania tej konstrukcji języka GLSL można zobaczyć w pierwszej aplikacji i we wszystkich kolejnych w tej książce.

9.5. Wyrażenia

Wyrażenia w GLSL-u, tak jak w C, buduje się z argumentów (stałych, zmiennych, funkcji) i operatorów zebranych w tabeli 9.1. Większość operatorów (ale nie wszystkie) ma takie samo działanie jak w języku C. W szczególności podobne są ich priorytety i łączność. Operatory o wyższym priorytecie są w tabeli podane wyżej (i od operatorów o niższym priorytecie są oddzielone kreską).

Łączność określa kolejność wykonywania działań realizowanych przez sąsiadujące operatory o tym samym priorytecie. Na przykład w wyrażeniu `a-b+c` odejmowanie `a-b` zostanie wykonane najpierw, ponieważ priorytet obu operatorów w tym wyrażeniu jest taki sam, a ich łączność jest lewostronna[9].

W języku GLSL nie ma wskaźników (z wyjątkiem opisanych dalej zmiennych wskazujących podprogramy), zatem nie ma znanych z C operatorów brania adresu (`&`) i sięgania pod adres (`*`). Nie ma też operatora `sizeof`; jest on zbędny, ponieważ szadery nie dokonują dynamicznej rezerwacji pamięci (wszelkie polecenia rezerwacji i likwidacji buforów w pamięci GPU wydaje aplikacja działająca na CPU)[10].

Nie ma znanych z C operatorów rzutowania typu, dokonujących konwersji typu wartości wyrażenia. Zamiast tego słowa kluczowe `int`, `uint`, `bool`, `float` i `double` mogą być użyte w celu dokonania odpowiedniej konwersji, przy czym składnia jest taka jak wywołanie funkcji — można napisać na przykład `double(3)`. Konwersja liczby zmiennopozycyjnej (typu `float` lub `double`) do całkowitej (`int` lub `uint`) wiąże się z odrzuceniem części ułamkowej. Wartości logiczne `false` i `true` są konwertowane na liczby 0 (lub 0.0) i 1 (lub 1.0).

Argumentami operatorów logicznych mogą być tylko zmienne lub wyrażenia logiczne (typu `bool`). Argumentami operatorów arytmetycznych mogą być tylko zmienne liczbowe. Dzielenie przez 0 nie jest sygnalizowane (tzn. nie przerywa normalnego działania programu wywołaniem jakiejś procedury obsługi sygnału), ale jego wynik jest nieokreślony.

[9] Przed nauczeniem się tabeli na pamięć, w razie wątpliwości, można używać nawiasów okrągłych do zapewnienia pożądanej kolejności wykonywania działań.

[10] Identyfikator `sizeof` (i 38 innych niebędących słowami kluczowymi) jest zarezerwowany dla przyszłych wersji języka GLSL, a jego wystąpienie w tekście szadera powoduje błąd kompilacji. Pełna lista słów kluczowych i identyfikatorów zarezerwowanych jest w specyfikacji [3].

Tabela 9.1. Operatory języka GLSL, ich priorytety i łączność

operator	opis	łączność
`( )`	ogranicznik podwyrażenia	nieokreślona
`[ ]`	indeks tablicy	lewostronna
`( )`	wywołanie podprogramu, konstruktor	
`.`	dostęp do pól struktury, przestawianie	
`++`, `--`	zwiększanie i zmniejszanie *po*	
`++`, `--`	zwiększanie i zmniejszanie *przed*	prawostronna
`+`, `-`, `~`, `!`	operacje jednoargumentowe	
`*`, `/`, `%`	mnożenie, dzielenie, reszta z dzielenia	lewostronna
`+`, `-`	dodawanie, odejmowanie	lewostronna
`<<`, `>>`	przesunięcia bitowe	lewostronna
`<`, `>`, `<=`, `>=`	operatory relacyjne	lewostronna
`==`, `!=`	operatory relacyjne	lewostronna
`&`	koniunkcja bitowa	lewostronna
`^`	bitowa alternatywa wyłączająca (xor)	lewostronna
`|`	alternatywa bitowa	lewostronna
`&&`	koniunkcja logiczna	lewostronna
`^^`	logiczna alternatywa wyłączająca	lewostronna
`||`	alternatywa logiczna	lewostronna
`? :`	wybór (operator trójargumentowy)	prawostronna
`=`	przypisanie	prawostronna
`+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`	działania z przypisaniem	
`,`	separator wyrażeń	lewostronna

Operatory jednoargumentowe `~` i `!` dokonują odpowiednio negacji bitowej (argument jest liczbą całkowitą, zmieniane są wszystkie bity) i logicznej (argument jest typu `bool`).

9.6. Instrukcje

Instrukcje w języku GLSL są podobne do tych w C, choć gramatyka opisująca ich składnię różni się od gramatyki C. Ale mamy w GLSL-u do dyspozycji **instrukcje proste** — deklaracje, instrukcje wyrażeniowe, instrukcje wyboru, pętle i skoki, z których można budować **instrukcje złożone**, ujmując ciąg instrukcji w nawiasy klamrowe `{ }`. Deklaracje, instrukcje wyrażeniowe i skoki, jak w C, muszą być zakończone średnikiem.

Instrukcja wyrażeniowa jest wyrażeniem; w szczególności może ono zawierać wywołania podprogramów i operatory przypisania. Wyrażenie może być puste, instrukcja wyrażeniowa z pustym wyrażeniem nic nie robi.

Instrukcje wyboru to znane z języka C instrukcje

`if` `(` <*warunek*> `)` <*instrukcja*>
`if` `(` <*warunek*> `)` <*instrukcja*> `else` <*instrukcja*>
`switch` `(` <*wyrażenie*> `)` `{` <*instrukcje do wyboru*> `}`

przy czym <*warunek*> musi być wyrażeniem logicznym (tj. typu `bool`)[11]. Wyrażenie w instrukcji przełącznika (`switch`) musi być całkowite. Instrukcje do wyboru są opatrzone etykietami postaci `case` <*wyrażenie stałe*>: albo `default:`, a całość działa tak jak instrukcja przełącznika w C.

Pętle mają trzy znane z C postaci

```
for ( <wyrażenie>; <warunek>; <wyrażenie> ) <instrukcja>
while ( <warunek> ) <instrukcja>
do <instrukcja> while ( <warunek> );
```

Warunek w każdym przypadku musi być wyrażeniem logicznym. Poza tym ograniczeniem powyższe instrukcje działają tak jak ich odpowiedniki w C.

Zagnieżdżanie instrukcji warunkowych i pętli może wymagać użycia nawiasów klamrowych, tak samo jak w C. Na przykład poniższe instrukcje, choćby nawet miały identyczne warunki i instrukcje składowe, robią co innego:

```
if ( <warunek> )
   while ( <warunek> )
     if ( <warunek> )
       <instrukcja>
else  <instrukcja>
```

```
if ( <warunek> ) {
  while ( <warunek> )
    if ( <warunek> )
      <instrukcja>
} else <instrukcja>
```

Instrukcje skoku to znane z C instrukcje `continue;`, `break;` i `return;` oraz nowa instrukcja `discard;`. Nie ma instrukcji skoku `goto`.

Instrukcja `continue;` może wystąpić w pętli. Jej wykonanie pomija instrukcje do końca najbardziej wewnętrznej pętli, w której ta instrukcja się znajduje. Zależnie od warunku sterującego wykonaniem tej pętli następuje potem kolejna iteracja albo pętla kończy działanie.

Instrukcja `break;` kończy działanie najbardziej wewnętrznej pętli lub instrukcji przełącznika.

Instrukcja `return;` albo `return` <*wyrażenie*>`;` kończy działanie podprogramu, ewentualnie przekazując wartość zawartego w niej wyrażenia jako wartość funkcji. Wystąpienie instrukcji `return;` w procedurze `main` szadera kończy jego działanie.

Instrukcja `discard;`, dopuszczalna tylko w szaderach fragmentów, kończy działanie szadera i powoduje odstąpienie od dalszego przetwarzania fragmentu — po jej wykonaniu odpowiedni piksel w buforze obrazu pozostanie niezmieniony.

9.7. Podprogramy

Tak jak w C, podprogramy w GLSL-u są oficjalnie nazywane *funkcjami*, co może gmatwać opis, w którym słowo „funkcja" jest używane także (albo przede wszystkim) w znaczeniu przyjętym w matematyce. Dlatego będę raczej używał słów „podprogram" lub „procedura", rezerwując słowo „funkcja" do tych podprogramów, które przekazują wynik obliczenia przez

[11] Inaczej niż w C, gdzie dopuszczalne są także warunki opisane przez wyrażenia liczbowe i wskaźnikowe.

swoją nazwę, przy czym wynik ten zależy tylko od parametrów, a podprogram nie ma efektów ubocznych.

Podprogram składa się z **nagłówka** i **bloku** (instrukcji złożonej, tj. ciągu instrukcji zamkniętego w nawiasach klamrowych). Nagłówek, po którym następuje średnik *zamiast* bloku, jest **prototypem** podprogramu. Prototyp zawiera informacje konieczne i wystarczające do wygenerowania przez kompilator ciągu rozkazów przekazujących parametry i wywołujących podprogram, który może być częścią innego szadera (tego samego etapu); ten mechanizm jest podobny do mechanizmu znanego z C. Zatem, w tekście źródłowym szadera przed pierwszą instrukcją wywołującą podprogram musi wystąpić albo ten podprogram, albo jego prototyp (oczywiście, nie dotyczy to podprogramów wbudowanych GLSL-a).

Ogólna postać podprogramu to

```
<typ wyniku> nazwa ( <param_1>, ..., <param_n> )
{
  ....  /* jakieś instrukcje */
  return <wyrażenie>;
}
```

Typ wyniku *musi* być jawnie podany (nie ma domyślnego typu wyniku, jakim w C jest `int`) i może to być typ liczbowy, strukturalny lub tablicowy. Jeśli podprogram nie przekazuje wyniku obliczeń przez swoją nazwę w wywołaniu, to powinien mieć typ `void` i wtedy wyrażenie w instrukcji `return` musi być puste. W przeciwnym razie w instrukcji `return` musi być wyrażenie typu identycznego z typem wyniku podanym w nagłówku lub typu umożliwiającego niejawną konwersję (np. `int` do `float`).

Lista parametrów może być pusta (między nawiasami okrągłymi można nic nie pisać lub napisać słowo kluczowe `void`) albo może zawierać parametry. Każdy parametr jest zmienną, która ma **nazwę** poprzedzoną **typem parametru**, przed którym *można* podać **kwalifikator parametru**. Jest on słowem kluczowym `in`, `out` lub `inout`. Przed kwalifikatorem `in` można podać dodatkowy kwalifikator `const`, który nie dopuszcza zmieniania wartości parametru przez instrukcje w podprogramie, mogą też być użyte **kwalifikatory precyzji**, których opis pominę.

Parametry podprogramu są jego zmiennymi lokalnymi[12]; **parametry wejściowe** (zadeklarowane bez kwalifikatora lub opatrzone kwalifikatorem `in`) otrzymują wartości początkowe podczas wywołania podprogramu. Wartości **parametrów wyjściowych** (z kwalifikatorem `out`) są w trakcie powrotu z podprogramu kopiowane do zmiennych przekazanych jako parametry, a **parametry wejściowo-wyjściowe** (z kwalifikatorem `inout`) podlegają odpowiednim operacjom i na początku i na końcu działania podprogramu.

Działanie podprogramu kończy instrukcja `return` (lub `discard`), ale jeśli takiej instrukcji autor podprogramu nie napisał, to powrót z podprogramu następuje po wykonaniu ostatniej instrukcji w jego bloku. Jeśli jednak podprogram przekazuje wynik (typu innego niż `void`) przez swoją nazwę, to trzeba napisać instrukcję `return` z wyrażeniem, którego wartość jest tym wynikiem.

[12] Dotyczy to także tablic, inaczej niż w języku C, w którym parametry tablicowe są wskaźnikami początków przekazanych tablic.

Nazwy podprogramów mogą być *przeciążane*, tj. może istnieć wiele podprogramów o tej samej nazwie. Muszą one różnić się listami parametrów, tj. mieć inne liczby parametrów lub przynajmniej inne *typy* parametrów, tak aby kompilator mógł na podstawie listy parametrów wywołania podprogramu wybrać odpowiedni podprogram do wywołania.

Punktem wejściowym szadera jest podprogram o nazwie `main`. Jego nagłówek powinien mieć postać

```
void main ( void )
```

ponieważ wszelkie dane i wyniki szadery przekazują przez zmienne interfejsu.

Rekurencyjne wywoływanie podprogramów jest niedozwolone.

9.8. Zmienne wskazujące podprogramy

Zamiast instrukcji wyboru (dokonywanego np. na podstawie wartości jakiejś liczbowej lub logicznej zmiennej jednolitej) można do modyfikowania działania szadera używać jednolitych zmiennych wskazujących podprogramy. W tym celu trzeba zadeklarować **typ podprogramu**, co wygląda tak:

```
subroutine <typ wyniku> nazwa_typu ( <param_1>, ..., <param_n> );
```

Deklaracja wygląda zatem jak prototyp podprogramu poprzedzony słowem kluczowym `subroutine`. Podprogram, który ma być wskazywany, musi być poprzedzony słowem kluczowym `subroutine` i podaną w nawiasach nazwą typu zadeklarowanego jak wyżej (lub listą nazw typów, oddzielonych przecinkami), przy czym typ wyniku i lista parametrów podprogramu muszą być takie same jak typ wyniku i lista parametrów w deklaracji typu podprogramu. Czyli na przykład

```
subroutine (nazwa_typu)
<typ wyniku> nazwa ( <param_1>, ..., <param_n> )
{
  ....  /* jakieś instrukcje */
  return <wyrażenie>;
}
```

Wreszcie zmienne wskazujące podprogram muszą być globalne i jednolite, zadeklarowane w taki sposób:

```
subroutine uniform nazwa_typu nazwa_zmiennej;
```

Przypisanie wartości takiej zmiennej może wykonać *tylko* aplikacja działająca na CPU, za pomocą procedur `glGetSubroutineUniformLocation`, która podaje identyfikator położenia zmiennej wskazującej, `glGetSubroutineIndex`, która podaje tzw. **indeks podprogramu**[13], i `glUniformSubroutinesuiv`, która dokonuje przypisania odpowiednich wartości

[13] Te dwie procedury trzeba wywołać po skompilowaniu i połączeniu programu szaderów i zapamiętać podane przez nie informacje.

zmiennym wskazującym. Składnia wywołania podprogramu wskazywanego jest taka jak składnia wywołania „zwykłego" podprogramu, ale zamiast nazwy podprogramu podaje się nazwę zmiennej wskazującej.

Można deklarować tablice zmiennych wskazujących podprogramy i w wywołaniu podawać indeks do tablicy, ale z uwagi na jednolitość obliczeń (zobacz niżej) wszystkie działające równolegle instancje szadera muszą w chwili wywołania podać ten sam indeks takiej tablicy. Sposób użycia wskaźników do procedur w GLSL-u jest opisany na przykładzie w p. 18.4.4.

9.9. Równoległość i jednolitość obliczeń

GPU składa się z wielu procesorów, które w odróżnieniu od poszczególnych rdzeni CPU nie pracują niezależnie. Przetwarzanie danych (np. dla wierzchołków lub fragmentów) odbywa się równolegle, przy czym poszczególne procesory GPU przetwarzają *różne dane*, wykonując jednocześnie *te same rozkazy*. Jeśli jednak poszczególne instancje szadera, po obliczeniu warunku w instrukcji wyboru (`if`, `if`–`else`, `switch`) albo warunku sterującego wykonaniem lub zakończeniem pętli (`while`, `do`–`while`, `for`), muszą wykonać różne instrukcje, to część procesorów (mających do wykonania te same instrukcje) działa dalej, podczas gdy pozostałe procesory czekają. Ma to oczywisty (i niekorzystny) wpływ na szybkość obliczeń.

Autorzy szaderów mogą podejmować wysiłki zmierzające do zminimalizowania kodu wykonywanego niejednolicie (np. usuwając z instrukcji wyboru wszelkie obliczenia wspólne dla wszystkich możliwych wyborów). Istotniejsze są ograniczenia dla konstrukcji językowych. To z tego powodu rekurencyjne wywoływanie podprogramów jest niedozwolone, a jeśli szader ma wykonać podprogram wskazywany przez zmienną będącą elementem tablicy, to wszystkie instancje szadera muszą obliczyć ten sam indeks do tej tablicy.

9.10. Bloki zmiennych interfejsu

Zmienne interfejsu służące do przekazywania informacji między szaderami a aplikacją działającą na CPU można grupować w **bloki interfejsu**. Deklaracja bloku interfejsu składa się z opcjonalnego **kwalifikatora układu** (`layout`), obowiązkowego **kwalifikatora interfejsu**, **nazwy zewnętrznej bloku**, **listy pól** w nawiasach klamrowych i opcjonalnej **nazwy instancji** (tj. prywatnej nazwy bloku dla szadera). Kwalifikator interfejsu jest słowem kluczowym `buffer`, `in`, `out` lub `uniform`, albo parą słów `patch in` lub `patch out`.

Dosyć liczne przykłady deklaracji bloku interfejsu spotkaliśmy już we wcześniejszych rozdziałach, zobacz na przykład listing 10.2, na którym są deklaracje bloków o nazwach (zewnętrznych) `Vertex` i `TransBlock`. Pierwszy z tych bloków służy do przekazania danych między etapami w potoku przetwarzania grafiki (z szadera wierzchołków do szadera geometrii — w aplikacji w rozdz. 10 nie ma szaderów rozdrabniania), a drugi jest blokiem zmiennych jednolitych. Bloki te mają prywatne nazwy (nazwy instancji) `Out` i `trb`. Szader do pól w tych blokach odwołuje się za pomocą operatora kropki następującego po nazwie instancji, na przykład `trb.pm`. Jeśli nie ma nazwy instancji, to w odwołaniach do pól szader nie podaje

także kropki, czyli gdyby deklaracja bloku miała postać

```
uniform TransBlock {
  mat4 mm, vm, pm;
};
```

to do jego pól `mm`, `vm`, `pm` szader odwoływałby się bez żadnych dodatkowych ceregieli. Ale nazwa instancji jest potrzebna, jeśli blok interfejsu jest tablicą. Po nazwie instancji podaje się wtedy nawiasy kwadratowe `[ ]`, ewentualnie obejmujące wyrażenie stałe (liczbę naturalną — długość tablicy). W odwołaniach do pól takiego bloku muszą być podane odpowiednie indeksy (na listingu 10.3 wygląda to tak: `In[i].Position`) [14].

9.11. Komunikacja między szaderami

Komunikacja między szaderami odbywa się za pomocą **zmiennych wbudowanych** i **bloków interfejsu**. Podstawowe dane są przekazywane w zmiennych wbudowanych, „gotowych do użycia" bez potrzeby deklarowania ich, a bloki interfejsu są wprowadzane przez autora szaderów, aby przekazywać wszelkie dane specyficzne dla konkretnego programu szaderów.

Na wszystkich etapach części przedniej potoku przetwarzania grafiki występują zmienne strukturalne o nazwie zewnętrznej `gl_PerVertex`. To *nie jest* nazwa typu strukturalnego, tylko zewnętrzna nazwa zmiennej strukturalnej, która może nie mieć nazwy instancji, ale jeśli występuje jednocześnie jako zmienna wejściowa i wyjściowa (np. szadera rozdrabniania lub geometrii), to jedna lub druga zmienna `gl_PerVertex` ma nazwę instancji — co najmniej jedna z nich jest tablicą i wtedy ma nazwę instancji `gl_in` lub `gl_out`.

```
<kwalifikator> gl_PerVertex {
    vec4  gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
    float gl_CullDistance[];
  };
```

Pole `gl_Position` jest wektorem współrzędnych jednorodnych wierzchołka; jest to zapewne najważniejsza dana, którą szadery odczytują ze swoich zmiennych wejściowych i zapisują na wyjście. Pozostałe pola mają wartości domyślne, które szader może nadpisać, ale może też zignorować. Pole `gl_PointSize` jest średnicą (w pikselach) kropki[15], która zostanie narysowana, jeśli wierzchołek jest wyświetlany w trybie `GL_POINTS` (parametr określający tryb podaje się w wywołaniu procedur z rodziny `glDraw*`). Ale aby szader mógł nadawać wielkość kropki, przed przystąpieniem do rysowania trzeba wykonać instrukcję

[14] Jeśli nazwa instancji bloku zmiennych jednolitych lub bloku magazynowego jest podana, to nazwy pól w tym bloku w tablicy symboli programu, skąd aplikacja może odczytać przesunięcia pól, są prefiksowane nazwą *zewnętrzną* bloku, na przykład `TransBlock.mm`. Jeśli nazwa instancji jest nieobecna, to w tablicy symboli programu nazwy pól bloku występują bez prefiksu. Mogę wyobrazić sobie powód (ktoś kiedyś zrobił to niechlujnie i tak już zostało), ale nie jestem w stanie znaleźć usprawiedliwienia dla czegoś takiego.

[15] a ściślej jest to wysokość i szerokość kwadratu

```
glEnable ( GL_PROGRAM_POINT_SIZE );
```

Pola `gl_ClipDistance` i `gl_CullDistance` są tablicami, których długości są liczbami płaszczyzn obcinających i odrzucających. Płaszczyznami obcinającymi są płaszczyzny ścian kostki standardowej (których jest 6) i dodatkowe płaszczyzny obcinające wprowadzone przez aplikację. Płaszczyzny odrzucające służą do odrzucania całych prymitywów. Szczegółami obcinania i odrzucania zajmiemy się w p. 10.6.5.

W programach szaderów odwołujących się do tablic `gl_ClipDistance` i `gl_CullDistance` należy przedefiniować strukturę `gl_PerVertex`, podając jawnie długości tych tablic (w postaci wyrażenia stałego). Inne dopuszczalne modyfikacje polegają na pominięciu pól nieużywanych, natomiast nie wolno niczego dodać. Zmodyfikowana struktura musi być taka sama we wszystkich szaderach programu.

9.11.1. Zmienne wbudowane szadera wierzchołków

`in int gl_VertexID;` — numer wierzchołka rysowanego prymitywu, może mieć wartość nieokreśloną.

`in int gl_InstanceID;` — numer instancji prymitywu rysowanego w wielu egzemplarzach (np. przez procedurę `glDrawArraysInstanced`) [16].

`out gl_PerVertex { .... };` — dane opisujące wierzchołek, obliczone przez szader.

9.11.2. Zmienne wbudowane szadera sterowania rozdrabnianiem

`in gl_PerVertex { .... } gl_in[gl_MaxPatchVertices];` — tablica wierzchołków płata dostarczonych przez szader wierzchołków.

`in int gl_PatchVerticesIn;` — liczba wierzchołków płata (tj. faktyczna długość tablicy `gl_in`).

`in int gl_PrimitiveID;` — wartość tej zmiennej jest liczbą prymitywów elementarnych (płatów) przetworzonych od początku bieżącego zbioru prymitywów.

`in int gl_InvocationID;` — numer wierzchołka w obrębie płata, od zera do liczby o 1 mniejszej niż liczba wierzchołków płata.

`out gl_PerVertex { .... } gl_out[];` — tablica, do której (jednego elementu) szader ma przypisać dane wyjściowe opisujące wierzchołek.

`patch out float gl_TessLevelOuter[4];` — poziomy rozdrabniania brzegu płata.

`patch out float gl_TessLevelInner[2];` — poziomy rozdrabniania wewnątrz płata.

Szader sterowania rozdrabnianiem ma za zadanie przekazać na wyjście dane opisujące *jeden* wierzchołek (o numerze `gl_InvocationID`) i wypełnić odpowiednimi danymi tablice

[16] W szaderach przeznaczonych do pracy z aplikacjami Vulkana zmienne `gl_VertexID` i `gl_InstanceID` zostały zastąpione przez nowe zmienne `gl_VertexIndex` i `gl_InstanceIndex` (zobacz specyfikację [4]).

`gl_TessLevelOuter` i `gl_TessLevelInner`, ale to ma zrobić *tylko jedna* instancja szadera — najprościej jest umieścić instrukcje przypisania do tych tablic w instrukcji warunkowej

```
if ( gl_InvocationID == 0 ) { .... }
```

9.11.3. Zmienne wbudowane szadera rozdrabniania

`in gl_PerVertex { .... } gl_in[gl_MaxPatchVertices];` — tablica wierzchołków płata dostarczonych przez szader sterowania rozdrabnianiem (jeśli jest obecny w programie) lub przez szader wierzchołków.

`in int gl_PatchVerticesIn;` — liczba wierzchołków płata (tj. długość tablicy `gl_in`).

`in int gl_PrimitiveID;` — ma tę samą wartość co dla szadera sterowania rozdrabnianiem.

`in vec3 gl_TessCoord;` — wektor współrzędnych kartezjańskich lub barycentrycznych punktu *w dziedzinie płata* otrzymanego w etapie rozdrabniania.

`patch in float gl_TessLevelOuter[4];` — poziomy rozdrabniania brzegu płata.

`patch in float gl_TessLevelInner[2];` — poziomy rozdrabniania wewnątrz płata.

`out gl_PerVertex { .... };` — opis wytworzonego przez szader wierzchołka płata (tj. końca odcinka lub wierzchołka trójkąta otrzymanego w wyniku rozdrabniania).

9.11.4. Zmienne wbudowane szadera geometrii

`in gl_PerVertex { .... } gl_in[];` — tablica wierzchołków prymitywu elementarnego (pojedynczego wierzchołka, końców odcinka lub wierzchołków trójkąta).

`in int gl_PrimitiveIDIn;` — wartość tej zmiennej jest liczbą prymitywów elementarnych (punktów, odcinków, trójkątów lub płatów) przetworzonych od początku bieżącego zbioru prymitywów, czyli jest to na przykład numer trójkąta w taśmie trójkątowej (licząc od 0). Zmienna ta ma tę samą wartość co zmienna `gl_PrimitiveID` szadera rozdrabniania, jeśli taki szader jest obecny w potoku przetwarzania grafiki.

`in int gl_InvocationID;` — numer instancji szadera dla przetwarzanego prymitywu. Jeśli wejście szadera ma kwalifikator `layout(invocations=`n`)` (zobacz podrozdz. 9.12), to poszczególne instancje są ponumerowane od 0 do $n-1$. Szader geometrii może każdy „egzemplarz” prymitywu przetworzyć w inny sposób.

`out gl_PerVertex { .... };` — opis wytworzonego przez szader wierzchołka prymitywu.

`out int gl_PrimitiveID;` — identyfikator prymitywu, przekazywany na wejście szadera fragmentów w jego zmiennej wejściowej `gl_PrimitiveID`. Szader geometrii może podzielić odcinek lub trójkąt na mniejsze kawałki i dowolnie je ponumerować; zmienna `gl_PrimitiveID` umożliwia przekazanie do szadera fragmentów numerów tych kawałków.

`out int gl_Layer;` — numer warstwy, może służyć m.in. do wyboru ściany sześcianu utworzonego z tekstur.

`out int gl_ViewportIndex;` — numer klatki, tj. obszaru na obrazie, na którym prymityw ma być narysowany (domyślnie otrzymuje wartość 0).

Szader geometrii otrzymuje tablicę wierzchołków prymitywu (z dwoma końcami odcinka lub z trzema wierzchołkami trójkąta) i ma wygenerować odpowiedni ciąg wierzchołków. Zamiast wpisywać je do tablicy (tak jak szader sterowania rozdrabnianiem) szader geometrii ma kolejno dla każdego wierzchołka wpisać dane do pól struktury wyjściowej `gl_PerVertex` (i w razie potrzeby do swojego wyjściowego bloku interfejsu z danymi dodatkowymi), a następnie wywołać procedurę `EmitVertex`. Na zakończenie powinien wywołać procedurę `EndPrimitive`.

Szader geometrii może rozdrobnić otrzymany na wejściu odcinek lub trójkąt, dzieląc go na kilka odcinków lub trójkątów i wyprowadzając do etapu obcinania łamane lub taśmy trójkątowe. Szader może też niczego nie wyprowadzić, co oznacza zaniechanie rysowania otrzymanej na wejściu figury.

9.11.5. Zmienne wbudowane szadera fragmentów

`in vec4 gl_FragCoord;` — wektor $(\xi, \eta, \zeta, 1/W)$ współrzędnych fragmentu w oknie. Liczby ξ i η określają punkt na obrazie, ζ jest głębokością otrzymanego w etapie rasteryzacji punktu, któremu odpowiada dany fragment. Liczba ta jest związana ze współrzędną z w układzie kostki standardowej wzorem $\zeta = (z+1)/2$, jeśli więc $z \in [-1,1]$, to $\zeta \in [0,1]$. Liczba W jest współrzędną wagową wektora współrzędnych jednorodnych.

`in bool gl_FrontFacing;` — ma wartość `true`, jeśli prymityw jest odwrócony przodem do obserwatora, i `false` w przeciwnym razie.

`in float gl_ClipDistance[];` — tablica odległości punktu od płaszczyzn obcinania.

`in float gl_CullDistance[];` — tablica odległości punktu od płaszczyzn odrzucania.

`in vec2 gl_PointCoord;` — dotyczy wierzchołków rysowanych jako punkty (w trybie `GL_POINTS`). Jego współrzędne przyjmują wartości od 0 do 1 dla punktów w obszarze (kwadratowej) kropki. Można na ich podstawie, korzystając z instrukcji `discard`, nadać kropce kształt inny niż kwadratowy.

`in int gl_PrimitiveID;` — identyfikator prymitywu, o wartości nadanej przez szader geometrii (przypisanej *jego* zmiennej *wyjściowej* `gl_PrimitiveID`).

`in int gl_Layer;` — numer warstwy nadany przez szader geometrii (ma wartość nieokreśloną, jeśli to nie nastąpiło)

`in int gl_ViewportIndex;` — numer klatki nadany przez szader geometrii (0, jeśli to nie nastąpiło).

`in bool gl_HelperInvocation;` — ma wartość `true` podczas **wywołania pomocniczego** i `false` kiedy indziej. Wywołania pomocnicze mogą być wykonywane w celu obliczenia gradientu koloru (albo tekstury).

`in int gl_SampleID;` — numer próbki w technice **wielokrotnego próbkowania** (*multisampling*) używanej do antyaliasingu.

`in vec2 gl_SamplePosition;` — położenie próbki w pikselu.

`in int gl_SampleMaskIn[];` — pole bitowe, które opisuje zbiór próbek pokrytych przez prymityw geometryczny, którego fragment jest przetwarzany.

`out int gl_SampleMask[];` — pole bitowe, które opisuje zbiór próbek pokrytych przez prymityw geometryczny, którego fragment jest przetwarzany. Szader fragmentów może zmodyfikować pole dane na wejściu w tablicy `gl_SampleMaskIn`.

`out float gl_FragDepth;` — głębokość punktu, używana dalej w testach widoczności. Szader fragmentów może zmodyfikować jej wartość, wpływając w ten sposób na wynik testu.

Wiele zmiennych interfejsu szadera fragmentów jest związanych z antyaliasingiem, tj. poprawianiem jakości obrazu przez „wygładzanie" ząbkowanych krawędzi na obrazie rastrowym. Najważniejszym zadaniem szadera fragmentów jest obliczenie koloru fragmentu, przy czym, co ciekawe, trzeba w tym celu jawnie zadeklarować zmienną wyjściową typu `vec4` (nie ma zmiennej wbudowanej służącej do tego celu). Współrzędne `r,g,b,a` tej zmiennej muszą mieć wartości z przedziału $[0,1]$ i opisują odpowiednio składowe czerwoną, zieloną i niebieską oraz **składową alfa**. Ta ostatnia jest parametrem dla ostatniego etapu w potoku przetwarzania grafiki, w którym (po przejściu fragmentu przez test widoczności) następuje końcowe obliczenie koloru przypisywanego pikselowi. Zależnie od przyjętej **funkcji mieszającej** w tym obliczeniu może być uwzględniony dotychczasowy kolor piksela (i wartości składowej alfa koloru dotychczasowego i koloru przekazanego przez szader).

Zmienne wbudowane szaderów obliczeniowych są opisane dalej (w podrozdz. 9.15).

9.12. Kwalifikatory układu zmiennych

W opisie zmiennych w GLSL-u **kwalifikatory układu** (*layout qualifiers*) spełniają dwie role: wpływają na sposób rozmieszczania składowych (np. pól struktury) w pamięci (czyli na przydzielanie tym składowym adresów przez kompilator GLSL-a), a także (dla zmiennych interfejsu) zawierają informacje umożliwiające właściwe zorganizowanie przepływu danych między etapami potoku przetwarzania grafiki. Kwalifikator układu może występować osobno, może poprzedzać zmienną, blok (tj. strukturę) lub indywidualne składowe takiego bloku. Ma on ogólną postać

`layout`(<*lista identyfikatorów kwalifikatora*>)

przy czym <*lista identyfikatorów kwalifikatora*> to jeden lub kilka (przedzielonych przecinkami) napisów postaci <*nazwa*> lub <*nazwa*>=<*wartość*>. Nie da się pokrótce opisać wszystkich nazw, ich znaczenia ani możliwych wartości (w konkretnych miejscach) i nie warto wkuwać na pamięć wszystkiego na ten temat, zwłaszcza „na sucho", tj. w oderwaniu od zastosowań. Dlatego poniżej przedstawiłem tylko kilka najważniejszych przykładów.

Dla **zmiennych jednolitych** możemy podać kwalifikator `shared` (domyślny), `std140` lub `packed`. Osobny kwalifikator ma na przykład postać

```
layout(std140) uniform;
```

i występujące za nim zmienne jednolite będą kompilowane w takim układzie, z wyjątkiem zmiennych indywidualnie opatrzonych innym kwalifikatorem układu, na przykład

```
layout(shared) uniform MojaZmienna { .... } jednolita;
```

Kwalifikator `packed` jest najbardziej oszczędny pod względem przydziału pamięci, ale nie może być używany dla bloków zmiennych jednolitych, do których dostęp ma więcej niż jeden szader (nawet w tym samym programie szaderów)[17].

Domyślny kwalifikator `shared` jest zależny od implementacji, ale zapewnia zgodność układu opisanych tak samo zmiennych w różnych (oddzielnie kompilowanych) szaderach i jest dosyć oszczędny, m.in. nie zostawia pustych miejsc między elementami tablic liczb całkowitych.

Kwalifikator układu `std140`[18] wyrównuje adres każdego elementu tablicy do wartości podzielnej przez 4 razy rozmiar skalarnej składowej elementu; na przykład tablica liczb typu `int` albo `bool` w układzie `std140` dla każdego elementu rezerwuje 16 bajtów, z których tylko 4 będą przechowywać odpowiednią liczbę. Dokładne reguły przydziału adresów w tym układzie można znaleźć w specyfikacji [3] lub w książkach [24] i [23]. Dają one możliwość obliczenia przesunięć poszczególnych elementów przez aplikację w C (albo przez jej autora) i uniknięcia korzystania z procedury `glGetActiveUniformsiv` (zobacz listing 7.4 i jego opis). W szczególności można napisać takie definicje typów strukturalnych w C (z nieużywanymi polami powodującymi odpowiednie modyfikacje położeń pól używanych), aby przesunięcia potrzebnych pól względem początku struktury w C były identyczne z przesunięciami odpowiadających im pól względem początku bufora w pamięci GPU, co umożliwia przesyłanie zawartości wielu pól za pomocą *jednego* wywołania procedury `glBufferData` albo `glBufferSubData`[19].

Kwalifikatory układu **zmiennych intefejsu** oprócz rozmieszczenia pól struktur w pamięci podają informacje potrzebne do dopasowania wejścia i wyjścia szaderów do innych etapów potoku przetwarzania grafiki. Poniżej opisuję tylko niektóre kwalifikatory zmiennych interfejsu.

[17] Przyczyna jest taka, że w ramach optymalizacji kompilator może usunąć pola, do których szader się nie odwołuje. Jeśli więc dwa różne szadery odwołują się do różnych pól, to programu nie da się poprawnie połączyć.

[18] Nazwa jest związana z wersją języka GLSL, w której ten układ został wprowadzony; to samo dotyczy nazwy bardziej upakowanego układu `std430`, który może być stosowany dla buforów magazynowych, ale nie dla bloków zmiennych jednolitych.

[19] Nie jestem przekonany, czy ta alternatywa jest wygodniejsza. Na pewno jest bardziej podatna na błędy, a marnowanie pamięci w układzie `std140` (motywowane osiąganiem jak najszybszego dostępu do danych) może być duże — zwłaszcza dla tablicy elementów typu `bool`, gdzie każdy znaczący bit ma do towarzystwa 127 bitów nieużywanych. Ale można, przynajmniej teoretycznie, napisać translator, który po przeczytaniu opisu struktury w GLSL-u obliczy przesunięcia poszczególnych pól i wygeneruje tekst źródłowy z deklaracją odpowiedniej struktury w C.

Zmienne przekazywane przez etap pobierania wierzchołków do **szadera wierzchołków** muszą mieć **kwalifikator miejsca**

```
layout(location=n)
```

(przykład był na listingu 7.1), w którym wyrażenie stałe *n* oznacza **numer miejsca** zajmowanego przez atrybut; każdy atrybut skalarny lub wektorowy zajmuje jedno miejsce (możemy więc numerować miejsca 0, 1, 2 itd.). Wyjątkiem są atrybuty typu `dvec3` i `dvec4`, które zajmują *dwa kolejne* miejsca. Zmienne z atrybutami wierzchołków mogą być tablicami i wtedy zajmują odpowiednio więcej miejsc (tyle, ile tablica ma elementów), zaczynając od miejsca podanego w kwalifikatorze.

Szader sterowania rozdrabnianiem może mieć *tylko* kwalifikator układu wyjścia, postaci

```
layout(vertices=n) out;
```

z wyrażeniem stałym *n* określającym liczbę wierzchołków płata, będącą też długością tablicy `gl_out` (zobacz opis wbudowanych zmiennych interfejsu) i liczbą wywołań szadera — po jednym razie dla każdego wierzchołka.

Wejście do **szadera rozdrabniania** powinno mieć jeden z następujących kwalifikatorów układu: `triangles`, `quads`, `isolines`. Dodatkowo można podać kwalifikatory `equal_spacing`, `fractional_even_spacing` albo `fractional_odd_spacing` oraz `cw`, `ccw` albo `point_mode`. Pierwsze trzy kwalifikatory określają kształt dziedziny rozdrabnianego płata (trójkąt, kwadrat albo rodzina linii równoległych w kwadracie). Kolejne trzy kwalifikatory sterują rozmieszczeniem w tej dziedzinie wierzchołków generowanych przez etap rozdrabniania dziedziny. Ostatnie trzy wybierają orientację trójkątów wytwarzanych przez ten etap (zgodnie z ruchem wskazówek zegara albo przeciwnie) lub tryb, w którym przekazywane są tylko osobne wierzchołki. Możemy napisać na przykład

```
layout(triangles,cw,equal_spacing) in;
```

Przykłady wyjaśniające, jak to działa, są w opisach aplikacji pierwszej D i drugiej (rozdz. 12 i 15).

Wejście **szadera geometrii** ma mieć jeden z następujących kwalifikatorów: `points`, `lines`, `lines_adjacency`, `triangles`, `triangles_adjacency`. Kwalifikatory te wybierają rodzaj prymitywu przetwarzanego przez szader geometrii, punkty, odcinki lub trójkąty, przy czym kwalifikatory „`adjacency`” powodują dostarczanie także wierzchołków sąsiednich odcinków lub trójkątów. Dodatkowo może być kwalifikator `invocations=n`, gdzie *n* jest liczbą wywołań szadera geometrii dla każdego prymitywu (domyślnie jest jedno wywołanie). Możemy zatem napisać na przykład

```
layout(triangles,invocations=4) in;
```

Spowoduje to powstanie z każdego trójkąta czterech jego kopii opisanych przez identyczne dane wejściowe. Numer kopii, od 0 do 3, podany w zmiennej `gl_InvocationID` może posłużyć do wybrania innego przekształcenia dla wierzchołków każdej kopii, można też każdą kopię inaczej podzielić na taśmy trójkątowe lub inne prymitywy wyjściowe.

Wyjście szadera geometrii może mieć jeden z kwalifikatorów `points`, `line_strip` albo `triangle_strip`, a ponadto musi mieć kwalifikator `max_vertices=`*n*, określający maksymalną liczbę wierzchołków wyjściowej łamanej lub taśmy trójkątowej — przykład będzie w opisie aplikacji pierwszej B (rozdz. 10).

Zmienna wejściowa `gl_FragCoord` **szadera fragmentów** może być przedeklarowana z dodatkiem kwalifikatora `origin_upper_left` i/lub `pixel_center_integer`, na przykład

```
layout(origin_upper_left) in vec4 gl_FragCoord;
```

Deklaracja ta zmienia domyślny układ współrzędnych w oknie (z początkiem w dolnym lewym narożniku i osią *y* skierowaną do góry) na układ odwrócony (z początkiem w górnym lewym narożniku i osią *y* skierowaną do dołu)[20]. Kwalifikator `pixel_center_integer` przesuwa początek układu w poziomie i pionie o połowę szerokości i wysokości piksela, wskutek czego środki pikseli mają współrzędne całkowite zamiast liczb o częściach ułamkowych równych 0.5.

Zmienna wyjściowa `gl_FragDepth` szadera fragmentów może być przedeklarowana z kwalifikatorem `depth_any` (domyślny), `depth_greater`, `depth_less` albo `depth_unchanged`, na przykład

```
layout(depth_unchanged) out float gl_FragDepth;
```

Używanie tych kwalifikatorów ma związek z optymalizacją obliczeń w powiązaniu z testami widoczności; jeśli dany fragment nie jest widoczny (bo jest zasłonięty przez pewien fragment narysowany wcześniej), to zazwyczaj nie warto tracić czasu na skomplikowane obliczenia teksturowania i oświetlenia tego fragmentu.

Opis pozostałych kwalifikatorów można znaleźć w dokumentacji [3], do czego zachęcam w miarę nabywania doświadczenia i rosnących potrzeb.

9.13. Funkcje i procedury wbudowane

Język GLSL dysponuje bogatą biblioteką podprogramów „gotowych" do wywoływania przez szadery. Nazwy tych podprogramów są przeciążone, tj. jednej nazwie zazwyczaj odpowiada wiele podprogramów, które spełniają podobną rolę, ale mają różne parametry. Na przykład nazwę `sin` mają funkcje obliczające wartości funkcji sinus, których parametry są typu `float`, `vec2`, `vec3` lub `vec4`. Każda z tych funkcji oblicza sinusy, odpowiednio jednej, dwóch, trzech lub czterech liczb — współrzędnych wektora podanego jako parametr, przy czym wartość tej funkcji jest tego samego typu co parametr. Podobnie funkcje `dot` obliczają iloczyny skalarne wektorów, odpowiednio typu `vec2`, `vec3` lub `vec4`. Poniżej wymieniam tylko niektóre funkcje wbudowane. Ich pełną listę ze szczegółowymi opisami można znaleźć w dokumencie [3].

[20] Ta konstrukcja języka GLSL nie ma odpowiednika w SPIR-V, należy jej zatem unikać w szaderach, które mają być skompilowane do tej postaci.

Aby łącznie opisać funkcje o tej samej nazwie, mające parametry o różnych typach, użyłem oznaczeń *genType*, *genIType*, *genUType* i *genDType*. **Uwaga:** oznaczenia stosowane w oficjalnych specyfikacjach są podobne, ale nie identyczne z moimi. Pierwsze z nich oznacza typ skalarny `float` lub dowolny z typów wektorowych `vec2`, `vec3` lub `vec4`, a pozostałe podobnie, odpowiednio typ skalarny lub wektorowy ze składowymi całkowitymi ze znakiem (`int`), całkowitymi bez znaku (`uint`) lub zmiennopozycyjnymi podwójnej precyzji (`double`). Symbol *genBType* oznacza typ `bvec2`, `bvec3` lub `bvec4`. Symbole *mat* i *dmat* oznaczają typy macierzowe pojedynczej i podwójnej precyzji, a *mtype* oznacza wszystkie typy macierzowe. Typ wartości funkcji, *jeśli nie napisałem inaczej*, jest taki sam jak typ parametru.

9.13.1. Funkcje elementarne

```
type abs ( type x );
type sign ( type x );
```

Funkcje obliczające wartości bezwzględne i znaki składowych parametru. Symbol *type* oznacza *genType*, *genIType* lub *genDType*.

```
type floor ( type x );
type trunc ( type x );
type round ( type x );
type ceil ( type x );
type fract ( type x );
```

Funkcje obliczające zaokrąglenia liczb rzeczywistych do liczb całkowitych, odpowiednio w dół, w stronę zera, do najbliższej i w górę, oraz część ułamkową. Symbol *type* oznacza *genType* lub *genDType*.

```
type roundEven ( type x );
```

Zaokrąglanie do najbliższej liczby parzystej. Symbol *type* oznacza *genType* lub *genDType*.

```
type mod ( type x, type y );
```

Reszta z dzielenia, $x - y * \lfloor x/y \rfloor$. Symbol *type* oznacza *genType* lub *genDType*, przy czym drugi parametr może być wektorem o tylu składowych co pierwszy, lub skalarem (typu `float` albo `double`, takiego jak typ składowych pierwszego parametru).

```
type modf ( type x, out type i );
```

Funkcja oblicza część ułamkową (pierwszego) parametru wejściowego, a parametr wyjściowy (drugi) otrzymuje wartość równą części całkowitej. Symbol *type* oznacza *genType* lub *genDType*.

```
type min ( type x, type y );
type max ( type x, type y );
```

Wybór mniejszego i większego argumentu. Symbol *type* oznacza *genType*, *genDType*, *genIType* lub *genUType*.

```
type clamp ( type x, type minval, type maxval );
```

Funkcja oblicza najbliższy wartości pierwszego parametru element przedziału określonego przez drugi i trzeci parametr.

```
type mix ( type x, type y, type a );
```

Funkcja dokonuje interpolacji afinicznej między pierwszym a drugim parametrem. Symbol *type* oznacza *genType* lub *genDType*. Trzeci parametr może być wektorem (o tylu składowych co dwa pierwsze) lub skalarem.

```
type step ( type a, type x );
type smootshtep ( type a0, type a1, type x );
```

Funkcja `step` ma wartość 0, jeśli parametr `x` ma wartość mniejszą niż `a`, oraz 1 w przeciwnym razie. Wartością funkcji `smoothstep` jest 0 jeśli $x \leqslant a_0$, 1 jeśli $x \geqslant a_1$ oraz $3t^2 - 2t^3$, gdzie $t = (x - a_0)/(a_1 - a_0)$, jeśli $a_0 < x < a_1$. Symbol *type* oznacza *genType* lub *genDType*. Ostatni parametr może być wektorem (o tylu składowych co poprzedni) lub skalarem.

```
genBType isnan ( type x );
genBType isinf ( type x );
```

Wartością tych funkcji jest `true`, jeśli wartość parametru `x` reprezentuje odpowiednio nieliczbę (*NaN, not a number*) lub nieskończoność, albo `false` w przeciwnym razie.

9.13.2. Funkcje związane z potęgowaniem

```
genType pow ( genType x, genType y );
```

Funkcja potęgowa, obliczająca x^y.

```
genType exp ( genType x );
genType exp2 ( genType x );
```

Funkcje obliczające e^x i 2^x.

```
genType log ( genType x );
genType log2 ( genType y );
```

Funkcje obliczające $\ln x$ i $\log_2 x$.

```
type sqrt ( type x );
type inversesqrt ( type x );
```

Funkcje obliczające $\sqrt{x}$ i $1/\sqrt{x}$. Symbol *type* oznacza *genType* albo *genDType*.

9.13.3. Funkcje geometryczne

```
stype length ( type x );
stype distance ( type p0, type p1 );
```

Obliczanie długości wektora i odległości punktów. Symbol *type* oznacza *genType* lub *genDType*, a *stype* to odpowiedni typ skalarny (`float` albo `double`).

```
stype dot ( type x, type y );
```

Obliczanie iloczynu skalarnego $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$. Symbol *type* oznacza *genType* lub *genDType*, a *stype* to odpowiedni typ skalarny (`float` albo `double`).

```
type cross ( type x, type y );
```

Obliczanie iloczynu wektorowego w $\mathbb{R}^3$, $\boldsymbol{x} \wedge \boldsymbol{y}$. Symbol *type* oznacza `vec3` albo `dvec3`.

```
type normalize ( type x );
```

Normalizacja wektora, tj. dzielenie go przez jego długość, czego wynikiem jest wektor jednostkowy. Symbol *type* oznacza *genType* albo *genDType*.

```
type faceforward ( type N, type L, type Nref );
```

Funkcja oddaje pierwszy parametr ze zwrotem dobranym na podstawie iloczynu skalarnego wektorów podanych jako drugi i trzeci parametr ($\boldsymbol{n}$, jeśli $\langle \boldsymbol{l}, \boldsymbol{n}_{\text{ref}} \rangle < 0$, lub $-\boldsymbol{n}$ w przeciwnym razie). Symbol *type* oznacza *genType* albo *genDType*.

```
type reflect ( type L, type N );
```

Funkcja znajdująca odbicie symetryczne wektora $\boldsymbol{l}$ w płaszczyźnie prostopadłej do wektora jednostkowego $\boldsymbol{n}$, opisane wzorem $\boldsymbol{l} - 2\langle \boldsymbol{n}, \boldsymbol{l} \rangle \boldsymbol{n}$ (zobacz podrozdz. 5.5). Symbol *type* oznacza *genType* albo *genDType*.

```
type refract ( type I, type N, float eta );
```

Funkcja znajdująca wektor $\boldsymbol{r}$ o kierunku załamania światła na granicy ośrodków o różnych współczynnikach załamania światła. Parametry `I`, `N` i `eta` reprezentują wektor kierunku padania światła $\boldsymbol{l}$, wektor normalny powierzchni $\boldsymbol{n}$ i iloraz współczynników załamania światła ośrodków η. Funkcja realizuje następujące obliczenie:

$$k = 1 - \eta^2\left(1 - \langle \boldsymbol{n}, \boldsymbol{l} \rangle^2\right),$$
$$\boldsymbol{r} = \begin{cases} \boldsymbol{0} & \text{jeśli } k < 0, \\ \eta \boldsymbol{l} - \left(\eta \langle \boldsymbol{n}, \boldsymbol{l} \rangle + \sqrt{k}\right)\boldsymbol{n} & \text{w przeciwnym razie.} \end{cases} \tag{9.1}$$

Symbol *type* oznacza *genType* albo *genDType*.

Uwaga: Aby wynik obliczony na podstawie wzoru (9.1) był poprawny, należy podać wektor $\boldsymbol{n}$ z takim zwrotem, dla którego iloczyn skalarny $\langle \boldsymbol{n}, \boldsymbol{l} \rangle$ jest niedodatni. Specyfikacje [3] ani [4] o tym nie wspominają. Zobacz wyprowadzenie wzoru w podrozdziale A.1.

9.13.4. Funkcje związane z kątami

```
genType sin ( genType x );
genType cos ( genType x );
genType tan ( genType x );
```

Funkcje obliczające sinusy, kosinusy i tangensy współrzędnych wektorów podanych jako parametry. Współrzędne te są miarami kątów podanymi w radianach.

```
genType asin ( genType x );
genType acos ( genType x );
genType atan ( genType x );
genType atan ( genType y, genType x );
```

Funkcje obliczające miary kątów (w radianach), których sinusy, kosinusy lub tangensy są współrzędnymi wektorów podanych jako parametry. Funkcja `atan` z dwoma parametrami oblicza arkus tangens ilorazu y/x z uwzględnieniem zwrotu wektora (x, y) — to jest odpowiednik funkcji `atan2` dostępnej dla programów w C w bibliotece `libm`.

```
genType sinh ( genType x );
genType cosh ( genType x );
genType tanh ( genType x );
```

Funkcje obliczające odpowiednio sinus, kosinus i tangens hiperboliczny, zdefiniowane za pomocą wzorów $\sinh x = (e^x - e^{-x})/2$, $\cosh x = (e^x + e^{-x})/2$, $\operatorname{tgh} x = \sinh x / \cosh x$.

```
genType asinh ( genType x );
genType acosh ( genType x );
genType atanh ( genType x );
```

Funkcje obliczające wartości funkcji odwrotnych do funkcji hiperbolicznych.

```
genType radians ( genType d );
genType degrees ( genType r );
```

Funkcje te mnożą swoje argumenty odpowiednio przez $\frac{\pi}{180}$ oraz $\frac{180}{\pi}$, co jest równoznaczne z przejściami od stopni do radianów i w drugą stronę.

9.13.5. Funkcje macierzowe

```
mtype matrixCompMult ( mtype x, mtype y );
```

Funkcja oblicza macierz iloczynów współczynników dwóch macierzy na odpowiadających sobie miejscach. Symbol *mtype* oznacza typ macierzowy. Oba parametry i wynik mają ten sam typ.

```
mtype outerProduct ( type x, type y );
```

Funkcja oblicza macierz $\boldsymbol{x}\boldsymbol{y}^T$. Symbol *type* oznacza *genType* albo *genDType*. Parametry są wektorami o m i n współrzędnych (ma być $m, n \in \{2, 3, 4\}$, przy czym w ogólności może być $m = n$ lub $m \neq n$), a wynik jest macierzą $m \times n$, której składowe są typu takiego jak składowe wektorów podanych jako parametry (`float` albo `double`).

```
mtype transpose ( mtype x );
```

Funkcja wyznacza macierz transponowaną do danej. Jeśli macierz dana ma wymiary $m \times n$, to wynik jest macierzą $n \times m$.

stype `determinant ( mtype x );`

Funkcja oblicza wyznacznik macierzy kwadratowej. Typ wyniku (*stype*) to `float` albo `double`, odpowiadający reprezentacji współczynników macierzy.

mtype `inverse ( mtype x );`

Funkcja oblicza odwrotność macierzy kwadratowej nieosobliwej. Symbol *mtype* oznacza `mat2`, `mat3`, `mat4`, `dmat2`, `dmat3` lub `dmat4`.

9.13.6. Funkcje relacji wektorowych

genBType `lessThan ( genType x, genType y );`
genBType `lessThanEqual ( genType x, genType y );`
genBType `greaterThan ( genType x, genType y );`
genBType `greaterThanEqual ( genType x, genType y );`
genBType `equal ( genType x, genType y );`
genBType `notEqual ( genType x, genType y );`

Funkcje porównują odpowiadające sobie współrzędne wektorów danych jako parametry i dostarczają wyniki porównań jako wartość typu wektorowego boolowskiego (`bvec2`, `bvec3` albo `bvec4`). Parametry funkcji muszą być tego samego liczbowego typu wektorowego. Relacje badane przez te funkcje to odpowiednio „mniejsze niż", „mniejsze lub równe", „większe niż", „większe lub równe", „równe" i „różne", co być może napisałem niepotrzebnie.

`bool any ( genBType x );`
`bool all ( genBType x );`

Funkcje mają wartość `true`, jeśli odpowiednio *co najmniej jedna* lub *wszystkie* współrzędne wektora boolowskiego podanego jako parametr mają wartość `true`, a w przeciwnym razie `false`.

genBType `not ( genBType x );`

Funkcja neguje współrzędne wektora boolowskiego.

9.13.7. Funkcje i procedury dla liczb całkowitych

W opisanych niżej funkcjach *type* oznacza *genIType* albo *genUType*.

genUType `uaddCarry ( genUType x, genUType y, out genUType carry );`
genUType `usubBorrow ( genUType x, genUType y, out genUType borrow );`

Funkcje realizują **dodawanie z przeniesieniem** (*carry*) i **odejmowanie z pożyczką** (*borrow*) liczb 32-bitowych bez znaku, umożliwiające zaimplementowanie arytmetyki dowolnie wielkich liczb (przechowywanych w wielu zmiennych typu `uint`).

```
void umulExtended ( genUType x, genUType y,
                    out genUType msb, out genUType lsb );
```

```
void imulExtended ( genIType x, genIType y,
                    out genIType msb, out genIType lsb );
```

Procedury wykonują mnożenie liczb 32-bitowych (bez i ze znakiem), dające wyniki 64-bitowe. Parametry wyjściowe `msb` i `lsb` przekazują bity wyniku na odpowiednio bardziej i mniej znaczących pozycjach.

```
type bitfieldExtract ( type value, int offset, int bits );
```

Funkcje „wycinające" wskazane bity z całkowitych współrzędnych wektora. Parametr `bits` określa liczbę potrzebnych bitów, a parametr `offset` wskazuje pozycję najmniej znaczącego z tych bitów.

```
type bitfieldInsert ( type base, type insert, int offset, int bits );
```

Funkcje wstawiające podane bity do całkowitych współrzędnych wektora na wskazaną pozycję. Parametr `base` jest wektorem, do którego współrzędnych należy wstawić bity z wektora `insert`, liczba tych bitów (dla każdej współrzędnej) i pozycja, na której ma się znaleźć najmniej znaczący bit, są określone przez parametry `bits` i `offset`.

```
type bitfieldReverse ( type value );
```

Funkcje odwracają odwracają kolejność bitów w każdej współrzędnej wektora `value`.

```
genIType bitCount ( type value );
genIType findLSB ( type value );
genIType findMSB ( type value );
```

Funkcje znajdujące odpowiednio liczbę niezerowych bitów oraz pozycję najmniej i najbardziej znaczącego niezerowego bitu.

9.13.8. Funkcje i procedury dla tekstur i obrazów

Zgodnie z wcześniejszym stwierdzeniem tekstura jest funkcją opisującą dowolną własność powierzchni wpływającą na wygląd tej powierzchni na obrazie. Podstawowa reprezentacja tekstury w OpenGL-u składa się z jednej lub wielu tablic wartości tej funkcji. Elementy tablic są nazywane **tekselami**. Szadery mogą odczytywać dane z tych tekseli bez żadnych dodatkowych obliczeń lub korzystać z **ewaluatorów tekstury** (*samplers*), czyli obiektów dokonujących interpolacji tekseli i filtrowania funkcji dla poprawienia jakości obrazów. W kontekście tekstur słowo **obraz** (*image*) oznacza odpowiedni zestaw tablic z tekselami.

Wymienione niżej funkcje i procedury mają bardzo wiele wariantów, dla tekstur o różnej budowie i zastosowaniach, dlatego ich opisy są tu bardzo skrótowe. Pierwszy parametr wszystkich tych procedur jest typu zamkniętego używanego do identyfikowania tekstur, na przykład `sampler2D` lub `imageBuffer`. Identyfikatory tych typów (zastąpione dalej symbolem *samplerType* albo *imageType*) są słowami kluczowymi języka GLSL zawierającymi rdzeń `sampler` albo `image`, przed którym *może być* przedrostek, a po nim *jest* przyrostek. Brak przedrostka oznacza, że funkcje dające dostęp do tekseli lub obliczające wartości tekstur,

Tabela 9.2. Przyrostki nazw typów dla tekstur i obrazów

przyrostek	opis
`1D`	tekstura jednowymiarowa
`2D`, `2DRect`	tekstura dwuwymiarowa
`2DMS`	tekstura dwuwymiarowa wielopróbkowa
`3D`	tekstura trójwymiarowa
`Cube`	dwuwymiarowe ściany kostki sześciennej
`1DArray`	tablica tekstur jednowymiarowych
`2DArray`	tablica tekstur dwuwymiarowych
`2DMSArray`	tablica dwuwymiarowych tekstur wielopróbkowych
`CubeArray`	zestaw tekstur ścian kostki
`1DShadow`	jednowymiarowa tekstura cienia
`2DShadow`, `2DRectShadow`	dwuwymiarowa tekstura cienia
`CubeShadow`	ściany kostki dla tekstury cienia
`1DArrayShadow`	tablica jednowymiarowych tekstur cienia
`2DArrayShadow`	tablica dwuwymiarowych tekstur cienia
`CubeArrayShadow`	tablica ścian kostki dla tekstur cienia
`Buffer`	tekstura jednowymiarowa w „zwykłej” tablicy

takie jak `texture`, `textureProj`, `textureLod`, i `textureOffset`, podają wartości tekstury reprezentowane za pomocą liczb zmiennopozycyjnych (typu `float`). Przedrostki `i` oraz `u` oznaczają używanie w tym celu liczb całkowitych bez i ze znakiem. Przyrostki są zebrane w tabeli 9.2. Trochę więcej na ten temat napisałem w p. 26.4.1.

```
type textureSize ( samplerType sampler );
type textureSize ( samplerType sampler, int lod );
```

Funkcje podają wymiary tekstur w tekselach. Parametr `sampler` jest identyfikatorem ewaluatora tekstury, do którego jest przywiązana badana tekstura. Parametr `lod`, jeśli jest obecny, wybiera poziom tekstury utworzonej dla mipmapingu (zobacz rozdz. 19). Symbol *type* oznacza typ `int`, `ivec2` lub `ivec3`, przy czym typy te są odpowiednie dla tekstur jedno-, dwu- i trójwymiarowych.

```
vec2 textureQueryLod ( samplerType sampler, type P );
```

Funkcje dostępne tylko dla szaderów fragmentów, których zadaniem jest obliczenie poziomu tekstury dla mipmapingu, na podstawie którego zostałaby obliczona wartość tekstury podczas przetwarzania fragmentu. Współrzędne x i y wektora podanego jako wartość funkcji zawierają numer poziomu (lub dwóch poziomów, jeśli x nie jest liczbą całkowitą) oraz obliczony poziom względem poziomu podstawowego (szczegóły w specyfikacji OpenGL-a [1]).

```
int textureQueryLevels ( samplerType sampler );
```

Funkcje podające liczbę istniejących (tj. utworzonych podczas przygotowywania tekstury) poziomów tekstury dla mipmapingu.

```
int textureSamples ( samplerType sampler );
```

Funkcje podające liczbę próbek dla tekstur wielopróbkowych.

```
type texture ( samplerType sampler, genType P );
type texture ( samplerType sampler, genType P, float bias );
```

Funkcje obliczające wartość tekstury (otrzymaną w wyniku interpolacji i filtrowania tekseli zgodnie z parametrami ewaluatora identyfikowanego przez parametr `sampler`). Liczba składowych parametru `P` musi odpowiadać typowi ewaluatora i wymiarowi tekstury, do której ten ewaluator daje dostęp. Symbol *type* oznacza `vec4`, `uvec4`, `ivec4` i jest zgodny z typem reprezentacji wartości tekstury (jeśli np. parametr `sampler` jest typu `usampler2D`, to wartość funkcji jest typu `uvec4`). Jeśli nazwa typu parametru `sampler` ma w przyrostku słowo `Shadow`, to wynik funkcji jest typu `float`. Parametr `bias`, jeśli jest obecny, jest dodawany do obliczonego poziomu tekstury, co umożliwia zmodyfikowanie dokładności wyniku filtrowania tekstury.

Współrzędne parametru `P` powinny należeć do przedziału $[0,1]$; jeśli któraś z nich „wystaje" poza ten przedział, to zależnie od parametrów ewaluatora zostaje zastąpiona przez liczbę 0 lub 1 albo zastąpiona przez swoją część ułamkową.

```
type textureProj ( samplerType sampler, genType P );
type textureProj ( samplerType sampler, genType P, float bias );
```

Funkcje działające jak funkcje `texture`, ale ich parametr `P` jest wektorem współrzędnych jednorodnych punktu w dziedzinie tekstury. Liczba współrzędnych parametru `P` jest o 1 większa od wymiaru tekstury, współrzędne kartezjańskie w dziedzinie tekstury są otrzymywane w wyniku dzielenia początkowych współrzędnych przez ostatnią. Typ parametru `sampler` nie może mieć nazwy, której przyrostek zawiera słowo `Array` lub `Cube`.

```
type textureLod ( samplerType sampler, genType P, float lod );
```

Funkcje działające jak `texture`, ale dodatkowy parametr `lod` jawnie wskazuje poziom tekstury, którego należy użyć do obliczania jej wartości.

```
type textureOffset ( samplerType sampler, genType P, genIType offset );
type textureOffset ( samplerType sampler, genType P, genIType offset,
                     float bias );
```

Funkcje działające jak `texture`, ale argument (punkt w dziedzinie tekstury) jest poddawany dodatkowemu przesunięciu, którego współrzędne są podane w parametrze `offset`, przy czym ich wartości muszą być zawarte między wartościami zmiennych wbudowanych `gl_MinProgramTexelOffset` i `gl_MaxProgramTexelOffset`.

```
type textureGrad ( samplerType sampler, genType P,
                   genType dPdx, genType dPdy );
```

Funkcje obliczające przefiltrowaną wartość tekstury przy użyciu dodatkowych parametrów wpływających na sposób interpolacji tekseli. Po ich dokładny opis odsyłam do specyfikacji języka GLSL.

genType `texelFetch ( samplerType sampler, genIType P, int lod );`

Funkcje podające wartość teksela bez dokonywania interpolacji ani filtrowania. Współrzędne parametru `P` są indeksami do tablicy tekseli.

Są jeszcze procedury o nazwach `textureProjOffset`, `textureProjLod`, `textureProjLodOffset`, `textureGradOffset`, `textureProjGrad`, `textureProjGradOffset` i `texelFetchOffset`, których działania można się domyślić na podstawie opisu procedur przedstawionych wyżej, a szczegóły można znaleźć w specyfikacji języka GLSL.

type `imageSize ( imageType image );`

Funkcja podaje wymiary obrazu w tekselach; symbol *type* oznacza `int`, `ivec2` lub `ivec3`, przy czym liczba składowych ma być zgodna z wymiarem obrazu (tj. liczbą indeksów tablicy tekseli).

`int imageSamples ( imageType image );`

Funkcje podają liczbę próbek na teksel dla obrazów wielopróbkowych. Symbol *imageType* oznacza typ dla obrazów wielopróbkowych (z przyrostkiem `2DMS` albo `2DMSArray`).

type `imageLoad ( .... );`
`void imageStore ( .... );`

Funkcje odczytujące teksel z obrazu i przekazujące go w wektorze o 4 współrzędnych (*type* oznacza `vec4`, `ivec4` lub `uvec4`) oraz procedury zapisujące teksel w obrazie. Po dokładny opis parametrów odsyłam do specyfikacji OpenGL-a i GLSL-a.

9.14. Liczniki niepodzielne i niepodzielne operacje na pamięci

Liczniki niepodzielne (*atomic counters*) bywają potrzebne w obliczeniach równoległych. Kiedy działające równolegle wątki szadera mają coś policzyć (np. piksele o jakimś kolorze na obrazie), operacja liczenia musi być niepodzielna. Jeśli dwa lub więcej wątków odczyta licznik, zwiększy odczytaną wartość, a potem ją zapisze, to końcowy stan licznika będzie niepoprawny. Wykluczeniu takiego scenariusza służy niepodzielność operacji modyfikowania liczników; dzięki niej żaden wątek nie odczyta licznika przed zakończeniem zapisu do niego wartości zmodyfikowanej przez inny wątek.

Liczniki niepodzielne są zmiennymi jednolitymi przechowywanymi w odpowiednich buforach. Deklaracja licznika może wyglądać tak:

```
layout(binding=1,offset=2) uniform atomic_uint c;
```

W powyższym przykładzie zmienna `c` typu zamkniętego `atomic_uint` jest identyfikatorem licznika umieszczonego w buforze przywiązanym do punktu dowiązania `1` w celu `GL_ATOMIC_COUNTER_BUFFER`, przy czym bufor ten zawiera tablicę liczników. Parametr `offset` kwalifikatora jest indeksem do tej tablicy.

Operacje niepodzielne na licznikach są realizowane przez następujące procedury:

```
uint atomicCounterIncrement ( atomic_uint c );
uint atomicCounterDecrement ( atomic_uint c );
uint atomicCounter ( atomic_uint c );
```

Pierwsze dwie procedury zwiększają i zmniejszają licznik o 1 i przekazują stan licznika *przed* wykonaniem każdej z tych operacji. Trzecia procedura podaje stan licznika bez zmieniania jego wartości.

Istnieje możliwość wykonywania operacji niepodzielnych także na innych zmiennych umieszczonych w buforach, zmiennych współdzielonych (*shared*) i obrazach. Po szczegóły odsyłam do specyfikacji języka GLSL [3].

9.15. Szadery obliczeniowe

GPU składa się z dużej liczby (od kilkudziesięciu do kilkunastu tysięcy) procesorów pracujących równolegle i mających dostęp do dosyć dużej (obecnie zazwyczaj od 300 MB do kilkudziesięciu GB) pamięci. Choć każdy z tych procesorów jest wolniejszy niż CPU i nie mogą one działać całkowicie niezależnie, masywna równoległość obliczeń prowadzonych przy ich użyciu przekłada się na bardzo dużą moc obliczeniową GPU. Szadery obliczeniowe służą do równoległego przetwarzania danych, przy czym mogą to być obliczenia wstępne (preprocesing) na potrzeby syntezy obrazu (jeśli wyniki obliczeń są zapamiętywane w buforach, do których później będą miały dostęp szadery pracujące w potoku przetwarzania grafiki) albo mogą to być zupełnie dowolne obliczenia (niekoniecznie związane z grafiką), których wyniki z pamięci GPU odczyta aplikacja.

Szader obliczeniowy jest wykonywany równolegle jako wiele wątków; wątki te działają w **lokalnych grupach roboczych** (*local workgroups*), zorganizowanych w jedno-, dwu- lub trójwymiarową tablicę, zwaną **globalną grupą roboczą** (*global workgroup*). Lokalna grupa robocza jest jedno-, dwu- lub trójwymiarową tablicą wątków[21]. Każdy wątek otrzymuje swoje wektory indeksów do tych tablic, przypisane przedstawionym dalej zmiennym wbudowanym. Na ich podstawie wątek może pobrać odpowiednie dane, na przykład pewien element tablicy, której wszystkie elementy mają być poddane obróbce, i wykonać obliczenie na tym elemencie, a wynik wpisać do odpowiedniego miejsca tablicy na wyniki.

Wymiary lokalnej grupy roboczej są określone w treści szadera i w skompilowanym i złączonym programie z takim szaderem nie mogą być zmieniane. Wymiary globalnej grupy roboczej są określone przez parametry procedury `glDispatchCompute`, która uruchamia program, i w poszczególnych wywołaniach mogą być dostosowane do rozmiarów bieżących zadań. Przyjęcie grupy lokalnej o wymiarach $1 \times 1 \times 1$ umożliwia zatrudnienie każdego wątku szadera do użytecznej pracy[22]. Grupy lokalne o większych wymiarach mogą się przydać

[21] Tablice jedno- i dwuwymiarowe mają jednoelementowe zakresy niepotrzebnych indeksów.

[22] Jeśli każdy wątek przetwarza jeden obiekt, trzeba przetworzyć n obiektów i liczba n jest niepodzielna przez liczbę wątków w grupie lokalnej, to pewne wątki będą bezproduktywne, ale muszą zostać uruchomione.

z dwóch powodów: wątki w tej samej grupie lokalnej mogą się komunikować za pośrednictwem opisanych niżej zmiennych współdzielonych i możliwa jest synchronizacja wątków w obrębie grupy lokalnej za pomocą procedur wywoływanych przez szader. Do synchronizacji wątków całej globalnej grupy roboczej konieczne jest zaangażowanie CPU, co zabiera więcej czasu.

Zmienne współdzielone to są globalne zmienne zadeklarowane z kwalifikatorem `shared`. Rozmaite algorytmy równoległe mogą ich używać w ten sposób, że na przykład każdy wątek nadaje wartość pewnemu elementowi tablicy będącej zmienną współdzieloną, a potem wykonuje obliczenie, odwołując się do elementów tablicy, którym wartości nadały inne wątki. Zmienne współdzielone mogą być przechowywane w specjalnych rejestrach GPU zapewniających bardzo szybki dostęp, co umożliwia daleko idącą optymalizację algorytmów. Jednakże, dostępna pamięć na zmienne współdzielone dla lokalnej grupy roboczej jest ograniczona. Ilość tej pamięci aplikacja może odczytać, wywołując procedurę `glGetIntegerv` z parametrem `GL_MAX_COMPUTE_SHARED_MEMORY_SIZE`. Specyfikacja [1] gwarantuje możliwość użycia 32 kB tej pamięci.

9.15.1. Zmienne wbudowane szadera obliczeniowego

`in uvec3 gl_NumWorkGroups;` — wektor $(x_{\mathrm{GWS}}, y_{\mathrm{GWS}}, z_{\mathrm{GWS}})$ z wymiarami globalnej grupy roboczej, określonymi przez parametry wywołania procedury

```
glDispatchCompute ( xgws, ygws, zgws );
```

która uruchamia wątki. Elementami tablicy o takich wymiarach są lokalne grupy robocze.

`const uvec3 gl_WorkGroupSize;` — wektor $(x_{\mathrm{LWS}}, y_{\mathrm{LWS}}, z_{\mathrm{LWS}})$ z wymiarami lokalnej grupy roboczej, zadeklarowanymi w kodzie szadera, w kwalifikatorze wejścia wyglądającym tak:

`layout(local_size_x=`x_{LWS}`,local_size_y=`y_{LWS}`,local_size_z=`z_{LWS}`) in;`

przy czym muszą być podane konkretne liczby (wielkość lokalnej grupy roboczej jest ustalana podczas kompilacji szadera). Jeśli program powstał z połączenia kilku szaderów obliczeniowych, to trzeba podać ten kwalifikator w przynajmniej jednym z nich (najlepiej tym z procedurą `main`), a każdy inny szader w programie musi mieć takie same wymiary, jeśli są w nim podane. Jeśli wymiary y_{LWS} lub z_{LWS} są równe 1, to można napisać krócej

`layout(local_size_x=`x_{LWS}`,local_size_y=`y_{LWS}`) in;`

albo

`layout(local_size_x=`x_{LWS}`) in;`

Jeśli lokalna grupa robocza ma mieć wymiary $1 \times 1 \times 1$, to wystarczy napisać

```
layout(local_size_x=1) in;
```

`in uvec3 gl_WorkGroupID;` — wektor $(x_{\mathrm{WID}}, y_{\mathrm{WID}}, z_{\mathrm{WID}})$ zawierający trójkę indeksów lokalnej grupy roboczej w globalnej grupie roboczej.

`in uvec3 gl_LocalInvocationID;` — wektor $(x_{\mathrm{LID}}, y_{\mathrm{LID}}, z_{\mathrm{LID}})$ zawierający trójkę indeksów wątku w lokalnej grupie roboczej.

`in uvec3 gl_GlobalInvocationID;` — wektor $(x_{\mathrm{GID}}, y_{\mathrm{GID}}, z_{\mathrm{GID}})$ zawierający indeksy wątku w globalnej grupie roboczej, określone wzorem

$$\begin{bmatrix} x_{\mathrm{GID}} \\ y_{\mathrm{GID}} \\ z_{\mathrm{GID}} \end{bmatrix} = \begin{bmatrix} x_{\mathrm{LWS}} x_{\mathrm{WID}} + x_{\mathrm{LID}} \\ y_{\mathrm{LWS}} y_{\mathrm{WID}} + y_{\mathrm{LID}} \\ z_{\mathrm{LWS}} z_{\mathrm{WID}} + z_{\mathrm{LID}} \end{bmatrix}$$

`in uint gl_LocalInvocationIndex;` — jedna liczba, numer wątku w lokalnej grupie roboczej uszeregowanej w jednowymiarowy ciąg, określona wzorem

$$i = (z_{\mathrm{LID}} y_{\mathrm{LWS}} + y_{\mathrm{LID}}) x_{\mathrm{LWS}} + x_{\mathrm{LID}}.$$

Aplikacja musi znać zadeklarowane w treści szadera wymiary lokalnej grupy roboczej, aby mogła dobierać wymiary grupy globalnej. Można w tym celu powtórzyć te wymiary (np. umieścić w kodach źródłowych szadera i aplikacji takie same makrodefinicje), ale wtedy zmieniając je w jednym miejscu, trzeba pamiętać o zmienieniu ich w drugim. Można też podać wymiary tylko w treści szadera, a w aplikacji napisać instrukcję odczytującą te wymiary z programu szaderów. Instrukcja ta ma postać

```
glGetProgramiv ( program_id, GL_COMPUTE_WORK_GROUP_SIZE, lgsize );
```

Pierwszy parametr jest identyfikatorem programu, a trzeci jest tablicą trzech zmiennych typu `GLint`, do której wymiary grupy roboczej zostaną wpisane.

Istnieje limit wielkości lokalnych grup roboczych; specyfikacja OpenGL 4.5 [1] gwarantuje, że współrzędne wektora $(x_{\mathrm{LWS}}, y_{\mathrm{LWS}}, z_{\mathrm{LWS}})$ mogą przyjmować wartości odpowiednio do 1024, 1024, 64. Istnieje też limit łącznej liczby wątków lokalnej grupy roboczej, tj. iloczynu tych trzech liczb, według specyfikacji nie mniejszy niż 1024. Implementacje mogą dopuszczać większe limity. Możemy je poznać, wykonując instrukcje

```
for ( i = 0; i < 3; i++ )
  glGetIntegeri_v ( GL_MAX_COMPUTE_WORK_GROUP_SIZE, i, &d[i] );
glGetIntegerv ( GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS, &n );
```

które wpiszą je do tablicy `d` i do zmiennej `n`.

Istnieje też limit wielkości globalnej grupy roboczej, której dopuszczalne wymiary według specyfikacji to $65535 \times 65535 \times 65535$. Wymiary dopuszczalne w implementacji odczytuje instrukcja

```
for ( i = 0; i < 3; i++ )
  glGetIntegeri_v ( GL_MAX_COMPUTE_WORK_GROUP_COUNT, i, &d[i] );
```

Na moim komputerze pierwsza współrzędna wektora $(x_{\mathrm{GWS}}, y_{\mathrm{GWS}}, z_{\mathrm{GWS}})$ może mieć wartość $2^{31} - 1$, czyli znacznie więcej, a pozostałe dwie tak jak w specyfikacji, $2^{16} - 1$.

9.15.2. Procedury synchronizacji obliczeń

Jeśli liczba wątków w grupie roboczej przekracza liczbę procesorów GPU, to grupa jest dzielona na części wykonywane kolejno, przy czym kolejność, w jakiej poszczególne wątki będą uruchamiane, jest nieokreślona. Co za tym idzie, kolejność, w jakiej poszczególne wątki zapiszą swoje wyniki w pamięci GPU jest nieokreślona, wskutek czego może się zdarzyć, że pewien wątek odczyta wartość zmiennej przedwcześnie, tj. przed nadaniem jej właściwej wartości przez inny wątek. Do zapewnienia, że wyniki obliczeń będące danymi dla kolejnego etapu zostały zapisane, służą wymienione niżej procedury synchronizacji.

```
void barrier ( void );
void memoryBarrierShared ( void );
void groupMemoryBarrier ( void );
```

Pierwsza z tych procedur czeka, aż wszystkie wątki *lokalnej* grupy roboczej dokończą wykonywanie instrukcji poprzedzających jej wywołanie, a pozostałe dwie czekają na dokończenie czytania lub pisania zmiennych współdzielonych albo pamięci RAM GPU.

Do synchronizacji zapisu i odczytu pamięci przez *globalną* grupę roboczą i do synchronizacji przed odczytem pamięci GPU przez CPU służy procedura `glMemoryBarrier`, którą musi w odpowiednich chwilach wywołać CPU. W większości opisanych dalej aplikacji ta procedura jest wywoływana po to, aby wszystkie wątki dokończyły obliczenia i zapisały ich wyniki w buforach magazynowych. Kody źródłowe tych aplikacji skróciłem przy użyciu makrodefinicji pokazanej na listingu 9.1.

Listing 9.1. Makrodefinicja `COMPUTE`

C

```
1: #define COMPUTE(SIZEX,SIZEY,SIZEZ) \
2:   { glDispatchCompute ( SIZEX, SIZEY, SIZEZ ); \
3:     glMemoryBarrier ( GL_SHADER_STORAGE_BARRIER_BIT ); }
```

Może być też potrzebna synchronizacja między wątkami szaderów działającymi w potoku przetwarzania grafiki, wykonującymi równolegle obliczenia dla wierzchołków lub fragmentów trójkąta i komunikującymi się za pośrednictwem obrazu, bloku magazynowego lub liczników niepodzielnych. Do tego służą następujące procedury:

```
void memoryBarrierImage ( void );
void memoryBarrierBuffer ( void );
void memoryBarrierAtomicCounter ( void );
void memoryBarrier ( void );
```

Również szadery obliczeniowe mogą wywoływać te procedury; ich zadaniem jest wstrzymanie wątku do chwili zakończenia przez pozostałe wątki operacji odpowiednio na obrazach, blokach magazynowych i licznikach niepodzielnych lub wszelkich operacji czytania i pisania pamięci RAM GPU.

Więcej informacji na temat szaderów obliczeniowych podam w dalszych rozdziałach (23 i kolejnych) oraz w dodatku G, przedstawiając je na przykładach.

10

Aplikacja pierwsza B

Każdy model matematyczny oświetlenia musi uwzględniać trzy elementy: źródła światła, oświetlaną powierzchnię i obserwatora. Model jest wzorem, do którego trzeba podstawić odpowiednie parametry tych trzech elementów. W tym rozdziale zrealizujemy model najprostszy — tzw. **oświetlenia lambertowskiego**, w którym przyjmuje się, że oświetlone powierzchnie są matowe[1].

Podstawową rolę we wszystkich modelach oświetlenia powierzchni odgrywa jej **wektor normalny**, który jest częścią opisu obiektu. Problemem do rozwiązania jest uzyskanie wektora normalnego dla każdego wierzchołka. Można go oczywiście podać jako atrybut wierzchołka. Ale dla bryły wielościennej, jaką jest dwudziestościan, każdy wierzchołek należy do kilku (pięciu) ścian i dlatego musiałby wystąpić wielokrotnie w tablicy wierzchołków — każdy egzemplarz z innym wektorem normalnym, właściwym dla odpowiedniej ściany. Niestety, etap pobierania wierzchołków nie ma możliwości podawania atrybutów *ścian*. Podanie wektora normalnego dla wierzchołka należącego do wielu ścian leżących w różnych płaszczyznach ma sens, jeśli ściany te stanowią przybliżenie fragmentu gładkiej powierzchni zakrzywionej (i w kolejnych aplikacjach będziemy z tego korzystać). Jeśli wektor normalny nie jest atrybutem wierzchołka, to szader wierzchołków nie dysponuje informacją umożliwiającą jego obliczenie dla ściany. Ale taką informacją może dysponować szader geometrii, którego właśnie użyjemy.

10.1. Szadery — pierwszy program

Aplikacja będzie rysować wierzchołki i krawędzie figury tak jak poprzednio, natomiast do wyświetlania ścian użyje innego programu szaderów, tak więc będą w użyciu dwa programy; w pierwszym zastosujemy szadery wypróbowane wcześniej (z pewną drobną modyfikacją), a w drugim programie zrealizujemy model oświetlenia. **Szader wierzchołków** pierwszego programu, pokazany na listingu 10.1, został przystosowany do nowo zdefiniowanego bloku

[1]Johann Heinrich Lambert, wynajdując w 1760 roku ten model, nie przewidział, jak ważny okaże się on w grafice komputerowej — zarówno w teorii (zobacz podrozdz. 28.3 i 28.5), jak i w praktyce (tu i w rozdz. 29).

zmiennych jednolitych `TransBlock` opisujących przekształcenia[2]. Blok ten jest wspólny dla obu programów, zatem musi (powinien) być wszędzie taki sam[3]. Do trzech macierzy przekształceń obecnych wcześniej doszła macierz będąca iloczynem ostatnich dwóch; dzięki temu przekształcenie wierzchołka wymaga tylko dwóch mnożeń wektora przez macierz (linia 15) zamiast trzech. Za obliczenie iloczynu tych macierzy i umieszczenie go w zmiennej jednolitej `TransBlock.vpm` odpowiada oczywiście aplikacja.

Listing 10.1. Szader wierzchołków pierwszego programu

GLSL

```
1: #version 420
2:
3: layout(location=0) in vec4 in_Position;
4: layout(location=1) in vec3 in_Colour;
5:
6: out vec3 Colour;
7:
8: uniform TransBlock {
9:     mat4 mm, vm, pm, vpm;
10:    vec4 eyepos;
11:  } trb;
12:
13: void main ( void )
14: {
15:   gl_Position = trb.vpm * (trb.mm * in_Position);
16:   Colour = in_Colour;
17: } /*main*/
```

Szader fragmentów pierwszego programu jest podany na listingu 7.2; nie ma potrzeby wprowadzać w nim żadnych zmian.

10.2. Model oświetlenia i drugi program szaderów

Lambertowski model oświetlenia jest opisany wzorem

$$L = \boldsymbol{a} \sum_{i=0}^{n-1} \left(I_i^{\text{amb}} + I_i^{\text{dir}} v_i |\langle \boldsymbol{l}_i, \boldsymbol{n} \rangle| \right), \tag{10.1}$$

[2]Gdyby ktoś spytał, czy nie można było tak od razu, to bym odpowiedział, że w żadnej sprawie nie istnieje „ostateczne rozwiązanie" i nigdy nie należy uważać, że się je znalazło; zresztą w podróży więcej można zobaczyć wzdłuż drogi niż na jej końcu. Wraz z pojawianiem się kolejnych potrzeb jeszcze nie raz będziemy dokonywać przeróbek działającego kodu. Gdybym zaś w początkowych rozdziałach pokazał na listingach szadery z wszystkimi elementami, które kiedyś mogą się przydać, to byłoby to wrzucanie Czytelników na zbyt głęboką wodę, a poza tym to by nie były bardzo dobre szadery.

[3]Pole `eyepos` bloku `TransBlock` nie jest używane przez pierwszy program szaderów i, jako że jest na końcu bloku, można by je pominąć. Ale lepiej, żeby opisy bloku używanego przez więcej niż jeden program były identyczne we wszystkich tych programach.

w którym poszczególne symbole mają następujące znaczenie: L oznacza intensywność[4] światła odbitego od powierzchni w stronę obserwatora. W świecie fizycznym jest to funkcja długości fali świetlnej $\lambda \in [380\,\text{nm}, 680\,\text{nm}]$. W komputerze jest to wektor w $\mathbb{R}^3$, którego współrzędne r, g, b określają wartości składowych czerwonej, zielonej i niebieskiej przypisywanych pikselowi (do tego doczepiona jest współrzędna a, czyli składowa alfa, której nie widać na ekranie); wszystkie współrzędne koloru przypisywanego pikselowi powinny mieć wartości z przedziału $[0,1]$.

Czynnik $\boldsymbol{a}$ (który też jest funkcją zmiennej λ reprezentowaną przez wektor w $\mathbb{R}^3$) opisuje zdolność powierzchni do odbijania światła; jest on kolorem podanym jako atrybut wierzchołka. Indeks i służy do ponumerowania źródeł światła, o których zakładamy, że są punktowe. Światło do powierzchni może dochodzić z określonego punktu (np. z lampy umieszczonej w rysowanej scenie lub w jej pobliżu) lub z określonego kierunku (czyli z lampy znajdującej się „nieskończenie daleko", takiej jak Słońce). Wektor jednostkowy $\boldsymbol{l}_i$ wyznacza kierunek od punktu na powierzchni do źródła światła, natomiast $\boldsymbol{n}$ oznacza jednostkowy wektor normalny powierzchni. Iloczyn skalarny tych dwóch wektorów, $\langle \boldsymbol{l}_i, \boldsymbol{n} \rangle$, jest kosinusem kąta między nimi. Czynnik v_i odpowiada za widoczność: ma on wartość 1, jeśli obserwator znajduje się po tej samej stronie powierzchni co źródło światła[5], i 0, jeśli po przeciwnej — tu zakładamy, że powierzchnia jest całkowicie nieprzezroczysta. W bardziej zaawansowanej grafice czynnik v_i ma wartość 0 także wtedy, gdy między źródłem światła a danym punktem na powierzchni znajduje się jakiś obiekt rzucający cień. Ale na razie nie bierzemy się jeszcze za wyznaczanie cieni.

Intensywność światła dochodzącego do danego punktu na powierzchni bezpośrednio od źródła światła jest oznaczona symbolem I_i^{dir}. Przyjęte jest założenie, że część światła wysyłanego przez to źródło rozprasza się (w atmosferze i w wielokrotnych odbiciach od różnych obiektów), w związku z czym we wzorze mamy składnik I_i^{amb}, opisujący światło pochodzące z tego samego źródła, ale dochodzące do danego punktu ze *wszystkich* kierunków[6].

Jeśli źródło światła znajduje się bardzo daleko od oświetlanych obiektów, to kierunek $\boldsymbol{l}_i$ dla wszystkich punktów powierzchni tych obiektów jest taki sam. Ponadto intensywność I_i^{dir} oświetlenia kierunkowego jest stała. Jeśli natomiast odległość między źródłem światła a obiektami jest skończona, to po pierwsze, kierunki $\boldsymbol{l}_i$ od poszczególnych punktów powierzchni do źródła światła są różne, a po drugie intensywność światła maleje ze wzrostem odległości od źródła. Z zasady zachowania energii wynika, że dla idealnego źródła punktowego intensywność powinna być odwrotnie proporcjonalna do kwadratu odległości, ale w praktyce lepiej sprawdza się wzór

$$I_i^{\text{dir}} = \frac{I_i^{\text{em}}}{at^2 + bt + c}, \tag{10.2}$$

[4] Tu i w dalszych rozdziałach wszelkie pojęcia radiometrii wrzucamy do jednego worka z napisem „intensywność", korzystając z tego, że można napisać szadery i aplikację bez wnikania w naturę tych pojęć. Ale zachęcam do zajrzenia do rozdziału 28.

[5] a właściwie płaszczyzny stycznej do powierzchni

[6] W tym najprostszym modelu nie badamy rozkładu kierunkowego oświetlenia, co umożliwiłoby otrzymanie półcieni na obrazie.

w którym t oznacza odległość, I_i^{em} to moc światła emitowanego przez źródło, z kolei a, b, c to współczynniki, które trzeba dobrać doświadczalnie (musi być $a > 0$, $b, c \geqslant 0$).

Wiemy już, co trzeba dostarczyć szaderowi fragmentów, aby można było obliczyć kolor piksela, możemy zatem zaprojektować szadery. **Szader wierzchołków** na listingu 10.2 oblicza w liniach 20 i 23 współrzędne wierzchołka w układzie kostki standardowej, a w linii 22 przepisuje do pola `Colour` struktury wyjściowej atrybut koloru wierzchołka, który będzie dalej wykorzystany jako współczynnik $\boldsymbol{a}$ w modelu oświetlenia. Do dalszych obliczeń jest też potrzebny wektor współrzędnych kartezjańskich wierzchołka w układzie świata. Wynikiem mnożenia przez macierz przekształcenia modelu w linii 20 jest wektor współrzędnych jednorodnych, a współrzędne kartezjańskie obliczamy w linii 21, dzieląc pierwsze trzy współrzędne jednorodne przez współrzędną wagową.

Dwa atrybuty wierzchołka — kolor i wektor współrzędnych w układzie świata — są przekazywane w polach struktury, kóra ma dwie nazwy: **nazwę zewnętrzną** `Vertex`, która jest taka sama w szaderze geometrii, i **nazwę lokalną** (**nazwę instancji**) `Out`, używaną w instrukcjach szadera i podkreślającą dodatkowo, że ta struktura zawiera wyniki obliczeń szadera.

Listing 10.2. Szader wierzchołków drugiego programu

GLSL

```
1: #version 420
2:
3: layout(location=0) in vec4 in_Position;
4: layout(location=1) in vec3 in_Colour;
5:
6: out Vertex {
7:     vec3 Colour;
8:     vec3 Position;
9:   } Out;
10:
11: uniform TransBlock {
12:     mat4 mm, vm, pm, vpm;
13:     vec4 eyepos;
14:   } trb;
15:
16: void main ( void )
17: {
18:   vec4 Pos;
19:
20:   Pos = trb.mm * in_Position;
21:   Out.Position = Pos.xyz / Pos.w;
22:   Out.Colour = in_Colour;
23:   gl_Position = trb.vpm * Pos;
24: } /*main*/
```

Zadaniem **szadera geometrii**, pokazanego na listingu 10.3, jest obliczenie wektora normalnego trójkątnej ściany. Kwalifikator `layout` w linii 3 opisuje dane wejściowe dla szadera;

podczas rysowania trójkątów (osobnych lub wchodzących w skład taśmy lub wachlarza — po etapie pobierania wierzchołków to już nie ma znaczenia) szader geometrii zostanie wywołany dla każdego trójkąta i otrzyma dane opisujące wszystkie trzy wierzchołki.

Listing 10.3. Szader geometrii drugiego programu

GLSL

```
1: #version 420
2:
3: layout(triangles) in;
4: layout(triangle_strip,max_vertices=3) out;
5:
6: in Vertex {
7:     vec3 Colour;
8:     vec3 Position;
9:   } In[];
10:
11: out FVertex {
12:     vec3 Colour;
13:     vec3 Position;
14:     vec3 Normal;
15:   } Out;
16:
17: void main ( void )
18: {
19:   int i;
20:   vec3 v1, v2, nv;
21:
22:   v1 = In[1].Position - In[0].Position;
23:   v2 = In[2].Position - In[0].Position;
24:   nv = normalize ( cross ( v1, v2 ) );
25:   for ( i = 0; i < 3; i++ ) {
26:     gl_Position = gl_in[i].gl_Position;
27:     Out.Position = In[i].Position;
28:     Out.Normal = nv;
29:     Out.Colour = In[i].Colour;
30:     EmitVertex ();
31:   }
32:   EndPrimitive ();
33: } /*main*/
```

Zgodnie z opisem w kwalifikatorze dane wejściowe są wierzchołkami trójkąta podanymi w tablicach o długości 3. Zmienna wbudowana o nazwie lokalnej `gl_in` składa się ze zmiennych strukturalnych `gl_PerVertex`, opisanych w poprzednim rozdziale. Każda z tych struktur zawiera współrzędne jednorodne wierzchołka w polu `gl_Position` typu `vec4`, przypisane przez szader wierzchołków (w linii 23 na listingu 10.2). Pozostałe pola tych struktur nie są używane.

Druga tablica danych wejściowych składa się ze struktur o nazwie zewnętrznej `Vertex` i o lokalnej nazwie `In`. Długość tej tablicy nie jest podana jawnie, bo jest ona określona przez wspomniany wyżej kwalifikator. Struktura `Vertex` musi mieć identyczną budowę jak na listingu 10.2, bo w przeciwnym razie szaderów nie dałoby się połączyć w program.

Kwalifikator `layout` w linii 4 opisuje postać wyjścia szadera. Szader geometrii może *zmienić rodzaj* obiektu geometrycznego; rodzaj ten jest określony właśnie przez ten kwalifikator. W naszym przypadku trójkąt ma pozostać trójkątem. Zapis w kwalifikatorze formalnie wygląda tak, że szader geometrii produkuje taśmę trójkątową, w której są najwyżej 3 wierzchołki — czyli jest w niej tylko jeden trójkąt.

Struktura wyjściowa `Out` (o nazwie globalnej `FVertex`, linie 11–15) *nie jest* tablicą. Szader ma do tej struktury kolejno dla każdego wierzchołka wpisać dane wyjściowe i wywołać procedurę `EmitVertex`, która wprowadza dane z tej struktury do dalszych etapów w potoku przetwarzania grafiki.

Procedura `main` szadera w liniach 22–23 oblicza dwa wektory o kierunkach i długościach boków trójkąta, a następnie w linii 24 oblicza ich iloczyn wektorowy i poddaje go normalizacji. W ten sposób obliczony jest jednostkowy wektor normalny płaszczyzny trójkąta, który zostanie wyprowadzony jako atrybut `Normal` wszystkich trzech wierzchołków. Współrzędne wierzchołków używane w tym obliczeniu są kartezjańskie, podane w układzie świata; obliczył je szader wierzchołków w liniach 20 i 21 na listingu 10.2.

W pętli w liniach 25–31 szader geometrii wpisuje do struktury wyjściowej kolejno położenie, wektor normalny i kolor; położenie (w układzie świata) i kolor są kopiowane z danych wejściowych. Po wpisaniu danych dla każdego wierzchołka szader wywołuje procedurę `EmitVertex`, a na zakończenie jest wywoływana procedura `EndPrimitive`, która sygnalizuje zakończenie wyprodukowanej przez szader „taśmy" złożonej z jednego trójkąta.

Uwaga: Choć atrybut `Normal` jest taki sam dla wszystkich wierzchołków, przypisanie

```
Out.Normal = nv;
```

trzeba wykonać w każdym przebiegu pętli. Procedura `EmitVertex` ma efekt uboczny, który sprawia, że wartości wszystkich pól struktury `Out` stają się nieokreślone.

Obliczenie koloru oświetlonego punktu powierzchni wykonuje **szader fragmentów** przedstawiony na listingu 10.4. Dane wejściowe dotyczące fragmentu są obecne w zmiennych wbudowanych opisanych w poprzednim rozdziale (ten szader w ogóle z nich nie korzysta) oraz w strukturze `FVertex` o budowie identycznej ze strukturą `FVertex` opisującą dodatkowe wyjście szadera geometrii.

Obliczony kolor piksela ma być przypisany zmiennej `out_Colour`. Pozostałe dane potrzebne do tego obliczenia są obecne w dwóch blokach zmiennych jednolitych.

W bloku `TransBlock`, do którego wcześniej odwoływał się szader wierzchołków, są podane macierze przekształceń (których ten szader nie używa) oraz (w polu `eyepos`) współrzędne położenia obserwatora w układzie świata.

W bloku `LSBlock` znajduje się opis źródeł światła. Pola `nls` i `mask` zawierają informację o liczbie zdefiniowanych źródeł światła i o tym, które z tych źródeł są w danej chwili

Listing 10.4. Szader fragmentów drugiego programu

GLSL

```
#version 420

#define MAX_NLIGHTS 8

in FVertex {
    vec3 Colour;
    vec3 Position;
    vec3 Normal;
  } In;

out vec4 out_Colour;

uniform TransBlock {
    mat4 mm, vm, pm, vpm;
    vec4 eyepos;
  } trb;

struct LSPar {
    vec4 position;
    vec3 ambient, direct;
    vec3 attenuation;
  };

uniform LSBlock {
    uint  nls;              /* liczba źródeł światła */
    uint  mask;             /* maska włączonych źródeł */
    LSPar ls[MAX_NLIGHTS];  /* poszczególne źródła światła */
  } light;

vec3 posDifference ( vec4 p, vec3 pos, out float dist )
{
  vec3 v;

  if ( p.w != 0.0 ) {
    v = p.xyz - pos.xyz;  /* tu zawsze p.w == 1.0 */
    dist = sqrt ( dot ( v, v ) );
  }
  else
    v = p.xyz;
  return normalize ( v );
} /*posDifference*/

float attFactor ( vec3 att, float dist )
{
  return 1.0/(((att.z*dist)+att.y)*dist+att.x);
```

```
46: } /*attFactor*/
47:
48: vec3 LambertLighting ( void )
49: {
50:   vec3  normal, lv, vv, Colour;
51:   float d, e, dist;
52:   uint  i, mask;
53:
54:   normal = normalize ( In.Normal );
55:   vv = posDifference ( trb.eyepos, In.Position, dist );
56:   e = dot ( vv, normal );
57:   Colour = vec3(0.0);
58:   for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
59:     if ( (light.mask & mask) != 0 ) {
60:       Colour += light.ls[i].ambient * In.Colour;
61:       lv = posDifference ( light.ls[i].position, In.Position, dist );
62:       d = dot ( lv, normal );
63:       if ( e > 0.0 ) {
64:         if ( d > 0.0 ) {
65:           if ( light.ls[i].position.w != 0.0 )
66:             d *= attFactor ( light.ls[i].attenuation, dist );
67:           Colour += (d * light.ls[i].direct) * In.Colour;
68:         }
69:       }
70:       else {
71:         if ( d < 0.0 ) {
72:           if ( light.ls[i].position.w != 0.0 )
73:             d *= attFactor ( light.ls[i].attenuation, dist );
74:           Colour -= (d * light.ls[i].direct) * In.Colour;
75:         }
76:       }
77:     }
78:   return clamp ( Colour, 0.0, 1.0 );
79: } /*LambertLighting*/
80:
81: #define AGamma(colour) pow ( colour, vec3(256.0/563.0) )
82:
83: void main ( void )
84: {
85:   out_Colour = vec4 ( AGamma ( LambertLighting () ), 1.0 );
86: } /*main*/
```

włączone. Pole `ls` jest tablicą struktur typu `LSPar`, zdefiniowanego w liniach 18–22. Struktura taka zawiera parametry jednego źródła światła. Długość tablicy jest określona przez makrodefinicję w linii 3. W razie potrzeby długość ta może być zmieniona, ale nie może ona być większa niż 32, ponieważ zmienna `light.mask` ma dokładnie 32 bity i każde źródło światła jest związane z jednym z nich.

Pole `ambient` struktury `LSPar` przechowuje współrzędne wektora I_i^{amb} (wzór (10.1)) opisującego intensywność światła rozproszonego w otoczeniu, pochodzącego od i-tego źródła.

Pole `direct` reprezentuje wektor I_i^{dir}, jeśli źródło światła jest położone bardzo daleko od obiektu, albo wektor I_i^{em} ze wzoru (10.2), jeśli źródło jest w skończonej odległości. Wektory I_i^{amb} i I_i^{em} nie muszą mieć tego samego kierunku, tj. mogą opisywać różne odcienie koloru światła. Rozpraszając się, światło może zmienić odcień, jeśli na przykład obiekt i żarówka oświetlająca go znajdują się w pomieszczeniu, którego ściany są fioletowe.

Pole `position` przechowuje współrzędne jednorodne położenia źródła światła. Jeśli współrzędna wagowa jest równa 0, to źródło to jest położone daleko od obiektu i pierwsze trzy współrzędne określają kierunek, z którego światło dochodzi do wszystkich punktów sceny. Jeśli współrzędna wagowa nie jest zerem, to jest równa 1 i pierwsze trzy współrzędne są kartezjańskie — zapewnia to opisana dalej procedura `SetLightPosition` (listing 10.9), która dla źródeł światła w skończonej odległości przed przesłaniem położenia do pamięci GPU dzieli ich współrzędne jednorodne przez współrzędną wagową.

Pole `attenuation` zawiera współczynniki a, b, c wielomianu w mianowniku ułamka we wzorze (10.2). Są one używane w obliczeniach, jeśli źródło światła jest w skończonej odległości od obiektu.

Procedura `PosDifference` (linie 30–41) oblicza wektor jednostkowy $\boldsymbol{v}$ skierowany od punktu `pos` do punktu `p`. Pierwszy z tych punktów znajduje się na oświetlanej powierzchni; podaje się jego współrzędne kartezjańskie. Dla drugiego z tych punktów podaje się współrzędne jednorodne, ponieważ *może* on być „punktem w nieskończoności" i wtedy pierwsze trzy współrzędne jednorodne wyznaczają kierunek i zwrot wektora $\boldsymbol{v}$. Jeśli współrzędna wagowa nie jest zerem, to jest jedynką; wtedy procedura w linii 35 odejmuje punkty (tj. wektory współrzędnych kartezjańskich), a potem normalizuje wektor różnicy. Dodatkowo w linii 36 jest obliczana odległość punktów. Odległość ta jest przypisywana do parametru wyjściowego `dist`.

Procedura `attFactor` (linie 43–46) oblicza (przy użyciu schematu Hornera) wartość wyrażenia $1/(at^2+bt+c)$ dla liczb c, b, a będących współrzędnymi wektora `att` i liczby t będącej wartością parametru `dist`.

Oświetlenie fragmentu jest obliczane przez procedurę `LambertLighting`, która w linii 54 dokonuje normalizacji wektora `In.Normal`, aby otrzymać jednostkowy wektor $\boldsymbol{n}$ potrzebny we wzorze (10.1). Wprawdzie dla płaskiej ściany, której wszystkie wierzchołki mają podany ten sam wektor jednostkowy, to jest zbędne, ale chcemy, aby szader nadawał się także do tworzenia obrazów powierzchni zakrzywionych, dla których będziemy podawać różne wektory dla różnych wierzchołków trójkąta. W takim przypadku interpolacja wektorów jednostkowych wytwarza wektory o długościach innych niż 1 i trzeba je normalizować.

W linii 55 obliczany jest wektor jednostkowy $\boldsymbol{v}$ opisujący kierunek od punktu na powierzchni do obserwatora. Punkt na powierzchni jest dany w zmiennej `In.Position` i został otrzymany przez interpolację położeń wierzchołków trójkąta (w układzie świata). Położenie obserwatora jest dane w zmiennej jednolitej `trb.eyepos`. Jeśli obraz jest wykonywany przy użyciu rzutu perspektywicznego, to obserwator jest w skończonej odległości od obiektu i współrzędna wagowa jego położenia jest niezerowa. Jeśli stosowany jest rzut równoległy, to

obserwator widzi scenę z ustalonego kierunku. Wtedy współrzędna wagowa jego położenia jest równa 0. Aplikacja musi zadbać o to, aby za każdym razem, gdy zmieniana jest macierz przejścia od układu świata do układu obserwatora (przechowywana w zmiennej `trb.vm`), zmienna `trb.eyepos` była odpowiednio uaktualniana. W linii 56 obliczany jest iloczyn skalarny wektora normalnego i wektora $\boldsymbol{v}$ w celu ustalenia, po której stronie płaszczyzny ściany (lub ogólniej, płaszczyzny stycznej do powierzchni obiektu) znajduje się obserwator. Istotny jest *znak* tego iloczynu.

W linii 57 szader kasuje zmienną `Colour`, a następnie w pętli oblicza i dodaje kolejne składniki sumy we wzorze (10.1). Liczba źródeł światła jest podana w zmiennej `light.nls`. Dodatkowo maska bitowa `light.mask` określa źródła światła „włączone" w chwili rysowania; źródła wyłączone są pomijane w obliczeniach. W linii 60 do światła odbijanego przez punkt powierzchni dodawany jest składnik $\boldsymbol{a}I_i^{\mathrm{amb}}$. W linii 61 jest obliczany wektor $\boldsymbol{l}_i$, a w linii 62 jego iloczyn skalarny z wektorem normalnym $\boldsymbol{n}$.

Instrukcje warunkowe w liniach 63, 64 i 71 mają na celu ustalenie, czy źródło światła i obserwator znajdują się po tej samej stronie płaszczyzny ściany. Polega to na zbadaniu, czy iloczyny skalarne $\langle \boldsymbol{l}_i, \boldsymbol{n} \rangle$ oraz (obliczony wcześniej) $\langle \boldsymbol{v}, \boldsymbol{n} \rangle$, gdzie wektor $\boldsymbol{v}$ wyznacza kierunek do obserwatora, mają ten sam znak. Zauważmy, że zwrot wektora $\boldsymbol{n}$ nie ma znaczenia. Jeśli obserwator widzi oświetloną stronę powierzchni, zależnie od potrzeb (sprawdzane to jest w liniach 65 i 72) jest obliczany czynnik osłabienia światła ze wzrostem odległości od jego źródła. Następnie składnik $\boldsymbol{a}I_i^{\mathrm{dir}}|\langle \boldsymbol{l}_i, \boldsymbol{n} \rangle|$ jest dodawany do światła odbijanego przez punkt powierzchni; zwracam uwagę na użyte w liniach 67 i 74 operatory „+=" i „-=", które efektywnie produkują wartość bezwzględną.

Procedura `main` w linii 85 oblicza wektor współrzędnych koloru, który zostanie przekazany na wyjście szadera. Współrzędne *r*, *g*, *b* obliczone przez procedurę realizującą model oświetlenia opisują intensywność światła odbitego od powierzchni, a zatem powinny też opisywać moc światła emitowanego przez piksel monitora. Ale zależność między tą mocą a liczbami wpisanymi do bufora obrazu jest nieliniowa — więcej wiadomości na ten temat zawiera podrozdział C.3. Przekształcenie realizowane w tym miejscu jest nazywane **korekcją gamma** i ma na celu skompensowanie tej nieliniowości.

10.3. Cztery procedury pomocnicze

Napisanie programu korzystającego z wielu bloków zmiennych jednolitych i złożonego z wielu niezależnie pisanych części wiąże się z pewnym problemem porządkowym: chcemy zapewnić, aby bloki wykorzystywane w różnych celach były przywiązywane do różnych punktów dowiązania. Twarde zakodowanie numerów bloków w programie w C stwarza ten kłopot, że trudno jest zadbać o zachowanie porządku, gdy część programu przenosi się do innego programu, albo gdy tworzy się bibliotekę procedur przeznaczonych do wykorzystania w wielu aplikacjach. Od wpisywania numerów w instrukcjach lepsze jest zdefiniowanie nazw używanych punktów (przy użyciu `#define`), ale to też jest półśrodek. Jeszcze lepszym rozwiązaniem jest zapamiętanie numerów punktów dowiązania w zmiennych zadekla-

rowanych w aplikacji. Numery te może generować procedura `NewUniformBindingPoint` przedstawiona na listingu 10.5 (najlepiej dopisać ją i pozostałe trzy procedury do pliku `utilities.c`). Za każdym kolejnym wywołaniem procedura ta podaje kolejny numer.

Procedura `glGetIntegerv` wywołana w linii 7 podaje liczbę dostępnych w danej implementacji OpenGL-a punktów dowiązania (czyli długość tablicy przedstawionej na rys. 1.3) w celu `GL_UNIFORM_BUFFER`[7]; w razie, gdyby procedura `NewUniformBindingPoint` została wywołana więcej razy, aplikacja zostanie zatrzymana.

Uwaga: Jeśli aplikacja korzysta z więcej niż jednego kontekstu OpenGL-a, to należy utworzyć osobny licznik wykorzystanych numerów dla każdego kontekstu[8].

Ponieważ uzyskiwanie podstawowych informacji na temat pól w każdym bloku zmiennych jednolitych wykonuje się tak samo, przydaje się do tego procedura `GetAccessToUniformBlock`. Jej parametry wejściowe to: `prog` — identyfikator programu szaderów, `n` — liczba pól, oraz `names` — tablica $n+1$ napisów, z których pierwszy jest nazwą zewnętrzną bloku, a pozostałe są nazwami pól w bloku. Zmienna `*size` otrzymuje wartość równą wielkości bloku w bajtach, do tablicy `ofs` wpisywane są przesunięcia w bajtach poszczególnych pól od początku bloku. Wreszcie, wartość przypisana zmiennej `*bpoint` to numer punktu dowiązania wygenerowany przez procedurę `NewUniformBindingPoint`.

Procedura `glGetUniformBlockIndex` przekazuje indeks bloku zmiennych jednolitych o podanej nazwie, a w razie nieistnienia takiego bloku przekazywany jest wynik o nazwie symbolicznej `GL_INVALID_INDEX`[9]. W takim przypadku procedura `GetAccessToUniformBlock` zawiadamia o niepowodzeniu. W zasadzie to jest powód, aby zakończyć działanie aplikacji, ale w p. 11.5.3 opisałem sytuację, w której warto ponowić próbę.

Listing 10.5. Cztery procedury pomocnicze

C

```
1: static GLint  max_uniform_bp = 0;
2: static GLuint next_uniform_bp = 0;
3:
4: GLuint NewUniformBindingPoint ( void )
5: {
6:   if ( !max_uniform_bp )
7:     glGetIntegerv ( GL_MAX_UNIFORM_BUFFER_BINDINGS, &max_uniform_bp );
8:   if ( next_uniform_bp < max_uniform_bp )
9:     return next_uniform_bp ++;
10:  else
11:    ExitOnError ( "NewUniformBindingPoint" );
12: } /*NewUniformBindingPoint*/
13:
14: char GetAccessToUniformBlock ( GLuint prog, int n, const GLchar **names,
15:                                GLint *size, GLint *ofs, GLuint *bpoint )
```

[7] Specyfikacja OpenGL 4 gwarantuje dostępność co najmniej 36 punktów dowiązania — w komputerze, na którym uruchamiałem aplikacje do tej książki, można korzystać z 84 punktów.

[8] Przypomnijmy, że FreeGLUT domyślnie tworzy nowy kontekst OpenGL-a dla każdego okna i podokna.

[9] Jest to liczba $2^{32}-1$, reprezentowana przez 32 bity o wartości 1.

```
16: {
17:   GLuint ind, *indt;
18:
19:   ind = glGetUniformBlockIndex ( prog, names[0] );
20:   if ( ind == GL_INVALID_INDEX )
21:     return false;
22:   glGetActiveUniformBlockiv ( prog, ind, GL_UNIFORM_BLOCK_DATA_SIZE,
23:                               size );
24:   if ( n > 0 ) {
25:     if ( !(indt = malloc ( n*sizeof(GLuint) )) )
26:       return false;
27:     glGetUniformIndices ( prog, n, &names[1], indt );
28:     glGetActiveUniformsiv ( prog, n, indt, GL_UNIFORM_OFFSET, ofs );
29:     free ( indt );
30:   }
31:   *bpoint = NewUniformBindingPoint ();
32:   glUniformBlockBinding ( prog, ind, *bpoint );
33:   ExitIfGLError ( "GetAccessToUniformBlock" );
34:   return true;
35: } /*GetAccessToUniformBlock*/
36:
37: GLuint NewUniformBuffer ( GLint size, GLuint bp )
38: {
39:   GLuint buf;
40:
41:   glGenBuffers ( 1, &buf );
42:   glBindBufferBase ( GL_UNIFORM_BUFFER, bp, buf );
43:   glBufferData ( GL_UNIFORM_BUFFER, size, NULL, GL_DYNAMIC_DRAW );
44:   ExitIfGLError ( "NewUniformBuffer");
45:   return buf;
46: } /*NewUniformBuffer*/
47:
48: void AttachUniformBlockToBP ( GLuint prog, const GLchar *name, GLuint bp )
49: {
50:   GLuint ind;
51:
52:   ind = glGetUniformBlockIndex ( prog, name );
53:   glUniformBlockBinding ( prog, ind, bp );
54:   ExitIfGLError ( "AttachUniformBlockToBP" );
55: } /*AttachUniformBlockToBP*/
```

Procedura `NewUniformBuffer` tworzy bufor, nadaje mu podaną wielkość i przywiązuje jako UBO do podanego punktu dowiązania.

Zadaniem procedury `AttachUniformBlockToBP` jest przywiązanie zdefiniowanego w programie `prog` bloku zmiennych jednolitych o nazwie `name` do punktu dowiązania `bp`. Założenie jest takie, że blok ten jest zdefiniowany także w innym programie szaderów i potrzebne informacje o nim (przesunięcia pól) zostały wcześniej uzyskane przez wywołaną

dla tego innego programu procedurę `GetAccessToUniformBlock`, która przydzieliła także punkt dowiązania.

10.4. Obsługa bloków zmiennych jednolitych aplikacji

Programy szaderów opisanej tu aplikacji korzystają z dwóch bloków zmiennych jednolitych — bloku `TransBlock` zawierającego macierze przekształceń i bloku `LSBlock` z opisami źródeł światła. Bloki te i bloki wprowadzone w aplikacjach opisanych dalej będziemy jeszcze wielokrotnie modyfikować. Takie przeróbki szaderów pociągają za sobą konieczność dostosowywania aplikacji w C, co może doprowadzić do bałaganu w kodzie i do powstawania trudnych do znalezienia błędów. Aby oszczędzić sobie kłopotów, dobrze jest wyodrębnić części kodu, których zadaniem jest przesyłanie danych do poszczególnych bloków. Dlatego procedury komunikacji z tymi blokami zebrałem w osobnych plikach źródłowych, ukrywając w nich wszystkie dane „wewnętrzne", takie jak tablice przemieszczeń pól. Dzięki temu modyfikacje bloku zmiennych jednolitych będą wymagały zmian we wszystkich szaderach korzystających z tego bloku, ale tylko w jednym pliku źródłowym w C[10].

Dla bloku `TransBlock` utworzyłem dwa pliki źródłowe — nagłówkowy `trans.h` i podstawowy `trans.c`. Pierwszy z tych plików zawiera pokazane na listingu 10.6 deklaracje typu oraz prototypy umieszczonych w drugim pliku procedur z listingu 10.7.

Listing 10.6. Interfejs do bloku zmiennych jednolitych `TransBlock`

C

```
1: typedef struct TransBl {
2:     GLuint  trbuf;
3:     GLfloat mm[16], vm[16], pm[16];
4:     GLfloat eyepos[4];
5:   } TransBl;
6:
7: GLuint GetAccessToTransBlockUniform ( GLuint program_id );
8: void AttachUniformTransBlockToBP ( GLuint program_id );
9: GLuint NewUniformTransBlock ( void );
10:
11: void LoadVPMatrix ( TransBl *trans );
12: void LoadMMatrix ( TransBl *trans, GLfloat mm[16] );
```

Po skompilowaniu i złączeniu programu szaderów zawierającego blok `TransBlock` aplikacja powinna wywołać procedurę `GetAccessToTransBlockUniform`, która odczyta przesunięcia pól w tym bloku, co przygotuje pozostałe procedury do pracy. Poza tym należy zarezerwować dla tego bloku punkt dowiązania w celu `GL_UNIFORM_BUFFER` i dla wszystkich programów szaderów[11], które odwołują się do bloku `TransBlock`, skojarzyć ten blok z tym punktem dowiązania.

[10] W języku C++ można i warto utworzyć obiekt odpowiedniej klasy i do przesyłania danych upoważnić tylko metody tego obiektu. I też warto implementację tej klasy zawrzeć w osobnym pliku źródłowym.

[11] Na razie mamy dwa programy, rysujące obiekt z oświetleniem i bez.

Informacje na temat budowy bloku `TransBlock`, odczytane z programu szaderów przez procedurę `GetAccessToTransBlockUniform`, zostaną zapisane w zmiennych globalnych `trbsize` i `trbofs`, a numer punktu dowiązania będzie zapamiętany w zmiennej `trbbp` (listing 10.7). Zmienne te mają atrybut `static`, dzięki czemu nie są widoczne poza plikiem źródłowym `trans.c`.

Procedura `GetAccessToTransBlockUniform` musi być wywołana z parametrem — identyfikatorem programu zawierającego blok `TransBlock`[12]. W pierwszym wywołaniu procedura za pomocą `GetAccessToUniformBlock` rezerwuje numer punktu dowiązania i odczytuje z programu wielkość bloku i przesunięcia jego pól. Kolejne wywołania (dla innych programów) tylko przywiązują blok `TransBlock` do tego punktu dowiązania.

Wartością powrotną procedury `GetAccessToTransBlockUniform` jest numer punktu dowiązania bloku `TransBlock`. Aplikacja może utworzyć więcej niż jeden bufor z takim blokiem i w trakcie działania zmieniać przywiązany do tego punktu bufor[13]. Zamiast kolejnych

Listing 10.7. Procedury obsługi bloku `TransBlock`

C

```
1:  .... /* dyrektywy #include */
2:
3: #define NTRUOFFS 5
4:
5: static GLuint       trbbp = GL_INVALID_INDEX;
6: static GLint        trbsize, trbofs[NTRUOFFS];
7: static const GLchar *UTBNames[] =
8:   { "TransBlock", "TransBlock.mm", "TransBlock.vm", "TransBlock.pm",
9:     "TransBlock.vpm", "TransBlock.eyepos" };
10:
11: GLuint GetAccessToTransBlockUniform ( GLuint program_id )
12: {
13:   if ( trbbp == GL_INVALID_INDEX )
14:     GetAccessToUniformBlock ( program_id, NTRUOFFS, &UTBNames[0],
15:                               &trbsize, trbofs, &trbbp );
16:   else
17:     AttachUniformBlockToBP ( program_id, UTBNames[0], trbbp );
18:   return trbbp;
19: } /*GetAccessToTransBlockUniform*/
20:
21: void AttachUniformTransBlockToBP ( GLuint program_id )
22: {
23:   AttachUniformBlockToBP ( program_id, UTBNames[0], trbbp );
24: } /*AttachUniformTransBlockToBP*/
```

[12] Opisy tego samego bloku zmiennych jednolitych w różnych programach szaderów powinny być identyczne, nawet jeśli pewne programy nie odwołują się do wszystkich pól w bloku. Teoretycznie można by pominąć końcowe pola, jeśli nie są używane (oczywiście program, z którego odczytujemy przesunięcia pól, musi mieć opis bloku z wszystkimi polami), ale stanowczo odradzam robienia tego.

[13] Zamiast przekazywać numer punktu aplikacji, lepiej byłoby dodać do pliku `trans.c` procedurę opakowującą wywołanie `glBindBufferBase`. Nie zrobiłem tego, bo aplikacja 1B i następne używa tylko jednego bufora z macierzami przekształceń.

```
25:
26: GLuint NewUniformTransBlock ( void )
27: {
28:   return NewUniformBuffer ( trbsize, trbbp );
29: } /*NewUniformTransBlock*/
30:
31: void LoadVPMatrix ( TransBl *trans )
32: {
33:   GLfloat vpm[16];
34:
35:   glBindBuffer ( GL_UNIFORM_BUFFER, trans->trbuf );
36:   M4x4Multf ( vpm, trans->pm, trans->vm );
37:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1], 16*sizeof(GLfloat),
38:                     trans->vm );
39:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[2], 16*sizeof(GLfloat),
40:                     trans->pm );
41:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[3], 16*sizeof(GLfloat), vpm );
42:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[4], 4*sizeof(GLfloat),
43:                     trans->eyepos );
44:   ExitIfGLError ( "LoadVPMatrix" );
45: } /*LoadVPMatrix*/
46:
47: void LoadMMatrix ( TransBl *trans, GLfloat mm[16] )
48: {
49:   if ( mm )
50:     memcpy ( trans->mm, mm, 16*sizeof(GLfloat) );
51:   glBindBuffer ( GL_UNIFORM_BUFFER, trans->trbuf );
52:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[0],
53:                     16*sizeof(GLfloat), trans->mm );
54:   ExitIfGLError ( "LoadMMatrix" );
55: } /*LoadMMatrix*/
```

wywołań procedury `GetAccessToTransBlockUniform` można użyć procedury `AttachUniformTransBlockToBP`, która wykonuje to zadanie.

Procedura `NewUniformTransBlock` tworzy bufor o wielkości odpowiedniej dla tego bloku. Procedury `AttachUniformTransBlockToBP` i `NewUniformTransBlock` wykonują tylko jedną instrukcję, tj. wywołują procedurę `AttachUniformBlockToBP` lub `NewUniformBlockObject`, ale parametrami tych wywołań są wspomniane zmienne statyczne, niewidoczne w pozostałych plikach źródłowych aplikacji. Takie opakowanie danych, wymagające niewiele dodatkowego wysiłku od autora programu, jest niesamowicie opłacalne, gdy realizowany projekt się rozrasta.

Do przesyłania danych służą procedury `LoadVPMatrix` i `LoadMMatrix`, których pierwszym parametrem jest adres zmiennej typu `TransBl`, w której aplikacja przechowuje identyfikator bufora i macierze przekształceń opisujących rzutowanie. Pierwsza z tych procedur przesyła do pól `vm` i `pm` macierze V (przejścia od układu świata do układu obserwatora) i P (przejścia do układu kostki standardowej), oblicza i przesyła do pola `vpm` iloczyn tych ma-

cierzy, a w liniach 42–43 przesyła do pola eyepos położenie obserwatora. Druga procedura przesyła do bufora z blokiem TransBlock macierz przejścia od układu modelu do układu świata podaną jako drugi parametr lub zapamiętaną wcześniej w strukturze wskazywanej przez pierwszy parametr.

Obsługa bloku LSBlock też jest opisana w dwóch plikach źródłowych: nagłówkowym lights.h i podstawowym lights.c. Procedury przesyłania danych do tego bloku aplikacja powinna przygotować, wywołując procedurę GetAccessToLightBlockUniform z parametrem — identyfikatorem skompilowanego i złączonego programu szaderów odwołującego się do tego bloku. W zmiennych statycznych lsbbp, lsbsize i lsbofs zostaną zapamiętane punkt dowiązania, wielkość bufora i tablica przesunięć pól względem początku bloku. Procedury AttachUniformLightBlockToBP i NewUniformLightBlock służą do związania bloku LSBlock w kolejnych programach szaderów z punktem dowiązania zarezerwowanym dla tego bloku oraz do utworzenia bufora, w którym blok ma być przechowywany — analogicznie do przypadku bloku TransBlock.

Listing 10.8. Interfejs do bloku zmiennych jednolitych LSBlock

C

```
1: #define MAX_NLIGHTS 8
2:
3: typedef struct LSPar {
4:     GLfloat position[4];
5:     GLfloat ambient[3];
6:     GLfloat direct[3];
7:     GLfloat attenuation[3];
8:   } LSPar;
9:
10: typedef struct LightBl {
11:     GLuint lsbuf;
12:     GLuint nls, mask;
13:     LSPar  ls[MAX_NLIGHTS];
14:   } LightBl;
15:
16: GLuint GetAccessToLightBlockUniform ( GLuint program_id );
17: void AttachUniformLightBlockToBP ( GLuint program_id );
18: GLuint NewUniformLightBlock ( void );
19:
20: void SetLightPosition ( LightBl *light, int l, GLfloat pos[4] );
21: void SetLightAmbient ( LightBl *light, int l, GLfloat amb[3] );
22: void SetLightDiffuse ( LightBl *light, int l, GLfloat dif[3] );
23: void SetLightAttenuation ( LightBl *light, int l,
24:                             GLfloat at3[3] );
25: void SetLightOnOff ( LightBl *light, int l, char on );
```

W zmiennej typu LightBl aplikacja będzie przechowywać identyfikator bufora z blokiem LSBlock i kopię danych w tym bloku, przy czym w tym przypadku uznałem, że wygod-

niej będzie, aby poszczególne atrybuty źródeł światła były parametrami osobnych procedur, które oprócz przesłania danych do pamięci GPU zapiszą je także w tej zmiennej.

Zwróćmy uwagę na sposób obliczania przesunięcia miejsca, do którego należy przesłać dane, względem początku bufora. Procedura `GetAccessToUniformBlock` wywołana w liniach 15–16 na listingu 10.9 wpisała do kolejnych elementów tablicy `lsbofs` przesunięcia pól `nls`, `mask`, `ls[0].ambient`, `ls[0].direct`, `ls[0].position`, `ls[0].attenuation` i `ls[1].ambient`. Różnica przesunięć pól `ls[1].ambient` i `ls[0].ambient` jest rozmiarem (w bajtach) miejsca zajmowanego przez każdy element tablicy `ls`, tj. strukturę `LSPar`. Zatem aby obliczyć przesunięcie dowolnego pola l-tego elementu tej tablicy, należy do przesunięcia odpowiedniego pola w elemencie zerowym dodać iloczyn indeksu l elementu i różnicy `lsbofs[6]-lsbofs[2]`. To obliczenie wykonują instrukcje w liniach 43, 56, 69 i 83.

Uwaga: Choć (pomijając pole `lsbuf`) struktura `LightBl` jest zbudowana tak samo jak blok zmiennych jednolitych `LSBlock` szadera na listingu 10.4, *nie można* liczyć na to, że kolejne pola obu struktur mają takie same przesunięcia względem pola `nls`[14] i dlatego nie wolno przesyłać danych do UBO „hurtem"; trzeba przesyłać dane pracowicie po jednym polu.

Listing 10.9. Procedury obsługi bloku `LSBlock`

C

```
1:  .... /* dyrektywy #include */
2:
3:  #define NLSUOFFS 7
4:
5:  static GLuint lsbbp = GL_INVALID_INDEX;
6:  static GLint  lsbsize, lsbofs[NLSUOFFS];
7:  static const GLchar *ULSNames[] =
8:    { "LSBlock", "LSBlock.nls", "LSBlock.mask", "LSBlock.ls[0].ambient",
9:      "LSBlock.ls[0].direct", "LSBlock.ls[0].position",
10:     "LSBlock.ls[0].attenuation", "LSBlock.ls[1].ambient" };
11:
12: GLuint GetAccessToLightBlockUniform ( GLuint program_id )
13: {
14:   if ( lsbbp == GL_INVALID_INDEX )
15:     GetAccessToUniformBlock ( program_id, NLSUOFFS, &ULSNames[0],
16:                               &lsbsize, lsbofs, &lsbbp );
17:   else
18:     AttachUniformBlockToBP ( program_id, ULSNames[0], lsbbp );
19:   return lsbbp;
20: } /*GetAccessToLightBlockUniform*/
21:
22: void AttachUniformLightBlockToBP ( GLuint program_id )
```

[14] Kompilator GLSL-a ma swoje zasady przydzielania adresów pól struktur, inne niż kompilator C. Przy tym istnieją kwalifikatory układu zmiennych strukturalnych (np. często stosowany dla bloków zmiennych jednolitych kwalifikator `layout(std140)`), które wręcz wykluczają takie same przesunięcia pól struktury w językach GLSL i C. Na podstawie reguł układu określonego przez ten kwalifikator można *obliczyć* przesunięcia pól względem początku struktury, zamiast wywoływać procedurę `glGetActiveUniformsiv`.

```
23: {
24:   AttachUniformBlockToBP ( program_id, ULSNames[0], lsbbp );
25: } /*AttachUniformLightBlockToBP*/
26:
27: GLuint NewUniformLightBlock ( void )
28: {
29:   return NewUniformBuffer ( lsbsize, lsbbp );
30: } /*NewUniformLightBlock*/
31:
32: void SetLightPosition ( LightBl *light, int l, GLfloat pos[4] )
33: {
34:   GLint   ofs;
35:   GLfloat w, *p;
36:
37:   if ( l < 0 || l >= MAX_NLIGHTS )
38:     return;
39:   memcpy ( p = light->ls[l].position, pos, 4*sizeof(GLfloat) );
40:   w = p[3];
41:   if ( w != 0.0 && w != 1.0 )
42:     p[0] /= w, p[1] /= w, p[2] /= w, p[3] = 1.0;
43:   ofs = l*(lsbofs[6]-lsbofs[2]) + lsbofs[4];
44:   glBindBuffer ( GL_UNIFORM_BUFFER, light->lsbuf );
45:   glBufferSubData ( GL_UNIFORM_BUFFER, ofs, 4*sizeof(GLfloat), p );
46:   ExitIfGLError ( "SetLightPosition" );
47: } /*SetLightPosition*/
48:
49: void SetLightAmbient ( LightBl *light, int l, GLfloat amb[3] )
50: {
51:   GLint ofs;
52:
53:   if ( l < 0 || l >= MAX_NLIGHTS )
54:     return;
55:   memcpy ( light->ls[l].ambient, amb, 4*sizeof(GLfloat) );
56:   ofs = l*(lsbofs[6]-lsbofs[2]) + lsbofs[2];
57:   glBindBuffer ( GL_UNIFORM_BUFFER, light->lsbuf );
58:   glBufferSubData ( GL_UNIFORM_BUFFER, ofs, 3*sizeof(GLfloat), amb );
59:   ExitIfGLError ( "SetLightAmbient" );
60: } /*SetLightAmbient*/
61:
62: void SetLightDirect ( LightBl *light, int l, GLfloat dif[3] )
63: {
64:   GLint ofs;
65:
66:   if ( l < 0 || l >= MAX_NLIGHTS )
67:     return;
68:   memcpy ( light->ls[l].direct, dif, 4*sizeof(GLfloat) );
69:   ofs = l*(lsbofs[6]-lsbofs[2]) + lsbofs[3];
```

```
70:   glBindBuffer ( GL_UNIFORM_BUFFER, light->lsbuf );
71:   glBufferSubData ( GL_UNIFORM_BUFFER, ofs, 3*sizeof(GLfloat), dif );
72:   ExitIfGLError ( "SetLightDirect" );
73: } /*SetLightDirect*/
74:
75: void SetLightAttenuation ( LightBl *light, int l,
76:                            GLfloat atn[3] )
77: {
78:   GLint ofs;
79:
80:   if ( l < 0 || l >= MAX_NLIGHTS )
81:     return;
82:   memcpy ( light->ls[l].attenuation, atn, 3*sizeof(GLfloat) );
83:   ofs = l*(lsbofs[6]-lsbofs[2]) + lsbofs[5];
84:   glBindBuffer ( GL_UNIFORM_BUFFER, light->lsbuf );
85:   glBufferSubData ( GL_UNIFORM_BUFFER, ofs, 3*sizeof(GLfloat), atn );
86:   ExitIfGLError ( "SetLightAttenuation" );
87: } /*SetLightAttenuation*/
88:
89: void SetLightOnOff ( LightBl *light, int l, char on )
90: {
91:   GLuint mask;
92:
93:   if ( l < 0 || l >= MAX_NLIGHTS )
94:     return;
95:   mask = 0x01 << l;
96:   if ( on ) {
97:     light->mask |= mask;
98:     if ( l >= light->nls )
99:       light->nls = l+1;
100:  }
101:  else {
102:    light->mask &=  mask;
103:    for ( mask = 0x01 << (light->nls-1); mask; mask >>= 1 ) {
104:      if ( light->mask & mask )
105:        break;
106:      else
107:        light->nls --;
108:    }
109:  }
110:  glBindBuffer ( GL_UNIFORM_BUFFER, light->lsbuf );
111:  glBufferSubData ( GL_UNIFORM_BUFFER, lsbofs[0], sizeof(GLuint),
112:                    &light->nls );
113:  glBufferSubData ( GL_UNIFORM_BUFFER, lsbofs[1], sizeof(GLuint),
114:                    &light->mask );
115:  ExitIfGLError ( "SetLightOnOff" );
116: } /*SetLightOnOff*/
```

Pierwsze dwa parametry procedur nadających wartości atrybutom źródeł światła to identyfikator bufora i adres zmiennej typu `LightBl`. Trzeci atrybut jest numerem źródła światła, a czwarty tablicą z odpowiednim wektorem: współrzędnych położenia źródła światła, intensywności światła rozproszonego w otoczeniu I_i^{amb}, intensywności oświetlenia kierunkowego I_i^{dir} albo intensywności światła emitowanego I_i^{em} i współczynników a, b, c występujących we wzorze (10.2).

Procedura `SetLightOnOff` przypisuje wartość `0` lub `1` odpowiedniemu bitowi maski, po czym znajduje pozycję najbardziej znaczącego bitu maski i przesyła dane do bufora. Dzięki temu w obliczeniach koloru fragmentu wykonywanie pętli w liniach 58–77 na listingu 10.4 zostanie zakończone po uwzględnieniu ostatniego „włączonego" źródła światła.

10.5. Aplikacja pierwsza B

Listing 10.10 przedstawia procedurę ładowania szaderów. Jest ona znacznie prostsza niż procedura z listingu 7.4, bo znajdowaniem dostępu do bloków zmiennych jednolitych zajmują się opisane wcześniej procedury, wywoływane w liniach 20–22. Ponadto zamiast za pomocą osobnych wywołań procedury kompilacji szadery są kompilowane w pętli w liniach 16–17, dzięki czemu kod procedury jest krótszy i chyba trochę czytelniejszy.

Listing 10.10. Procedura `LoadMyShaders`

C

```
1:  GLuint  program_id[2];
2:  TransBl trans;
3:  LightBl light;
4:
5:  void LoadMyShaders ( void )
6:  {
7:    static const char *filename[] =
8:      { "app1b0.vert.glsl", "app1b0.frag.glsl",
9:        "app1b1.vert.glsl", "app1b1.geom.glsl", "app1b1.frag.glsl" };
10:   static const GLenum shtype[5] =
11:     { GL_VERTEX_SHADER, GL_FRAGMENT_SHADER,
12:       GL_VERTEX_SHADER, GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER };
13:   GLuint shader_id[5];
14:   int    i;
15:
16:   for ( i = 0; i < 5; i++ )
17:     shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
18:   program_id[0] = LinkShaderProgram ( 2, &shader_id[0], "0" );
19:   program_id[1] = LinkShaderProgram ( 3, &shader_id[2], "1" );
20:   GetAccessToTransBlockUniform ( program_id[1] );
21:   GetAccessToLightBlockUniform ( program_id[1] );
22:   AttachUniformTransBlockToBP ( program_id[0] );
23:   for ( i = 0; i < 5; i++ )
24:     glDeleteShader ( shader_id[i] );
25:   ExitIfGLError ( "LoadMyShaders" );
```

```
} /*LoadMyShaders*/
```

W procedurach inicjalizacji pokazanych na listingu 10.11 są dopisane instrukcje inicjalizacji świateł. Dodatkowo przed wpisaniem pierwszych nietrywialnych danych zmienne `trans` i `light` są wypełniane zerami. Tak trzeba. Do sprzątania doszło kasowanie drugiego programu szaderów i bufora z opisem źródeł światła.

Procedura `InitLights` przesyła dane opisujące *jedno* źródło światła, o numerze 0, położone daleko od obiektu, i włącza to światło. Parametry tego źródła światła są przesyłane do pamięci GPU za pomocą procedur podanych na listingu 10.9.

Listing 10.11. Inicjalizacja i sprzątanie

C

```
void InitLights ( void )
{
  GLfloat amb0[3] = { 0.2, 0.2, 0.3 };
  GLfloat dir0[3] = { 0.8, 0.8, 0.8 };
  GLfloat pos0[4] = { 0.0, 1.0, 1.0, 0.0 };
  GLfloat atn0[3] = { 1.0, 0.0, 0.0 };

  SetLightAmbient ( &light, 0, amb0 );
  SetLightDirect ( &light, 0, dir0 );
  SetLightPosition ( &light, 0, pos0 );
  SetLightAttenuation ( &light, 0, atn0 );
  SetLightOnOff ( &light, 0, true );
} /*InitLights*/

void InitMyWorld ( int argc, char *argv[], int width, int height )
{
  LoadMyShaders ();
  memset ( &trans, 0, sizeof(TransBl) );
  memset ( &light, 0, sizeof(LightBl) );
  trans.trbuf = NewUniformTransBlock ();
  light.lsbuf = NewUniformLightBlock ();
  TimerInit ();
  SetupModelMatrix ( model_rot_axis, model_rot_angle );
  InitViewMatrix ();
  ConstructIcosahedronVAO ();
  InitLights ();
  ResizeMyWorld ( width, height );
} /*InitMyWorld*/

void DeleteMyWorld ( void )
{
  glUseProgram ( 0 );
  glDeleteProgram ( program_id[0] );
  glDeleteProgram ( program_id[1] );
```

```
35:   glDeleteBuffers ( 1, &trans.trbuf );
36:   glDeleteBuffers ( 1, &light.lsbuf );
37:   glDeleteVertexArrays ( 1, &icos_vao );
38:   glDeleteBuffers ( 3, icos_vbo );
39:   ExitIfGLError ( "DeleteMyWorld" );
40: } /*DeleteMyWorld*/
```

Na listingu 10.12 są pokazane modyfikacje mające na celu obsługę zmian przekształceń wierzchołków. Procedura `SetupModelMatrix` oblicza macierz obrotu obiektu i przesyła ją do bufora. Procedura `InitViewMatrix` nadaje początkowe położenie obserwatorowi (biorąc je ze zmiennej `viewer_pos`) i oblicza macierz przesunięcia odpowiadającego temu położeniu.

Listing 10.12. Procedury przetwarzania przekształceń

C

```
1: void SetupModelMatrix ( float axis[3], double angle )
2: {
3:   GLfloat mm[16];
4:
5:   M4x4RotateVf ( mm, axis[0], axis[1], axis[2], angle );
6:   LoadMMatrix ( &trans, mm );
7: } /*SetupModelMatrix*/
8:
9: void InitViewMatrix ( void )
10: {
11:   memcpy ( trans.eyepos, viewer_pos0, 4*sizeof(GLfloat) );
12:   M4x4Translatef ( trans.vm,
13:                    -viewer_pos0[0], -viewer_pos0[1], -viewer_pos0[2] );
14:   LoadVPMatrix ( &trans );
15: } /*InitViewMatrix*/
16:
17: void RotateViewer ( int delta_xi, int delta_eta )
18: {
19:   .... /* instrukcje bez zmian */
20:   viewer_rangle = angk;
21:   M4x4RotateVfv ( trans.vm, viewer_rvec, -viewer_rangle );
22:   M4x4MultMTVf ( trans.eyepos, trans.vm, viewer_pos0 );
23:   M4x4InvTranslateMfv ( trans.vm, viewer_pos0 );
24:   LoadVPMatrix ( &trans );
25: } /*RotateViewer*/
```

Przemieszczenie obserwatora konstruowane przez procedurę `RotateViewer` jest obrotem wokół osi przechodzącej przez początek układu współrzędnych świata. W linii 22 wektor współrzędnych położenia obserwatora w układzie obserwatora jest mnożony przez transpozycję macierzy obrotu będącego etapem tego przekształcenia. Ponieważ macierz ta jest ortogonalna, jej transpozycja jest jej odwrotnością, zatem następuje tu obliczenie położenia obserwatora w układzie świata. Przesyłaniem danych do pamięci GPU zajmuje się procedura

`LoadVPMatrix`, dzięki czemu instrukcje konstruujące przekształcenie w tym miejscu nie są zaciemniane szczegółami budowy bloku zmiennych jednolitych.

Trzeba jeszcze zmienić procedurę rysowania dwudziestościanu i procedurę obsługi poleceń wydawanych za pomocą klawiatury, w sposób pokazany na listingu 10.13. Dodatkowy parametr procedury `DrawIcosahedron` wybiera sposób rysowania ścian dwudziestościanu — przy użyciu pierwszego programu, czyli tak jak wcześniej, czy przy użyciu drugiego programu, z oświetleniem. Krawędzie i wierzchołki rysujemy przy użyciu pierwszego programu, bo drugi jest dostosowany tylko do trójkątów.

Procedura `ProcessCharCommand` po napisaniu litery `L` albo `l` zmienia wartość zmiennej globalnej `enlight` i przekazuje wartość `true`, co sygnalizuje potrzebę wykonania nowego obrazu w oknie (zob. listing 7.10).

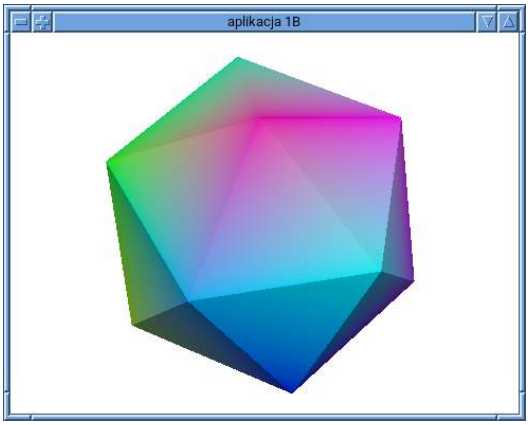
Listing 10.13. Procedury rysowania i włączania/wyłączania oświetlenia

C

```
char enlight = true;

void DrawIcosahedron ( int opt, char enlight )
{
  glBindVertexArray ( icos_vao );
  switch ( opt ) {
case 0:    /* wierzchołki */
    glUseProgram ( program_id[0] );
    .... /* instrukcje bez zmian */
    break;
case 1:    /* krawędzie */
    glUseProgram ( program_id[0] );
    .... /* instrukcje bez zmian */
    break;
default:   /* ściany */
    glUseProgram ( program_id[enlight ? 1 : 0] );
    .... /* instrukcje bez zmian */
    break;
  }
  glBindVertexArray ( 0 );
} /*DrawIcosahedron*/

char ProcessCharCommand ( char charcode )
{
  int oldoption;

  oldoption = option;
  switch ( toupper ( charcode ) ) {
case 'L': enlight = !enlight;
    return true;
case 'W': option = 0;  break;  /* przełączamy na wierzchołki */
case 'K': option = 1;  break;  /* przełączamy na krawędzie */
```

```
33: case 'S': option = 2;  break;  /* przełączamy na ściany */
34:  default:
35:     return false;              /* ignorujemy wszystkie inne klawisze */
36:   }
37:   return option != oldoption;
38: } /*ProcessCharCommand*/
```

Jeszcze jedna zmiana jest potrzebna w procedurze `ResizeMyWorld` — zamiast bezpośrednich wywołań procedur OpenGL-a, przywiązujących bufor i przesyłających dane, macierz P obliczona przez procedurę `M4x4Frustumf` (zapamiętana w zmiennej `trans.pm`) jest razem z macierzą V przejścia do układu obserwatora i położeniem obserwatora przesyłana do pamięci GPU przez procedurę `LoadVPMatrix`.



Rysunek 10.1. Okno aplikacji pierwszej B

10.6. Uzupełnienia

10.6.1. Cieniowanie Gourauda i Phonga

W opisanej tu aplikacji kolory pikseli są obliczane przez szader fragmentów, który zawiera pełną implementację modelu oświetlenia. Możliwe jest też obliczanie koloru wierzchołków przez jeden z szaderów części przedniej potoku przetwarzania grafiki — dla brył wielościennych może to robić szader geometrii, który przetwarza wierzchołki trójkąta i dysponuje wektorem normalnym płaszczyzny ściany. Szader geometrii mógłby obliczyć kolory wszystkich wierzchołków trójkąta; w procesie rasteryzacji kolory te zostałyby poddane interpolacji i zadaniem szadera fragmentów byłoby tylko przepisanie koloru fragmentu z wejścia na wyjście[15]. Ponieważ liczba fragmentów jest zwykle od dwóch do pięciu rzędów wielkości większa

[15] Można by tu użyć szadera fragmentów z listingu 7.2.

od liczby wierzchołków, takie rozwiązanie, zwane **cieniowaniem Gourauda**, może działać szybciej.

Zastosowana w aplikacji metoda, w której wektor normalny jest interpolowany między wierzchołkami trójkąta, normalizowany i podstawiany do modelu oświetlenia dla każdego fragmentu (tj. piksela), ma nazwę **cieniowania Phonga**. Jakość obrazów otrzymanych przy użyciu obu metod cieniowania, dla lambertowskiego modelu oświetlenia, niewiele się różni. Ale wykonywanie pełnych obliczeń oświetlenia dla każdego fragmentu jest koniecznością, jeśli używamy bardziej skomplikowanego modelu oświetlenia (w którym powierzchnie nie muszą być matowe) lub gdy na powierzchnię nakładamy teksturę. Tym zajmiemy się w drugiej aplikacji.

10.6.2. Mgła

Można wprowadzić na obrazie **mgłę** lub inny czynnik, który uzależnia kolor piksela od odległości punktu rysowanej powierzchni od obserwatora. W tym celu trzeba ustalić pewną funkcję, której argumentem jest ta odległość (czyli w przybliżeniu grubość warstwy mgły między obserwatorem a punktem) i której wartość określa proporcję, w jakiej będzie zmieszany kolor oświetlonego punktu powierzchni z kolorem mgły. Wzorując się[16] na starym OpenGL-u, można zaproponować trzy takie funkcje, określone wzorami

$$f_1(z) = \min\left(\mathrm{e}^{d(a+z)}, 1\right), \quad f_2(z) = \mathrm{e}^{-d(a+z)^2}, \quad f_3(z) = \min\left(\max\left(\frac{b+z}{b-a}, 0\right), 1\right).$$

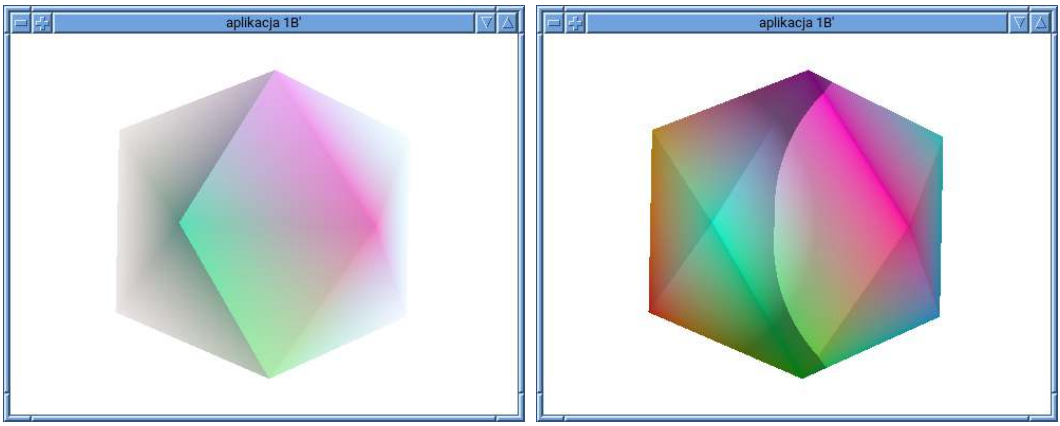
Przyjmują one wartości z przedziału $[0,1]$. Argumentem tych funkcji jest (ujemna) współrzędna z punktu w układzie obserwatora, a (dodatnie) parametry d, a i b, odpowiadające za gęstość mgły, mają wartości nadane przez aplikację. Można przyjąć początkowo wartości parametrów a i b równe parametrom `near` i `far` użytym do określenia bryły widzenia dla rzutu perspektywicznego i ewentualnie, na podstawie eksperymentów, zmodyfikować je.

Jeśli mgła ma określony kolor (albo jest to czarny dym), to kasując tło przed narysowaniem obiektów, trzeba wybrać kolor odpowiadający nieskończenie (lub maksymalnie) grubej warstwie mgły. Po obliczeniu intensywności L światła odbitego od punktu powierzchni na podstawie wzoru (10.1) końcowy kolor, *przed* dokonaniem korekcji gamma, szader fragmentów obliczy ze wzoru

$$L^{\mathrm{out}} = f(z)L + \big(1 - f(z)\big)L^{\mathrm{fog}},$$

w którym f jest wybraną funkcją, a L^{fog} oznacza kolor mgły. Przykładowy obrazek otrzymany w ten sposób jest pokazany z lewej strony rysunku 10.2. Opisana tu technika wytwarzania mgły na obrazie ma angielską nazwę *depth cueing*.

[16] niedosłownie



Rysunek 10.2. Obiekt we mgle i obiekt oświetlony reflektorami

10.6.3. Reflektory

Źródła światła położone w skończonej odległości od sceny lub wewnątrz niej nie muszą świecić jednakowo we wszystkie strony; można określić „reflektor" (*spotlight*), który wysyła światło tylko wewnątrz pewnego stożka. W tym celu trzeba dodać dwa atrybuty źródła światła: wektor kierunkowy osi reflektora i należący do przedziału $(0, \pi)$ kąt α między tą osią a tworzącą stożka. Odpowiednio skierowanym reflektorem można oświetlić dowolnie wybrany fragment sceny.

Zamiast wektora kierunkowego można podać punkt na osi reflektora i wtedy wektor kierunkowy otrzymamy, odejmując położenie $\boldsymbol{s}$ reflektora od tego punktu; różnicę trzeba unormować, otrzymując jednostkowy wektor kierunkowy, oznaczmy go symbolem $\boldsymbol{w}$. Wektor ten należy przesłać do (odpowiednio rozbudowanego) opisu źródła światła w pamięci GPU (czyli umieścić go przykładowo w dodatkowym polu `w` struktury `LSPar` w bloku zmiennych jednolitych `LSBlock`). W kolejnym dodatkowym polu trzeba umieścić liczbę $\cos\alpha$.

Punkt $\boldsymbol{p}$ odpowiadający przetwarzanemu fragmentowi powierzchni jest oświetlony przez reflektor, jeśli leży wewnątrz stożka. Jest tak wtedy, gdy kąt β między wektorami $\boldsymbol{w}$ a $\boldsymbol{p}-\boldsymbol{s}$ jest mniejszy niż α, czyli gdy $\cos\beta > \cos\alpha$. To ma miejsce wtedy, gdy

$$\langle \boldsymbol{w}, \boldsymbol{p}-\boldsymbol{s}\rangle > \|\boldsymbol{p}-\boldsymbol{s}\|\cos\alpha.$$

Długość wektora $\boldsymbol{p}-\boldsymbol{s}$ i jego iloczyn skalarny z wektorem $\boldsymbol{w}$ szader fragmentów może obliczać za pomocą funkcji `length` i `dot`.

Światło reflektora nie musi mieć stałej intensywności wewnątrz stożka; może ono być najsilniejsze na osi stożka i słabnąć ze wzrostem kąta β. Sposób dostępny w starym OpenGL-u (łatwy do odtworzenia w nowym) jest taki, że zamiast wzoru (10.2) jest używany wzór

$$I_i^{\mathrm{dir}} = \max\{\cos^e\beta, 0\}\frac{I_i^{\mathrm{em}}}{at^2+bt+c},$$

w którym występuje dodatkowy atrybut reflektora — wykładnik e, określający, jak szybko światło słabnie ze wzrostem kąta β (jeśli $e = 0$, to światło nie słabnie). Jeśli użyjemy tego

wzoru, to nasz reflektor nie oświetla takich punktów $\boldsymbol{p}$, że $\beta > \pi/2$. Ale lepsze[17] efekty można otrzymać za pomocą wzoru

$$I_i^{\text{dir}} = \max\left\{\left(\frac{\cos\beta - \cos\alpha}{1 - \cos\alpha}\right)^e, 0\right\}\frac{I_i^{\text{em}}}{at^2 + bt + c},$$

przy użyciu którego powstał obrazek na rysunku 10.2 po prawej stronie.

Rozbudowywanie modeli, na przykład zamiana punktowych źródeł światła na reflektory, ma swoją cenę. Składa się na nią więcej pracy dla autora programu[18] oraz wydłużenie czasu obliczeń — „uniwersalny" szader nie musi wykonywać opisanych tu obliczeń, jeśli źródło światła nie jest reflektorem, ale zawsze musi co najmniej sprawdzać, że nie musi. W świetle tego spostrzeżenia warto szadery dostosowywać do konkretnych potrzeb i nie rozbudowywać ich „na zapas".

10.6.4. *Powiększanie danych

Ważną rolę w aplikacjach graficznych pełni **powiększanie danych** (*data amplification*), czyli generowanie przez GPU opisów skomplikowanych obiektów[19] lub powielanie obiektów na podstawie stosunkowo niewielkiej ilości danych przesłanych przez CPU, a także jednoczesne wykonywanie wielu obrazów. Powiększanie danych ma na celu pełne wykorzystanie mocy obliczeniowej GPU, znacznie większej niż CPU, przy jednoczesnej minimalizacji przesyłania danych przez wąskie gardło, jakim jest magistrala łącząca oba procesory w komputerze.

W OpenGL-u powiększanie danych jest możliwe w wielu etapach potoku przetwarzania grafiki — w etapie pobierania wierzchołków (przez rysowanie wielu instancji obiektów) oraz w szaderach rozdrabniania, geometrii i fragmentów, a także w pracujących poza potokiem szaderach obliczeniowych. Tu przedstawię dwa najprostsze (choć nie jedyne) sposoby zastosowania szaderów geometrii do powiększania danych.

Dane mogą być powielone przez podanie kwalifikatora układu wejścia `invocations=`*n*; szader geometrii z takim kwalifikatorem dla każdego wierzchołka, odcinka lub trójkąta zostanie wywołany w *n* instancjach, tj. wątkach obliczeniowych. Listingi 10.14, 10.15 i 10.16 przedstawiają szadery wierzchołków i geometrii, których można użyć w aplikacji 1B, dodając dwa nowe programy szaderów. Pierwszy program składa się z szaderów z listingów 10.14, 10.15 i 10.4, a w drugim jest użyty szader geometrii z listingu 10.16.

Zadaniem szadera wierzchołków w nowych programach jest obliczenie współrzędnych wierzchołka w układzie *świata* i przekazanie atrybutu dodatkowego, koloru. Pole `Position` zmiennej wyjściowej, jako zbędne, zostało usunięte, ponieważ informacja wcześniej przekazywana w tym polu (współrzędne wierzchołka w układzie świata) jest teraz wyprowadzana w zmiennej `gl_Position`. W linii `16` szader dzieli współrzędne jednorodne wierzchołka przez współrzędną wagową, co gwarantuje otrzymanie przez szadery geometrii współrzęd-

[17] moim skromnym zdaniem

[18] Akurat to jest przyjemność.

[19] CPU może przygotować opis kształtu terenu górzystego, a GPU może odpowiadać za pokrycie go kamieniami. Nie ma sensu produkowanie przez CPU opisów wszystkich kamieni w Tatrach.

nych kartezjańskich, także wtedy, gdy aplikacja po (ewentualnych dalszych) modyfikacjach będzie rysować obiekty, których wierzchołki mają współrzędne wagowe różne od 1.

Kwalifikator wejścia szadera geometrii w linii 3 na listingu 10.15 sprawia, że każdy trójkąt zostanie powielony czterokrotnie. Numer „egzemplarza" trójkąta, od 0 do 3, jest wartością zmiennej wbudowanej `gl_InvocationID`. Poszczególne egzemplarze mają być poprzesuwane względem oryginalnego trójkąta, na różne odległości, w kierunku prostopadłym do jego płaszczyzny.

Listing 10.14. Szader wierzchołków współpracujący z szaderami na listingach 10.15 i 10.16

GLSL

```
1: #version 420
2:
3: layout(location=0) in vec4 in_Position;
4: layout(location=1) in vec3 in_Colour;
5:
6: layout(location=0) out vec3 Colour;
7:
8: uniform TransBlock { .... } trb;  /* blok identyczny jak na listingu 10.2 */
9:
10: void main ( void )
11: {
12:   vec4 Pos;
13:
14:   Colour = in_Colour;
15:   Pos = trb.mm * in_Position;
16:   gl_Position = vec4 ( Pos.xyz/Pos.w, 1.0 );
17: } /*main*/
```

W liniach 17–19 szader oblicza jednostkowy wektor normalny oryginalnego trójkąta, będący też wektorem normalnym trójkąta przesuniętego. Wielkość przesunięcia jest obliczana w linii 20 na podstawie numeru egzemplarza trójkąta; wektor przesunięcia `t` jest następnie dodawany do wektora współrzędnych każdego wierzchołka w linii 22, po czym w linii 23 następuje przejście od układu współrzędnych świata do układu kostki standardowej.

Listing 10.15. Szader geometrii powielający trójkąty

GLSL

```
1: #version 420
2:
3: layout(triangles,invocations=4) in;
4: layout(location=0) in vec3 Colour[];
5:
6: layout(triangle_strip,max_vertices=3) out;
7: out FVertex { .... } Out;          /* blok identyczny jak na listingu 10.3 */
8:
9: uniform TransBlock { .... } trb;  /* blok identyczny jak na listingu 10.2 */
10:
```

```
11: void main ( void )
12: {
13:   int  i;
14:   vec3 v1, v2, nv;
15:   vec4 Pos, t;
16:
17:   v1 = gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz;
18:   v2 = gl_in[2].gl_Position.xyz - gl_in[0].gl_Position.xyz;
19:   nv = normalize ( cross ( v1, v2 ) );
20:   t = vec4 ( ((gl_InvocationID+1)*0.05)*nv, 0.0 );
21:   for ( i = 0; i < 3; i++ ) {
22:     Pos = gl_in[i].gl_Position + t;
23:     gl_Position = trb.vpm * Pos;
24:     Out.Position = Pos.xyz;
25:     Out.Normal = nv;
26:     Out.Colour = Colour[i];
27:     EmitVertex ();
28:   }
29:   EndPrimitive ();
30: } /*main*/
```

Szader geometrii na listingu 10.16 zamiast oryginalnego trójkąta wyprowadza trzy trójkąty będące jego częściami. Wektor normalny wszystkich tych części jest taki sam, jest on obliczany przez instrukcje w liniach 24–26, po czym w liniach 27–29 następuje obliczenie środków boków, a w liniach 30–31 jest znajdowany środek ciężkości oryginalnego trójkąta. W taki sam sposób w liniach 32–35 są interpolowane kolory podane dla wierzchołków.

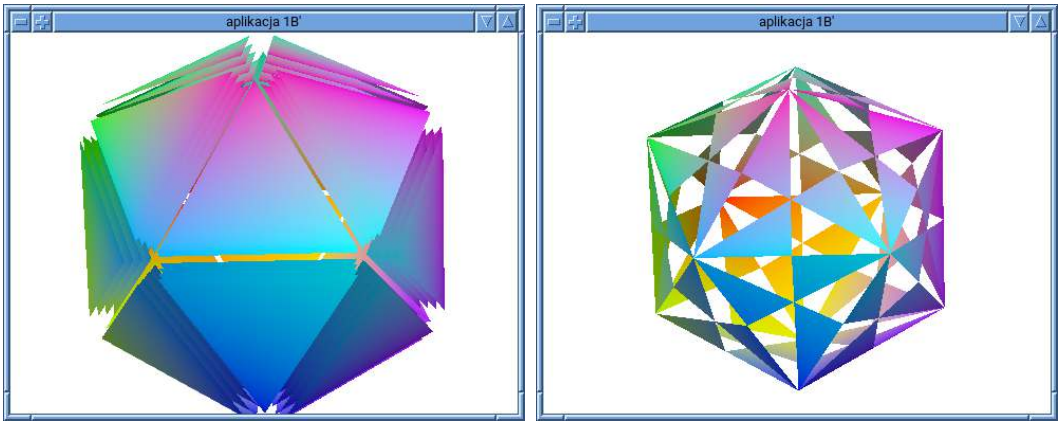
Listing 10.16. Szader geometrii wyprowadzający części trójkąta

GLSL

```
1: #version 420
2:
3: layout(triangles) in;
4: layout(location=0) in vec3 Colour[];
5:
6: layout(triangle_strip,max_vertices=9) out;
7: out FVertex { .... } Out;         /* blok identyczny jak na listingu 10.3 */
8:
9: uniform TransBlock { .... } trb;  /* blok identyczny jak na listingu 10.2 */
10:
11: #define OUT_VERTEX(p,c) \
12:   Out.Normal = nv; \
13:   Out.Position = p.xyz; \
14:   Out.Colour = c; \
15:   gl_Position = trb.vpm * p; \
16:   EmitVertex ();
17:
18: void main ( void )
```

```
19: {
20:   int  i;
21:   vec3 v1, v2, nv, acol[4];
22:   vec4 apos[4];
23:
24:   v1 = gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz;
25:   v2 = gl_in[2].gl_Position.xyz - gl_in[0].gl_Position.xyz;
26:   nv = normalize ( cross ( v1, v2 ) );
27:   apos[0] = 0.5*(gl_in[0].gl_Position + gl_in[1].gl_Position);
28:   apos[1] = 0.5*(gl_in[1].gl_Position + gl_in[2].gl_Position);
29:   apos[2] = 0.5*(gl_in[2].gl_Position + gl_in[0].gl_Position);
30:   apos[3] = (gl_in[0].gl_Position + gl_in[1].gl_Position +
31:              gl_in[2].gl_Position)/3.0;
32:   acol[0] = 0.5*(Colour[0] + Colour[1]);
33:   acol[1] = 0.5*(Colour[1] + Colour[2]);
34:   acol[2] = 0.5*(Colour[2] + Colour[0]);
35:   acol[3] = (Colour[0] + Colour[1] + Colour[2])/3.0;
36:   for ( i = 0; i < 3; i++ ) {
37:     OUT_VERTEX ( gl_in[i].gl_Position, Colour[i] )
38:     OUT_VERTEX ( apos[i], acol[i] )
39:     OUT_VERTEX ( apos[3], acol[3] )
40:     EndPrimitive ();
41:   }
42: } /*main*/
```

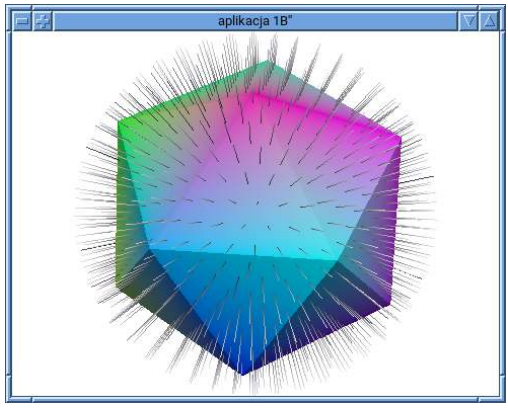
Pętla w liniach 36–41 wyprowadza kolejno wybrane części trójkąta; każda z nich ma jeden wierzchołek oryginalnego trójkąta, jeden wierzchołek w środku jego boku i jeden wierzchołek w jego środku ciężkości. Po wyprowadzeniu każdej części (tj. kompletu atrybutów trzech wierzchołków) następuje wywołanie procedury `EndPrimitive`. W ten sposób do etapu obcinania zostają przekazane trzy taśmy trójkątowe, z których każda składa się z jednego



Rysunek 10.3. Obrazy trójkątów wytworzonych przez szadery geometrii

trójkąta. Kwalifikator układu wyjścia szadera w linii 6 deklaruje wyprowadzanie taśm trójkątowych, przy czym jest tam również zadeklarowana suma liczb wierzchołków tych taśm. Rysunek 10.3 przedstawia wyniki działania programów zawierających opisane wyżej szadery geometrii, wbudowanych w aplikację 1B.

Szader geometrii może zmienić rodzaj prymitywu, na przykład zamiast trójkąta otrzymanego na wejściu może wyprowadzić odcinki; wiele odcinków, których jeden koniec jest punktem trójkąta, może utworzyć model sierści porastającej ten trójkąt. Przykład mamy na listingu 10.17 (na następnej stronie) i na rysunku 10.4. Pokazany szader geometrii współpracuje z szaderem wierzchołków, który tylko przypisuje zmiennej `gl_Position` otrzymany na wejściu wektor współrzędnych położenia wierzchołka (bez żadnych przekształceń) i z szaderem fragmentów z listingu 10.4. W podwójnej pętli, dokonując interpolacji wierzchołków, szader oblicza punkty wewnątrz trójkąta, Dla każdego z nich szader oblicza drugi punkt, będący końcem „włosa" wyrastającego z pierwszego punktu i wyprowadza tę parę punktów, czyli odcinek. Na potrzeby modelu oświetlenia dla każdego włosa jest obliczany „wektor normalny", który jest wektorem jednostkowym prostopadłym do włosa. Niestety, tak otrzymana sierść nie może być bardzo gęsta (w pokazanym przykładzie udało się zapuścić tylko 36 włosów na każdym trójkącie) z powodu ograniczenia liczby punktów wyprowadzanych przez wątek szadera geometrii[20].



Rysunek 10.4. Dwudziestościan z sierścią

Gdy szader geometrii zamienia otrzymany na wejściu trójkąt na wiele trójkątów albo odcinków, musi ich wierzchołki wyprowadzać po kolei, tzn. sekwencyjnie. Okazję do lepszego zrównoleglenia procesu powiększania danych stwarzają szadery rozdrabniania, dlatego używanie ich zamiast szaderów geometrii jest polecane wszędzie tam, gdzie można ich użyć[21]. Szaderom rozdrabniania przyjrzymy się w aplikacji 1D i we wszystkich wersjach drugiej aplikacji.

[20] A ściślej biorąc, ograniczenia objętości wyprowadzanych danych, zależnej od ilości atrybutów każdego punktu. Można ominąć to ograniczenie; zobacz (a najlepiej zrób) ćwiczenia 8 i 9.

[21] W aplikacjach OpenGL-a 3.0–3.3 szaderów rozdrabniania nie można użyć.

Listing 10.17. Szader geometrii wytwarzający sierść

GLSL

```
#version 420

#define N 8
#define R 1.1

layout(triangles) in;
layout(line_strip,max_vertices=72) out;  /* 72 = N*(N+1) */

out FVertex {
    vec3 Colour, Position, Normal;
  } Out;

uniform TransBlock {
    mat4 mm, vm, pm, vpm;
    vec4 eyepos;
  } trb;

void main ( void )
{
  int   i, j;
  float u, v;
  vec4  p, q;
  vec3  nv;

  for ( i = 1; i <= N; i++ ) {
    u = float(2*i-1)/float(2*N);
    for ( j = 0; j < i; j++ ) {
      v = float (2*j+1)/float(2*i);
      q = mix ( gl_in[1].gl_Position, gl_in[2].gl_Position, v );
      p = trb.mm * mix ( gl_in[0].gl_Position, q, u );
      Out.Position = p.xyz;
      gl_Position = trb.vpm*p;
      Out.Normal = nv = normalize ( cross ( p.xyz, vec3(p.y,-p.z,p.z) ) );
      Out.Colour = vec3(0.1);
      EmitVertex ();
      Out.Position = p.xyz = R * normalize ( p.xyz );
      gl_Position = trb.vpm*p;
      Out.Normal = nv;
      Out.Colour = vec3(0.9);
      EmitVertex ();
      EndPrimitive ();
    }
  }
} /*main*/
```

10.6.5. *Obcinanie i odrzucanie

Płaszczyzny ścian kostki standardowej służą do **obcinania** (*clipping*) prymitywów, tj. wierzchołków, odcinków i trójkątów przed etapem rasteryzacji. Polega ono na odrzuceniu części wspólnej prymitywu i półprzestrzeni po „niewłaściwej" stronie każdej z tych płaszczyzn; pozostaje po tym punkt, odcinek lub wielokąt wypukły (który dalej jest dzielony na trójkąty przed przekazaniem do rasteryzacji). Aplikacja może określić dodatkowe płaszczyzny obcinania.

Równanie płaszczyzny danej przy użyciu jej punktu $\boldsymbol{p}_0 = (x_0, y_0, z_0)$ i wektora normalnego $\boldsymbol{n} = (a, b, c)$ ma postać

$$ax + by + cx + d = 0,$$

a korzystając ze współrzędnych jednorodnych, możemy napisać

$$aX + bY + cZ + dW = 0.$$

Współczynnik d jest równy $-ax_0 - by_0 - cz_0 = \langle \boldsymbol{n}, \boldsymbol{o} - \boldsymbol{p}_0 \rangle$ (symbol $\boldsymbol{o}$ oznacza początek układu współrzędnych).

Wyrażenie po lewej stronie równania dla punktu $\boldsymbol{p} = (x, y, z)$ (ewentualnie reprezentowane przez współrzędne jednorodne (X, Y, Z, W) z wagą $W > 0$) opisuje pomnożoną przez W odległość ze znakiem punktu od płaszczyzny. Jeśli jest ono równe 0, to punkt $\boldsymbol{p}$ leży na płaszczyźnie. Jeśli jest różne od zera, to jego wartość jest proporcjonalna do odległości punktu od tej płaszczyzny. Obowiązuje umowa, że jeśli wyrażenie to jest ujemne, to punkt leży po „niewłaściwej" stronie płaszczyzny, a jeśli dodatnie, to po „właściwej". Mając odcinek, którego końce znajdują się po przeciwnych stronach, OpenGL oblicza punkt wspólny tego odcinka z płaszczyzną; punkt ten rozdziela fragmenty odcinka do odrzucenia i zostawienia.

Aby spowodować obcinanie prymitywów dodatkowymi płaszczyznami, należy:

- zadeklarować w treści *ostatniego szadera części przedniej* potoku przetwarzania grafiki (najlepiej w tablicy, niech ma nazwę `ClipPlane`) zmienne jednolite typu `vec4` i przypisać tym zmiennym wektory współczynników (a, b, c, d) równań poszczególnych płaszczyzn;
- przedeklarować tablicę `gl_ClipDistance` (zobacz p. 9.11), która jest wyjściową zmienną szadera, podając jej długość, równą maksymalnej liczbie potrzebnych płaszczyzn, na przykład

```
out float gl_ClipDistance[2];
```

- przetwarzając wierzchołek, obliczyć dla każdej płaszczyzny wartość odpowiedniego wyrażenia; jeśli na przykład współrzędne jednorodne wierzchołka są dane w zmiennej `in_Position` typu `vec4`, to najprościej jest użyć instrukcji

```
for ( i = 0; i < 2; i++ )
  gl_ClipDistance[i] = dot ( in_Position, ClipPlane[i] );
```

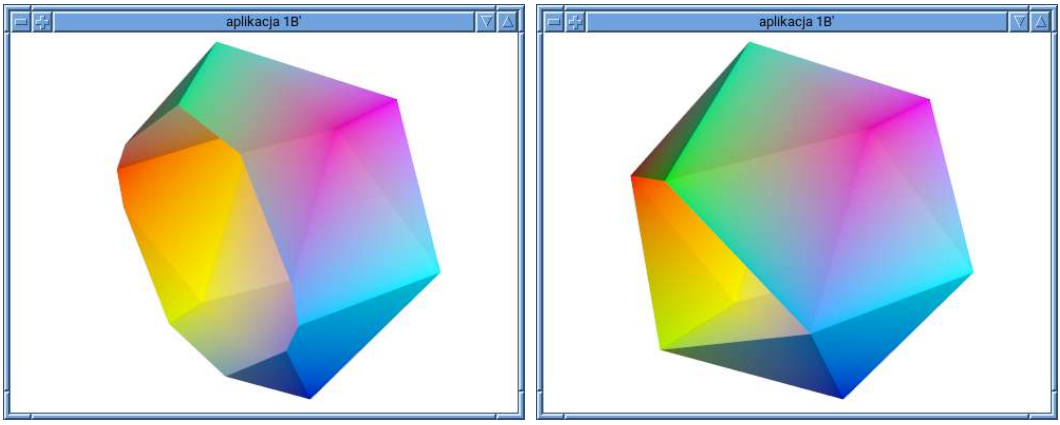
- w aplikacji, przed rysowaniem, włączyć obcinanie poszczególnymi płaszczyznami; dla i-tej płaszczyzny robi to instrukcja

```
glEnable ( GL_CLIP_DISTANCE0 + i );
```

Niepotrzebne płaszczyzny obcinające wyłącza się za pomocą procedury `glDisable`.

Na podobnej zasadzie opiera się (dostępne w OpenGL 4.5 i nowszych) **odrzucanie** prymitywów (*culling*), przy czym różnica polega na tym, że prymityw jest „przepuszczany" w całości, jeśli choć jeden z jego wierzchołków leży po właściwej stronie płaszczyzny (i odrzucany w przeciwnym razie). Nie ma więc (bardziej czasochłonnego) wyznaczania części wspólnych prymitywów z półprzestrzeniami. Odległości wierzchołków od płaszczyzn odrzucających należy podać w tablicy `gl_CullDistance`, którą również trzeba przedeklarować podobnie jak `gl_ClipDistance`. Płaszczyzna odrzucająca staje się aktywna przez sam fakt przypisania wartości odpowiedniemu elementowi tablicy `gl_CullDistance` i nie można jej wyłączyć za pomocą `glDisable`, zatem chcąc wybierać aktywne płaszczyzny odrzucające w czasie działania aplikacji, trzeba umieścić przypisanie w instrukcji warunkowej[22].

Implementacje OpenGL-a nakładają ograniczenia na długości tablic `gl_ClipDistance` i `gl_CullDistance` i na ich łączną długość; aplikacja może je poznać, wywołując procedurę `glGetIntegerv` z parametrem `GL_MAX_CLIP_DISTANCES`, `GL_MAX_CULL_DISTANCES` lub `GL_MAX_COMBINED_CLIP_AND_CULL_DISTANCES` (specyfikacja wymaga, aby mogło być co najmniej 8 płaszczyzn każdego rodzaju i 8 w sumie).



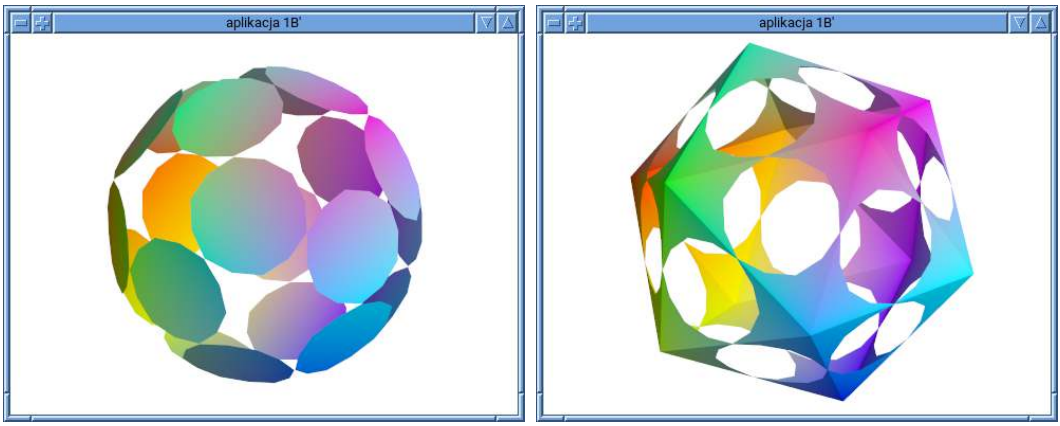
Rysunek 10.5. Obrazy dwudziestościanu ze ścianami obciętymi i bez ścian odrzuconych

Rysunek 10.5 przedstawia okno zmodyfikowanej aplikacji, wyświetlającej obraz bryły poddanej obcinaniu i odrzucaniu, z płaszczyzną $x = -0.4$.

Można zaprząc OpenGL-a do obcinania prymitywów powierzchniami zakrzywionymi; wystarczy obliczać i wyprowadzać w tablicy `gl_ClipDistance` wartości wyrażeń bardziej wymyślnych niż wielomiany pierwszego stopnia. Na przykład wyrażenie $r - \|\boldsymbol{p} - \boldsymbol{c}\|$ opisuje

[22] W tej sprawie specyfikacja [1] nie jest zbyt precyzyjna, a niektóre informacje na temat odrzucania podane w książkach [22], [23] i [24] wypadły słabo w konfrontacji z eksperymentem.

odległość ze znakiem punktu $\boldsymbol{p}$ od sfery o środku $\boldsymbol{c}$ i promieniu r. Wyprowadzanie takich odległości wierzchołków w tablicy `gl_ClipDistance` prowadzi do obcięcia i odrzucenia fragmentów prymitywu wystających poza kulę, której brzegiem jest ta sfera. Ale aby to dało poprawny efekt, obcinane prymitywy powinny mieć znacznie mniejsze wymiary niż promień sfery. Jeśli na przykład spróbujemy obciąć ściany lub krawędzie dwudziestościanu wpisanego w kulę do nieco mniejszej kuli o tym samym środku, to każda ściana i każda krawędź zostanie w całości odrzucona, bo wszystkie wierzchołki dwudziestościanu znajdują się na zewnątrz. Aby części ścian lub krawędzi przeszły etap obcinania, trzeba je podzielić na mniejsze kawałki. Może to zrobić szader geometrii w sposób opisany w podrozdziale E.4. Można też użyć do tego szadera rozdrabniania (zobacz podrozdz. 12.1), co być może jest lepszym rozwiązaniem, ale wymusza rysowanie trójkątów lub odcinków jako płatów.



Rysunek 10.6. Ściany dwudziestościanu obcięte do kuli i do jej dopełnienia

Obrazek z lewej strony na rysunku 10.6 przedstawia wynik obcinania podzielonych na 36 mniejszych trójkątów ścian dwudziestościanu do kuli o promieniu 0.85. Wyprowadzając kolejne wierzchołki, których współrzędne położenia (kartezjańskie, w układzie modelu) zostały wcześniej zapisane w tablicy `pos`, szader geometrii wierzchołka wykonywał instrukcję

```
gl_ClipDistance[0] = 0.85 - length ( pos[k] );
```

Wynikiem obcinania są wielokąty, choć części wspólne ścian dwudziestościanu z tą kulą są kołami. Dokładniejsze (ale zawsze wielokątne) przybliżenia tych kół możemy otrzymać, dzieląc ściany na mniejsze kawałki. Na obrazku z prawej strony jest widoczny skutek zmiany na przeciwny znaku przypisywanego wyrażenia.

10.7. Ćwiczenia

1. Włącz dodatkowe źródła światła, położone w innych miejscach, i poeksperymentuj z ich położeniami i intensywnościami.

2. Oprogramuj interakcyjne zmienianie położeń źródeł światła.
3. Oprogramuj animację źródeł światła — mogą one okrążać obiekt, mogą też zmieniać intensywność i kolor.
4. Napisz szadery geometrii i wierzchołków w taki sposób, aby obliczenie koloru według przyjętego modelu oświetlenia było realizowane przez szader geometrii, co doprowadzi do uzyskania obrazu z cieniowaniem Gourauda. Porównaj obrazy — wyniki cieniowania metodami Gourauda i Phonga.
5. Zmodyfikuj szader fragmentów tak, aby umożliwić otrzymanie mgły na obrazie. Parametry mgły przekaż w odpowiednich zmiennych jednolitych.
6. Zaimplementuj reflektory i wykonaj eksperymenty z nimi.
7. *Zmodyfikuj aplikację 1B tak, aby można było wykonywać obrazy przy użyciu szaderów opisanych w p. 10.6.4. Wprowadź do szadera z listingu 10.15 zmienną jednolitą, której wartość określa przesunięcie (zamiast „na twardo" zakodowanej stałej 0.05) i zaprogramuj taką animację tej zmiennej, aby przesunięte trójkąty „pulsowały" wokół dwudziestościanu.
8. *Obiekty na obrazie można powielić, rysując pewną liczbę ich **instancji**, co jest opisane w rozdziale 15. Napisz i wypróbuj szader geometrii, który na każdej instancji trójkąta wytworzy inne włosy, dzięki czemu powstanie sierść gęstsza niż na rysunku 10.4.
9. *Do zagęszczenia sierści można zaprząc też inny wspomniany w tym rozdziale sposób powielenia obiektów, który polega na spowodowaniu wykonania wielu wątków szadera geometrii dla każdego rysowanego trójkąta, czego skutkiem będzie przekazanie do dalszych etapów potoku danych wyprowadzonych przez każdy z tych wątków. W tym celu trzeba w treści szadera podać kwalifikator

   ```
   layout(triangles,invocations=n) in;
   ```

   z odpowiednio dobraną stałą n. Każdy wątek szadera otrzyma w zmiennej wbudowanej `gl_InvocationID` swój numer (od 0 do $n-1$) i na jego podstawie może wyprowadzić przydzielone sobie włosy.
10. *Włosy mogą być powyginane, przy czym kształt gładkiej krzywej trzeba przybliżyć łamaną złożoną z dostatecznie krótkich odcinków. W podrozdz. 15.1 jest opisana reprezentacja Béziera krzywych, wygodna w tym zastosowaniu (krzywe mogą mieć stopień 2 lub 3). Napisz i uruchom szader geometrii, który dla każdej instancji trójkąta wyprowadzi jeden włos, tj. jedną łamaną, „wyrastającą" z odpowiedniego punktu trójkąta i wygiętą na przykład „pod własnym ciężarem".

11

Aplikacja pierwsza C

Do aplikacji dodamy pisanie tekstu; niech obracanie obserwatora wokół obiektu powoduje wyświetlenie na tle obrazu współrzędnych położenia obserwatora (czyli początku układu obserwatora) w układzie świata. Wyświetlanie tekstu jest o tyle trudne, że w większości krojów pisma poszczególne znaki mają różne szerokości, a ponadto zajmowane przez nie pola mogą (wskutek kerningu) nachodzić na siebie. Tu ograniczymy się do wyświetlania napisów złożonych ze znaków o ustalonej szerokości, zostawiając składanie bardziej skomplikowanego tekstu jako temat przyszłych badań.

11.1. Reprezentacja fontów

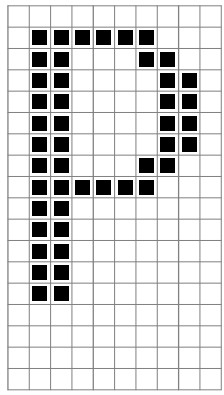
W kroju pisma (foncie) o stałej szerokości każdy znak zajmuje prostokąt o tych samych wymiarach, powiedzmy $h \times w$ pikseli; liczby w i h są naturalne. Załóżymy, że $1 \leqslant w \leqslant 32$. Każdy wiersz rastra w polu znaku chcemy zakodować w jednym, dwóch lub czterech bajtach, to znaczy w jednej liczbie typu `GLubyte`, `GLushort` albo `GLuint`, przy czym wybieramy typ zajmujący najmniejszą wystarczającą ilość miejsca. Dla każdego znaku podamy h liczb tego typu, zaczynając od wiersza na samej górze.

Przykład jest pokazany na rysunku 11.1: piksele w każdym wierszu zamieniamy na zera i jedynki, których kolejność w wierszu jest odwrócona[1]. Otrzymane zera i jedynki możemy zgrupować po 4 i każdą taką czwórkę zamienić na jedną cyfrę szesnastkową. Dla każdego znaku zajmującego pole o wymiarach 18×10 podajemy 18 liczb typu `GLushort`.

Chcemy, aby piksele wzorców znaków odpowiadały pikselom w oknie. W tym celu skonstruujemy macierz rzutowania zgodnie z przepisem w podrozdziale 6.5.

Przed wyświetlaniem tekstu należy umieścić w pamięci GPU (w odpowiednich zmiennych jednolitych) font (czyli zestaw wzorców znaków z potrzebnymi informacjami dodatkowymi), kolory znaków i tła oraz sam napis, tj. ciąg kodów ASCII kolejnych znaków.

[1]Powód odwrócenia jest taki, że wygodniejsza jest reprezentacja fontu, w której najmniej znaczący bit odpowiada pikselowi z lewej strony, a zapis pozycyjny umieszcza najmniej znaczące cyfry z prawej.



```
0 0 0 0 0 0 0 0 0 0  0x000
0 0 0 1 1 1 1 1 1 0  0x07E
0 0 1 1 0 0 0 0 1 1 0  0x0C6
0 1 1 0 0 0 0 0 1 1 0  0x186
0 1 1 0 0 0 0 0 1 1 0  0x186
0 1 1 0 0 0 0 0 1 1 0  0x186
0 1 1 0 0 0 0 0 1 1 0  0x186
0 0 1 1 0 0 0 0 1 1 0  0x0C6
0 0 0 1 1 1 1 1 1 1 0  0x07E
0 0 0 0 0 0 0 0 1 1 0  0x006
0 0 0 0 0 0 0 0 1 1 0  0x006
0 0 0 0 0 0 0 0 1 1 0  0x006
0 0 0 0 0 0 0 0 1 1 0  0x006
0 0 0 0 0 0 0 0 1 1 0  0x006
0 0 0 0 0 0 0 0 0 0 0  0x000
0 0 0 0 0 0 0 0 0 0 0  0x000
0 0 0 0 0 0 0 0 0 0 0  0x000
0 0 0 0 0 0 0 0 0 0 0  0x000
```

Rysunek 11.1. Reprezentacja jednej litery

Algorytm wyświetlania tekstu w zarysie wygląda tak: długość (tj. liczbę znaków) napisu pomnożymy przez w, tj. szerokość znaku. Należy narysować prostokąt, zwany dalej **obszarem tekstu**, o otrzymanej w ten sposób szerokości i o wysokości znaku h. *Dolny lewy* wierzchołek obszaru tekstu ma współrzędne ξ', η' podane w układzie okna[2]. Zadaniem szadera fragmentów jest określenie, czy przetwarzany piksel obszaru tekstu ma otrzymać kolor znaku albo kolor tła, czy też ma zachować niezmienioną wartość.

Możemy chcieć wyświetlić znaki w ustalonym kolorze na tle w innym ustalonym kolorze, czyli przypisać jeden albo drugi kolor *każdemu* pikselowi obszaru tekstu. Inna możliwość to przypisanie koloru tylko tym pikselom, które należą do obszaru znaku (a więc tło miałoby zostać niezmienione), albo tylko tym pikselom, które należą do tła. Aby wybrać jedną z tych możliwości, użyjemy **składowej alfa** (tj. współrzędnej a wektora (r, g, b, a)) koloru znaków lub tła. Jeśli wartość tej składowej jest zerem, to odpowiedni piksel zostawimy niezmieniony, a w przeciwnym razie przypiszemy mu odpowiedni kolor.

11.2. Szadery

Program szaderów zbudujemy z szadera wierzchołków podanego na listingu 11.1 i z szadera fragmentów na listingu 11.2. Trzy bloki zmiennych jednolitych, `MyGC`, `MyFont` i `MyText` opisują odpowiednio minikontekst graficzny, reprezentację fontu i reprezentację tekstu.

Zadaniem szadera wierzchołków jest obliczenie współrzędnych wierzchołków obszaru tekstu w kostce standardowej. Blok zmiennych jednolitych `MyFont` w polach `chw` i `chh` zawiera szerokość i wysokość każdego znaku. Blok `MyText` zawiera tekst; w polach `x`, `y` i `l` są podane współrzędne ξ', η' położenia tekstu na ekranie oraz liczba znaków napisu.

Dane te umożliwiają skonstruowanie wszystkich wierzchołków obszaru tekstu, co następuje w liniach 24–30. Zmienna wbudowana `gl_VertexID` jest numerem przetwarzanego przez wątek szadera wierzchołka prymitywu, w tym przypadku wachlarza trójkątów, którego

[2] Czyli położenie napisu podajemy w dosyć tradycyjny sposób. Rzutowanie zapewnia, że współrzędne wierzchołków obszaru tekstu są współrzędnymi w układzie okna.

dwa trójkąty dają w sumie prostokąt — obszar tekstu[3]. Współrzędne wierzchołka są przypisywane zmiennej p, po czym w linii 31 następuje przejście do układu współrzędnych kostki standardowej.

Pole pm w bloku MyGC zawiera macierz skonstruowaną według przepisu podanego w podrozdziale 6.5; współrzędne punktu przekształconego za pomocą tej macierzy do układu kostki standardowej zostaną odtworzone w etapie rasteryzacji. Dzięki temu punkt (wierzchołek obszaru tekstu) wygenerowany przez szader wierzchołków ma w oknie współrzędne obliczone w linii 25, 26, 28 lub 29.

Listing 11.1. Szader wierzchołków do pisania tekstu

GLSL

```
1: #version 420
2:
3: uniform MyGC {
4:     mat4 pm;      /* macierz rzutowania */
5:     vec4 fg, bk;  /* kolory znakow i tła */
6:   } gc;
7:
8: uniform MyFont {
9:     int  chw, chh;   /* szerokość i wysokość znaku */
10:    int  chf, chl;   /* kody pierwszego i ostatniego znaku */
11:    uint glyphs[288];
12:  } font;
13:
14: uniform MyText {
15:    int  x, y;      /* pozycja pierwszego znaku */
16:    int  l;         /* długość napisu */
17:    uint text[64];  /* upakowane znaki napisu */
18:  } text;
19:
20: void main ( void )
21: {
22:   vec4 p;
23:
24:   switch ( gl_VertexID ) {
25:  case 0: p = vec4(text.x, text.y-font.chh, 0.0, 1.0);  break;
26:  case 1: p = vec4(text.x+text.l*font.chw, text.y-font.chh, 0.0, 1.0);
27:          break;
28:  case 2: p = vec4(text.x+text.l*font.chw, text.y, 0.0, 1.0);  break;
29: default: p = vec4(text.x, text.y, 0.0, 1.0);  break;
30:   }
31:   gl_Position = gc.pm * p;
32: } /*main*/
```

[3] Szader nie otrzymuje od etapu pobierania wierzchołków żadnych atrybutów wierzchołka, z wyjątkiem numeru podanego w zmiennej gl_VertexID.

Wbudowana zmienna wejściowa `gl_FragCoord` szadera fragmentów (listing 11.2) jest zmienną typu `vec4`, która może być używana bez deklaracji. Jej redeklaracja w linii 3 ma na celu zmianę układu, w którym współrzędne fragmentu są podawane — początek układu, którego tu chcemy używać, ma się znajdować w lewym górnym narożniku okna.

Oprócz szerokości i wysokości znaku blok zmiennych jednolitych `MyFont` zawiera (w polach `chf` i `chl`) kody pierwszego i ostatniego znaku, którego wzorzec bitowy jest obecny w foncie, oraz tablicę `glyphs` z wzorcami wszystkich znaków. Taka tablica musi mieć zadeklarowaną długość, przy czym faktyczna długość tablicy *nie musi* odpowiadać deklaracji. Ważne jest to, aby szader, odczytując dane podczas obliczeń, nie sięgał poza obszar pamięci zarezerwowany na bufor zawierający używany blok zmiennych jednolitych. Liczba 288, podana jako długość tablicy `glyphs` (w linii 11 na listingu 11.1) odpowiada 1152 bajtom; można w nich zmieścić reprezentacje 96 znaków, z których każda zajmuje 12 bajtów.

Nie ma w GLSL-u „krótkich" typów liczb całkowitych (takich jak `char` lub `short` w C), dlatego wzorce bitowe znaków są upakowane w liczbach czterobajtowych (typu `uint`). Jeśli szerokość znaku nie jest większa niż 8 pikseli, to poszczególne bajty odpowiadają kolejnym wierszom wzorca. Jeśli szerokość znaku jest większa, ale nie przekracza 16 pikseli, to szesnastobitowe „połówki" liczby reprezentują dwa wiersze wzorca, a dla znaków jeszcze szerszych (do 32 pikseli) jedna liczba to jeden wiersz wzorca. Powoduje to konieczność „wydobycia" podczas obliczeń potrzebnego bajtu lub połówki z liczby czterobajtowej.

Dobrze napisane makrodefinicje mogą znacznie skrócić i uczytelnić kod źródłowy. Szader na listingu 11.2 wykorzystuje dwie. Makro `EXTRACTBYTE` służy do „wycinania" bajtu ze zmiennej typu `uint` podanej jako pierwszy parametr tego makra. Wskazany bajt, jeden z czterech, zostaje przesunięty na najmniej znaczącą pozycję.

Kod generowany przez makro `SETFRAG`, zależnie od składowej alfa koloru podanego jako parametr (koloru znaków albo tła), przypisuje ten kolor zmiennej wyjściowej `out_Colour` i kończy (za pomocą instrukcji `return`) działanie szadera albo kończy działanie szadera za pomocą instrukcji `discard`, co powoduje odrzucenie fragmentu (czyli pozostawienie niezmienionego koloru piksela).

Listing 11.2. Szader fragmentów do pisania tekstu

GLSL

```
1: #version 420
2:
3: layout(origin_upper_left) in vec4 gl_FragCoord;
4: out vec4 out_Colour;
5:
6: uniform MyGC { .... } gc;      /* zobacz listing 11.1 */
7: uniform MyFont { .... } font;
8: uniform MyText { .... } text;
9:
10: #define EXTRACTBYTE(x,b) \
11:   switch ( b ) { \
12: default: break; \
13: case  1: x >>=  8;  break; \
```

```
14: case  2: x >>= 16;  break; \
15: case  3: x >>= 24;  break; \
16:   }
17:
18: #define SETFRAG(C) \
19:  { if ( gc.C.a > 0.0 ) { \
20:      out_Colour = gc.C; \
21:      return; \
22:    } \
23:    else discard; }
24:
25: void main ( void )
26: {
27:   int  x0, y0;
28:   uint c, r, mask, chrow;
29:
30:   x0 = int(gl_FragCoord.x) - text.x;
31:   if ( x0 < 0 || x0 >= font.chw*text.l )
32:     SETFRAG ( bk )
33:   y0 = int(gl_FragCoord.y) - (text.y-font.chh) - 1;
34:   if ( y0 < 0 || y0 >= font.chh )
35:     SETFRAG ( bk )
36:   c = text.text[x0/(font.chw*4)];
37:   EXTRACTBYTE ( c, (x0/font.chw) % 4 )
38:   c &= 0xFF;
39:   if ( c < font.chf || c > font.chl )
40:     SETFRAG ( bk )
41:   c -= font.chf;
42:   r = c*font.chh + y0;
43:   if ( font.chw <= 8 ) {
44:     chrow = font.glyphs[r/4];
45:     EXTRACTBYTE ( chrow, r % 4 )
46:   }
47:   else if ( font.chw <= 16 ) {
48:     chrow = font.glyphs[r/2];
49:     if ( r % 2 != 0 )
50:       chrow >>= 16;
51:   }
52:   else
53:     chrow = font.glyphs[r];
54:   x0 %= font.chw;
55:   mask = 0x01 << x0;
56:   if ( (chrow & mask) != 0 )
57:     SETFRAG ( fg )
58:   else
59:     SETFRAG ( bk )
60: } /*main*/
```

Blok zmiennych jednolitych nazwany `MyText` (a lokalnie `text`) zawiera informacje związane z konkretnym napisem do wyświetlenia: są tam współrzędne `x`, `y` (czyli ξ', η') położenia początku tekstu, długość (tj. liczba znaków) napisu `l` i tablica liczb typu `uint`; w elementach tej tablicy kody ASCII kolejnych znaków napisu są upakowane po 4.

Zgodnie z deklaracją zmiennej `gl_FragCoord` w linii 3, wartości pól `x` i `y` tej zmiennej są współrzędnymi ξ', η' piksela w oknie (w układzie o początku w górnym lewym narożniku okna). W linii 30 szader oblicza (i przypisuje zmiennej `x0`) przesunięcie piksela względem lewego brzegu obszaru tekstu. Jeśli piksel jest poza tym obszarem[4], to zależnie od składowej alfa (badamy to w linii 19) nadajemy pikselowi (przez przypisanie do zmiennej `out_Colour`) kolor tła albo odrzucamy fragment.

W liniach 33–35 obliczamy numer wiersza we wzorcu znaku, odpowiadający pikselowi przetwarzanemu przez szader. Jeśli ten numer jest mniejszy niż 0, to piksel leży powyżej obszaru tekstu, a jeśli jest większy lub równy wysokości znaku, to poniżej. W obu tych przypadkach, zależnie od składowej alfa koloru tła, przypisujemy zmiennej `out_Colour` kolor tła albo odrzucamy fragment.

W liniach 36–38 z tablicy, w której jest podany napis, jest wyciągany kod ASCII znaku, w obszarze którego znajduje się przetwarzany piksel[5]; jest do tego użyte makro `EXTRACTBYTE`, które w zmiennej `c`, zawierającej początkowo kod potrzebnego znaku i kody trzech znaków sąsiednich, przesuwa potrzebny bajt na najmniej znaczącą pozycję. W linii 38 pozostałe bajty są kasowane. Jeśli otrzymana w ten sposób liczba jest poza zakresem kodów obecnych w foncie znaków, to w linii 40 nadajemy pikselowi kolor tła.

Zmniejszenie wartości zmiennej `c` w linii 41 o kod pierwszego reprezentowanego znaku (`font.chf`) jest początkiem obliczenia indeksu potrzebnego wiersza wzorca znaku. Jeśli wiersze wzorców wszystkich znaków ustawimy kolejno jeden pod drugim, to wartość nadana zmiennej `r` w linii 42 jest numerem wiersza w takim wzorcu całego fontu. Instrukcje warunkowe w liniach 43 i 47 wybierają właściwe postępowanie dla przypadków, gdy znaki są wąskie (co najwyżej 8 pikseli), średniej szerokości (od 9 do 16) lub szerokie (do 32 pikseli). W linii 44 z tablicy `font.glyphs` wybierany jest element zawierający odpowiedni wiersz, a następnie makro `EXTRACTBYTE` wybiera odpowiedni bajt z tego elementu. Analogicznie działa wybieranie potrzebnej połówki liczby 32-bitowej dla znaków o średniej szerokości zaprogramowane w liniach 48–50.

W linii 54 zmienna `x0` otrzymuje wartość będącą numerem kolumny we wzorcu znaku. Wartość nadana zmiennej `mask` w linii 55 jest maską bitową, w której jest jedynka na pozycji tej kolumny i zera wszędzie indziej. Warunek w linii 56 jest spełniony, gdy w wierszu wzorca znaku też jest jedynka na tej pozycji i wtedy należy nadać pikselowi kolor znaku (albo od-

[4] To mogłoby się zdarzyć, gdybyśmy chcieli narysować figurę większą niż obszar tekstu, przy użyciu zmodyfikowanego szadera wierzchołków.

[5] Zauważmy, że dla kroju pisma o stałej szerokości ustalenie, o który znak napisu chodzi, jest możliwe przy użyciu jednego dzielenia. Pismo tzw. proporcjonalne, ze znakami o różnych szerokościach, wymagałoby albo wyświetlania znaków napisu po kolei (zazwyczaj tak się robi), albo wcześniejszego utworzenia dodatkowej tablicy, w której dla każdego znaku napisu należałoby podać odległość jego początku od początku napisu, tj. od lewej krawędzi obszaru napisu. Szader fragmentów musiałby wyszukać w tej dodatkowej tablicy numer znaku w napisie, na przykład metodą bisekcji. A jeszcze należałoby obsłużyć kerning. Brrr …

rzucić fragment, gdy składowa alfa tego koloru jest zerem). Gdy warunek jest niespełniony, wówczas makro SETFRAG w linii 59 nadaje pikselowi kolor tła albo fragment jest odrzucany.

Zwróćmy uwagę na postać warunku w linii 56. W GLSL-u wyrażenie w warunku musi być boolowskie. Tu jego wartość jest ustalana przez zbadanie relacji między liczbami. W języku C można by napisać po prostu if (chrow & mask) ..., ponieważ wyrażenie liczbowe jest uznawane za opis warunku, który jest spełniony wtedy, gdy wartość wyrażenia nie jest zerem, ale GLSL na to nie pozwala.

11.3. Fonty i procedury wyświetlania tekstu

Procedury związane z wyświetlaniem tekstu umieściłem w pliku mygltext.c, a dwa przygotowane fonty są w plikach font12x6.c i font18x10.c. Na listingu 11.3 jest pokazany fragment pliku nagłówkowego mygltext.h zawierający definicje struktur danych reprezentujących font i tekst do wyświetlenia. Sposób ich używania prześledzimy dalej.

Listing 11.3. Definicje struktur myFontObject i myTextObject

C

```
1:  typedef struct {
2:      GLint  chw, chh;     /* szerokość i wysokość znaku */
3:      GLuint ubo;          /* bufor bloku MyFont */
4:    } myFontObject;
5:
6:  typedef struct {
7:      GLuint buf;          /* bufor bloku MyText */
8:      int    maxlength;    /* maksymalna długość napisu */
9:      myFontObject *font;  /* font dla tego napisu */
10:   } myTextObject;
```

Listing 11.4 pokazuje fragmenty pliku z definicją fontu, którego znaki mają 18 pikseli wysokości i 10 pikseli szerokości. Tak samo jest „zrobiony" font o wymiarach 12 × 6 i podobnie *można* przygotować dalsze fonty.

Listing 11.4. Reprezentacja przykładowego fontu

C

```
1:  #include "openglheader.h"
2:  #include "mygltext.h"
3:
4:  /* ten font został otrzymany przez konwersję fontu z pliku */
5:  /* 10x20-ISO8859-1-pcf.gz z systemu X Window */
6:  static GLushort glyphs18x10[] =
7:   {0x000,0x000,0x000,0x000,0x000,0x000,0x000,0x000,0x000,0x000,
8:    0x000,0x000,0x000,0x000,0x000,0x000,0x000,0x000,0x000,0x000,  /* ' ' */
9:    ....
10:   0x000,0x07e,0x0c6,0x186,0x186,0x186,0x186,0x0c6,0x07e,0x000,
11:   0x006,0x006,0x006,0x006,0x006,0x000,0x000,0x000,0x000,0x000,  /* 'P' */
```

```
  ....
  0x000,0x1ce,0x102,0x102,0x000,0x000,0x102,0x102,0x102,
  0x000,0x000,0x102,0x102,0x1ce,0x000,0x000,0x000,0x000}; /* DEL */

myFontObject *NewFont18x10 ( void )
{
  return NewFontObject ( 10, 18, 0x20, 0x7F,
                         sizeof(glyphs18x10), glyphs18x10 );
} /*NewFont18x10*/
```

W tablicy `glyphs18x10` jest podany ciąg liczb 16-bitowych z wzorcami znaków; znaków tych jest 96 (od spacji o kodzie 32 do znaku `DEL` o kodzie 127), zatem jest w niej $96 \cdot 18 = 1728$ liczb, czyli 3456 bajtów. Wykropkowałem na listingu większość liczb, których i tak nikt by nie przeczytał, zostawiając tylko wzorce trzech znaków, w tym litery z rysunku 11.1. Procedura `NewFontObject`, która rezerwuje w pamięci RAM CPU strukturę typu `myFontObject`, tworzy UBO, umieszcza w nim dane i przekazuje wskaźnik do tej struktury, jest pokazana na listingu 11.6.

Listing 11.5 przedstawia zmienne używane podczas wyświetlania tekstu i procedurę, która kompiluje szadery, a następnie łączy je w program szaderów i przygotowuje go do pracy.

Listing 11.5. Procedura `LoadTextShaders`

C

```
static GLuint text_program_id = 0;
static GLuint gcbp, fontbp, textbp;
static GLint  gcofs[3], fontofs[5], textofs[4];
static GLuint gcbuf;

char LoadTextShaders ( void )
{
  static const char *filename[] =
    { "font.vert.glsl", "font.frag.glsl" };
  static const GLchar *UGCNames[] =
    { "MyGC", "MyGC.pm", "MyGC.fg", "MyGC.bk" };
  static const GLchar *UFontNames[] =
    { "MyFont", "MyFont.chw", "MyFont.chh",
      "MyFont.chf", "MyFont.chl", "MyFont.glyphs" };
  static const GLchar *UTextNames[] =
    { "MyText", "MyText.x", "MyText.y", "MyText.l", "MyText.text" };
  GLuint shader_id[2];
  GLint  gcbsize, fontsize, textsize;
  int    i;

  shader_id[0] = CompileShaderFiles ( GL_VERTEX_SHADER, 1, &filename[0] );
  shader_id[1] = CompileShaderFiles ( GL_FRAGMENT_SHADER, 1, &filename[1] );
  text_program_id = LinkShaderProgram ( 2, shader_id, "text" );
  GetAccessToUniformBlock ( text_program_id, 3, UGCNames,
                            &gcbsize, gcofs, &gcbp );
```

```
26:   GetAccessToUniformBlock ( text_program_id, 5, UFontNames,
27:                             &fontsize, fontofs, &fontbp );
28:   GetAccessToUniformBlock ( text_program_id, 4, UTextNames,
29:                             &textsize, textofs, &textbp );
30:   glGenBuffers ( 1, &gcbuf );
31:   glBindBufferBase ( GL_UNIFORM_BUFFER, gcbp, gcbuf );
32:   glBufferData ( GL_UNIFORM_BUFFER, gcbsize, NULL, GL_DYNAMIC_DRAW );
33:   ConstructEmptyVAO ();
34:   for ( i = 0; i < 2; i++ )
35:     glDeleteShader ( shader_id[i] );
36:   ExitIfGLError ( "LoadTextShaders" );
37:   return 1;
38: } /*LoadTextShaders*/
```

Zmienna `text_program_id` przechowuje identyfikator programu, w zmiennych `gcbp`, `fontbp` i `textbp` są pamiętane numery punktów dowiązania bloków zmiennych jednolitych `MyGC`, `MyFont` i `MyText` w celu `GL_UNIFORM_BUFFER`, a w tablicach `gcofs`, `fontofs` i `textofs` są przechowywane położenia poszczególnych pól w tych blokach. Zmienna `gcbuf` jest identyfikatorem bufora, który zawiera blok `MyGC`. Wszystkim tym zmiennym wartości nadaje procedura `LoadTextShaders`, która po złączeniu programu po trzykroć wywołuje `GetAccessToUniformBlock` (listing 10.5), a następnie (w liniach 30–32) rezerwuje bufor na blok `MyGC` i przywiązuje go do odpowiedniego punktu dowiązania.

Opisana dalej procedura `ConstructEmptyVAO`, umieszczona w pliku `utilities.c` i wywołana w linii 33, tworzy obiekt tablicy wierzchołków (VAO), który nie będzie zawierał opisu żadnych atrybutów wierzchołków, ale będzie potrzebny podczas wyświetlania tekstu. Identyfikator tego obiektu zostanie zapamiętany w zmiennej globalnej `empty_vao`.

Zadaniem procedury `NewFontObject` (listing 11.6) jest utworzenie UBO zawierającego blok `MyFont` zdefiniowany w liniach 8–12 na listingu 11.1 i nadanie odpowiednich wartości polom zmiennej typu `myFontObject` zarezerwowanej w linii 8. W liniach 10–11 jest tworzony i przywiązywany do celu `GL_UNIFORM_BUFFER` bufor, który będzie wykorzystywany jako UBO. Wcześniej wywołana procedura `LoadTextShaders` umieściła w globalnej tablicy `fontofs` przesunięcia poszczególnych pól struktury `MyFont` używanej przez szader: kolejno są to przesunięcia pól `chw`, `chh`, `chf`, `chl` i `glyphs`. Pola te są umieszczone w bloku jedno za drugim, zatem tablica `glyphs` jest ostatnim polem struktury. Wielkość bufora, określona przez wywołanie procedury `glBufferData` w liniach 12–13, jest sumą przesunięcia pola `glyphs` i podanej jako parametr długości tej tablicy w bajtach, zaokrąglonej w górę (przez makro `ROUNDUP4`) do wartości podzielnej przez 4 — tablica zawiera liczby czterobajtowe, ostatnia liczba też musi w buforze mieścić się cała.

Wywołania procedury `glBufferSubData` w liniach 14–18 przypisują polom w buforze potrzebne dane; ponadto wymiary znaku są zapamiętywane w utworzonej zmiennej typu `myFontObject`, której adres jest podawany jako wartość procedury.

Procedura `DeleteFontObject`, którą aplikacja powinna wywołać podczas sprzątania po sobie, po prostu zwalnia bufor utworzony przez `NewFontObject` i zwalnia pamięć CPU zajmowaną przez font.

Listing 11.6. Procedury `NewFontObject` i `DeleteFontObject`

C

```
#define ROUNDUP4(X) (4*(((X)+3)/4))

myFontObject *NewFontObject ( GLint chw, GLint chh, GLint chf, GLint chl,
                        int size, GLvoid *glyphs )
{
  myFontObject *font;

  if ( (font = malloc ( sizeof(myFontObject) )) ) {
    font->chw = chw;  font->chh = chh;
    glGenBuffers ( 1, &font->ubo );
    glBindBuffer ( GL_UNIFORM_BUFFER, font->ubo );
    glBufferData ( GL_UNIFORM_BUFFER, fontofs[4]+ROUNDUP4(size), NULL,
                   GL_STATIC_DRAW );
    glBufferSubData ( GL_UNIFORM_BUFFER, fontofs[0], sizeof(GLint), &chw );
    glBufferSubData ( GL_UNIFORM_BUFFER, fontofs[1], sizeof(GLint), &chh );
    glBufferSubData ( GL_UNIFORM_BUFFER, fontofs[2], sizeof(GLint), &chf );
    glBufferSubData ( GL_UNIFORM_BUFFER, fontofs[3], sizeof(GLint), &chl );
    glBufferSubData ( GL_UNIFORM_BUFFER, fontofs[4], size, glyphs );
    ExitIfGLError ( "NewFontObject" );
  }
  return font;
} /*NewFontObject*/

void DeleteFontObject ( myFontObject *font )
{
  glDeleteBuffers ( 1, &font->ubo );
  free ( font );
  ExitIfGLError ( "DeleteFontObject" );
} /*DeleteFontObject*/
```

Listing 11.7 przedstawia cztery procedury związane z obiektami reprezentującymi teksty do wyświetlenia. Procedura `NewTextObject` rezerwuje w pamięci RAM CPU zmienną typu `myTextObject`, a w pamięci GPU bufor na reprezentację tekstu w bloku zmiennych jednolitych `MyText`. W liniach 10–11 ustalana jest wielkość tego bufora; jest ona sumą przesunięcia tablicy `text` względem początku bloku `MyText` (zobacz listing 11.1) i maksymalnej, zadeklarowanej za pomocą parametru `maxlength` długości napisu, zaokrąglonej w górę do wartości podzielnej przez 4.[6]

Procedura `SetTextObjectContents` umieszcza w obiekcie tekstowym konkretny tekst. Ciąg kodów ASCII podany w tablicy `text` powinien reprezentować *jedną linię* tekstu; znaki specjalne, takie jak znak końca linii, są przez opisany tu program ignorowane, a dokładniej, zamieniane na spacje. Dlatego teksty złożone z kilku linii muszą być reprezentowane przez kilka obiektów tekstowych — każda linia przez osobny obiekt. Parametry `x` i `y` określają

[6] Pamiętamy, że elementy tablicy `text` też są liczbami czterobajtowymi.

Listing 11.7. Procedury przetwarzania obiektów tekstowych

C

```
myTextObject *NewTextObject ( int maxlength )
{
  myTextObject *to;

  if ( (to = malloc ( sizeof(myTextObject) )) ) {
    memset ( to, 0, sizeof(myTextObject) );
    to->maxlength = ROUNDUP4(maxlength);
    glGenBuffers ( 1, &to->buf );
    glBindBuffer ( GL_UNIFORM_BUFFER, to->buf );
    glBufferData ( GL_UNIFORM_BUFFER, textofs[3]+maxlength, NULL,
                   GL_DYNAMIC_DRAW );
    ExitIfGLError ( "NewTextObject" );
  }
  return to;
} /*NewTextObject*/

void SetTextObjectContents ( myTextObject *to, GLchar *text,
                             GLint x, GLint y, myFontObject *font )
{
  int lgt;

  lgt = strlen ( text );
  if ( lgt > to->maxlength )
    lgt = to->maxlength;
  glBindBuffer ( GL_UNIFORM_BUFFER, to->buf );
  glBufferSubData ( GL_UNIFORM_BUFFER, textofs[0], sizeof(GLint), &x );
  glBufferSubData ( GL_UNIFORM_BUFFER, textofs[1], sizeof(GLint), &y );
  glBufferSubData ( GL_UNIFORM_BUFFER, textofs[2], sizeof(GLint), &lgt );
  glBufferSubData ( GL_UNIFORM_BUFFER, textofs[3], lgt*sizeof(GLchar),
                    text );
  to->font = font;
  ExitIfGLError ( "SetTextObjectContents" );
} /*SetTextObjectContents*/

void DisplayTextObject ( myTextObject *to )
{
  glDisable ( GL_CULL_FACE );
  glDisable ( GL_DEPTH_TEST );
  glUseProgram ( text_program_id );
  glBindBufferBase ( GL_UNIFORM_BUFFER, fontbp, to->font->ubo );
  glBindBufferBase ( GL_UNIFORM_BUFFER, textbp, to->buf );
  glBindVertexArray ( empty_vao );
  glDrawArrays ( GL_TRIANGLE_FAN, 0, 4 );
  glBindVertexArray ( 0 );
  ExitIfGLError ( "DisplayTextObject" );
```

```
46: } /*DisplayTextObject*/
47:
48: void DeleteTextObject ( myTextObject *to )
49: {
50:   glDeleteBuffers ( 1, &to->buf );
51:   free ( to );
52:   ExitIfGLError ( "DeleteTextObject" );
53: } /*DeleteTextObject*/
```

współrzędne dolnego lewego piksela obszaru tekstu w oknie[7]. Parametr `font` jest adresem struktury `myFontObject` utworzonej przez procedurę `NewFontObject`; adres ten zostaje zapamiętany w obiekcie tekstowym (tj. w strukturze wskazywanej przez parametr `to`), ale przedtem procedura oblicza długość napisu i ewentualnie zmniejsza ją do limitu podanego podczas tworzenia obiektu, po czym przesyła do bufora współrzędne położenia początku napisu i jego długość.

Procedura `DisplayTextObject` wyświetla tekst. Przed wywołaniem tej procedury trzeba zadbać o właściwą zawartość bloku zmiennych jednolitych `MyGC`, co procedura `SetupTextFrame`, opisana dalej, powinna była zrobić wcześniej.

W liniach 37–38 procedura wyłącza odrzucanie ścian odwróconych tyłem i test widoczności, które nijak nie pasują do wyświetlania tekstu. Następnie jest uaktywniany program utworzony przez `LoadTextShaders`. W liniach 40–41 jako bloki zmiennych jednolitych `MyFont` i `MyText` są podłączane UBO zawierające odpowiednio font, którym tekst ma być wyświetlony oraz sam tekst. W linii 42 bieżącym VAO staje się pusty obiekt tablicy wierzchołków utworzony przez procedurę `LoadTextShaders`, po czym procedura `glDrawArrays` w linii 43 rysuje prostokąt — obszar tekstu. Resztę, w tym kaligrafię, załatwiają szadery. Po narysowaniu obszaru tekstu odczepiamy pusty bufor tablicy wierzchołków, wywołując procedurę `glBindVertexArray` z parametrem `0`.

Procedura `DeleteTextObject` likwiduje bufor tekstu i zwalnia pamięć CPU zajmowaną przez strukturę `myTextObject`.

Procedura `SetupTextFrame` na listingu 11.8 powinna być wywołana za każdym razem, gdy okno otrzymało nowe wymiary — czyli jej wywołanie trzeba dodać do procedury `ResizeMyWorld`. Procedura ta konstruuje macierz przekształcenia bryły widzenia na kostkę standardową zgodnie z przepisem w podrozdziale 6.5: współrzędne x, y wierzchołka podane w VAO mają być równe współrzędnym ξ', η' obrazu wierzchołka w oknie. Współczynniki tej macierzy są w linii 8 przesyłane do UBO przywiązanego do bloku zmiennych jednolitych `MyGC` w programie szaderów.

Procedury `SetTextForeground` i `SetTextBackground` przesyłają kolory znaków i tła do odpowiednich pól w bloku `MyGC`. Procedura sprzątająca `TextCleanup` likwiduje program szaderów i zwalnia UBO przywiązany do bloku `MyGC`.

[7] Wyświetlając kilka linii tekstu, należy je odpowiednio rozmieścić względem siebie. W szczególności trzeba zadbać o odpowiednią **interlinię**. Kolejna linia tekstu wyświetlonego przy użyciu fontu 12×6 powinna być 13 pikseli niżej niż poprzednia; podobnie dla fontu 18×10 linie tekstu powinny być rozmieszczone w pionie co 20 pikseli.

Listing 11.8. Procedury ustawiania przekształcenia i koloru oraz sprzątania

C

```
void SetupTextFrame ( GLint width, GLint height )
{
  GLfloat pm[16];

  M4x4Orthof ( pm, NULL, -0.5, (float)width-0.5,
               (float)height-0.5, -0.5, -1.0, 1.0 );
  glBindBuffer ( GL_UNIFORM_BUFFER, gcbuf );
  glBufferSubData ( GL_UNIFORM_BUFFER, gcofs[0], 16*sizeof(GLfloat), pm );
  ExitIfGLError ( "SetupTextFrame" );
} /*SetupTextFrame*/

void SetTextForeground ( GLfloat fg[4] )
{
  glBindBuffer ( GL_UNIFORM_BUFFER, gcbuf );
  glBufferSubData ( GL_UNIFORM_BUFFER, gcofs[1], 4*sizeof(GLfloat), fg );
  ExitIfGLError ( "SetForeground" );
} /*SetTextForeground*/

void SetTextBackground ( GLfloat bk[4] )
{
  glBindBuffer ( GL_UNIFORM_BUFFER, gcbuf );
  glBufferSubData ( GL_UNIFORM_BUFFER, gcofs[2], 4*sizeof(GLfloat), bk );
  ExitIfGLError ( "SetBackground" );
} /*SetTextBackground*/

void TextCleanup ( void )
{
  glUseProgram ( 0 );
  glDeleteProgram ( text_program_id );
  glDeleteBuffers ( 1, &gcbuf );
  ExitIfGLError ( "TextCleanup" );
} /*TextCleanup*/
```

Listing 11.9 przedstawia procedurę `ConstructEmptyVAO`, która tworzy obiekt tablicy wierzchołków (VAO) oraz procedurę `DeleteEmptyVAO`, która go niszczy. Obiekt ten pozostanie pusty, tj. nie będą w nim opisane *żadne* tablice atrybutów wierzchołków, ale jest potrzebny do uruchomienia potoku przetwarzania grafiki, a położenia wierzchołkom wprowadzonym do potoku zostaną nadane przez szader wierzchołków. W drugiej i trzeciej aplikacji pusty obiekt tablicy wierzchołków przyda się też do rysowania innych obiektów niż tekst, na przykład płatów Béziera. Dlatego procedura, która go tworzy, może być wywołana ponownie i wtedy natychmiast wykonuje powrót. Z tego też powodu procedura `TextCleanup` nie usuwa tego obiektu; należy go usunąć za pomocą procedury `DeleteEmptyVAO`, gdy przestanie być potrzebny do wszystkich zastosowań w aplikacji.

Listing 11.9. Procedury tworzenia i likwidacji pustego VAO

C

```
GLuint empty_vao = GL_INVALID_INDEX;

void ConstructEmptyVAO ( void )
{
  if ( empty_vao == GL_INVALID_INDEX ) {
    glGenVertexArrays ( 1, &empty_vao );
    ExitIfGLError ( "ConstructEmptyVAO" );
  }
} /*ConstructEmptyVAO*/

void DeleteEmptyVAO ( void )
{
  if ( empty_vao != GL_INVALID_INDEX ) {
    glBindVertexArray ( 0 );
    glDeleteVertexArrays ( 1, &empty_vao );
    empty_vao = GL_INVALID_INDEX;
    ExitIfGLError ( "DeleteEmptyVAO" );
  }
} /*DeleteEmptyVAO*/
```

11.4. Aplikacja pierwsza C

Świeżo nabytej umiejętności pisania użyjemy do wyświetlania położenia obserwatora, tj. współrzędnych początku układu obserwatora w układzie świata — w czasie, gdy aplikacja jest w trybie obracania obserwatora wokół obiektu. Na listingu 11.10 są pokazane zmiany w kodzie części „graficznej" aplikacji; trzeba dodać deklaracje kilku zmienych globalnych i dopisać bardzo już niewiele instrukcji, aby wszystko pięknie działało.

W zmiennej `trans.eyepos` (listing 10.6) będziemy zapisywać współrzędne położenia obserwatora — to te liczby chcemy wyświetlać w oknie. Czwarty element tablicy jest potrzebny, bo zawiera ona wektor współrzędnych jednorodnych, z których czwarta zawsze będzie jedynką, a więc pierwsze trzy liczby są też współrzędnymi kartezjańskimi. W tablicy `fonts` zapamiętamy wskaźniki struktur reprezentujących fonty; wartość zmiennej `font` ma być równa jednemu z tych wskaźników, identyfikując bieżąco używany font. Zmienna `vptext` jest wskaźnikiem struktury reprezentującej napis do wyświetlenia.

Do procedury `InitMyWorld` zostały dopisane instrukcje, które przygotowują program szaderów do pisania, tworzą oba fonty i obiekt tekstu. W bloku zmiennych jednolitych `MyGC` są zapisywane kolory tekstu — będzie wyświetlany niebieski tekst na przezroczystym tle. Nowa procedura `NotifyViewerPos` tworzy i zapisuje w pamięci GPU napis ze współrzędnymi początkowego położenia obserwatora.

W procedurze `ResizeMyWorld` zostało dodane wywołanie procedury `SetupTextFrame`. Ponieważ przekształcenia brył widzenia na kostkę standardową, używane do rysowania obiektu (dwudziestościanu) i do wyświetlania tekstu, są reprezentowane przez macierze

Listing 11.10. Dodatkowe zmienne i zmienione procedury „graficzne” aplikacji pierwszej C

C

```
myTextObject *vptext;
myFontObject *fonts[2], *font;

void InitMyWorld ( int argc, char *argv[], int width, int height )
{
  static GLfloat fg[4] = { 0.0, 0.0, 1.0, 1.0 };
  static GLfloat bk[4] = { 0.0, 0.0, 0.0, 0.0 };

  LoadMyShaders ();
  memset ( &trans, 0, sizeof(TransBlock) );
  memset ( &light, 0, sizeof(LSBlock) );
  trans.trbuf = NewUniformTransBlock ();
  light.lsbuf = NewUniformLightBlock ();
  LoadTextShaders ();
  TimerInit ();
  .... /* instrukcje bez zmian */
  InitViewMatrix ();
  ConstructIcosahedronVAO ();
  InitLights ();
  font = fonts[0] = NewFont18x10 ();
  fonts[1] = NewFont12x6 ();
  vptext = NewTextObject ( 60 );
  SetTextForeground ( fg );
  SetTextBackground ( bk );
  NotifyViewerPos ();
  ResizeMyWorld ( width, height );
} /*InitMyWorld*/

void NotifyViewerPos ( void )
{
  GLchar s[60];

  sprintf ( s, "x = %5.2f, y = %5.2f, z = %5.2f",
            trans.eyepos[0], trans.eyepos[1], trans.eyepos[2] );
  SetTextObjectContents ( vptext, s, 0, win_height-1, font );
} /*NotifyViewerPos*/

void ResizeMyWorld ( int width, int height )
{
  .... /* instrukcje bez zmian */
  SetupTextFrame ( width, height );
} /*ResizeMyWorld*/

void RedrawMyWorld ( void )
{
```

```
46:   glClearColor ( 1.0, 1.0, 1.0, 1.0 );
47:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
48:   glEnable ( GL_DEPTH_TEST );
49:   DrawIcosahedron ( option, enlight );
50:   if ( showpos )
51:     DisplayTextObject ( vptext );
52:   glFlush ();
53:   ExitIfGLError ( "RedrawMyWorld" );
54: } /*RedrawMyWorld*/
55:
56: void InitViewMatrix ( void )
57: {
58:   GLfloat m[16];
59:
60:   M4x4Translatef ( m, -viewer_pos0[0], -viewer_pos0[1], -viewer_pos0[2] );
61:   memcpy ( trans.eyepos, viewer_pos0, 4*sizeof(float) );
62:   LoadVPMatrix ( &trans );
63: } /*InitViewMatrix*/
64:
65: void RotateViewer ( int delta_xi, int delta_eta )
66: {
67:   .... /* instrukcje bez zmian */
68:   LoadVPMatrix ( &trans );
69:   NotifyViewerPos ();
70: } /*RotateViewer*/
71:
72: char ProcessCharCommand ( char charcode )
73: {
74:   int oldoption;
75:
76:   oldoption = option;
77:   switch ( toupper ( charcode ) ) {
78: case 'M':
79:     font = font == fonts[0] ? fonts[1] : fonts[0];
80:     NotifyViewerPos ();
81:     return true;
82:     .... /* reakcje na pozostałe litery bez zmian */
83:   }
84:   return option != oldoption;
85: } /*ProcessCharCommand*/
86:
87: void DeleteMyWorld ( void )
88: {
89:   int i;
90:
91:   glUseProgram ( 0 );
92:   glDeleteProgram ( program_id[0] );
```

```
93:   glDeleteProgram ( program_id[1] );
94:   glDeleteBuffers ( 1, &trans.trbuf );
95:   glDeleteBuffers ( 1, &light.lsbuf );
96:   glDeleteVertexArrays ( 1, &icos_vao );
97:   glDeleteBuffers ( 3, icos_vbo );
98:   DeleteTextObject ( vptext );
99:   for ( i = 0; i < 2; i++ )
100:    DeleteFontObject ( fonts[i] );
101:  TextCleanup ();
102:  DeleteEmptyVAO ();
103:  ExitIfGLError ( "DeleteMyWorld" );
104: } /*DeleteMyWorld*/
```

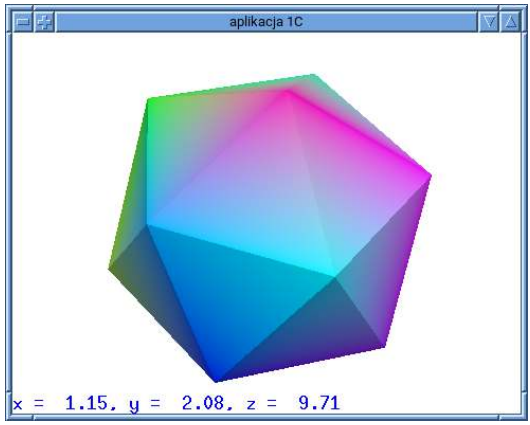
przechowywane w różnych buforach, podłączonych do różnych punktów dowiązania, można te macierze skonstruować i przesłać do buforów w tym miejscu.

Instrukcja dodana do procedury `RedrawMyWorld` wyświetla tekst, jeśli w chwili wywołania tej procedury aplikacja jest w trybie obracania obserwatora. Test głębokości, który mógłby „zepsuć” wyświetlany tekst, zostaje przez procedurę `DisplayTextObject` wyłączony.

Procedura `RotateViewer` ma dodatkową instrukcję, której zadaniem jest przygotowanie napisu — współrzędnych nowego położenia obserwatora — do wyświetlenia na ekranie; zajmuje się tym procedura `NotifyViewerPos`.

Instrukcje dodane do procedury `KeyboardFunc` powodują przełączenie fontu po naciśnięciu klawisza z literą `M`; font wybrany do wyświetlania napisów jest zmieniany na ten drugi, po czym obraz trzeba ponownie narysować.

Wreszcie, do procedury `Cleanup` zostały dopisane instrukcje likwidujące obiekty tekstu i fonty oraz wywołanie procedury `TextCleanup`, która likwiduje program szaderów wyświetlający tekst i bufor z blokiem `MyGC` używany przez ten program. Likwidujemy też pusty obiekt tablicy wierzchołków.



Rysunek 11.2. Okno aplikacji pierwszej C

Jedyną zmienioną procedurą w części „okienkowej” jest MouseFunc (listing 11.11), która powiększyła się o instrukcje unieważniające obraz w oknie po zmianie trybu działania aplikacji: po wejściu w tryb obracania obserwatora napis w oknie ma się pojawić, a po wyjściu z tego trybu ma zniknąć, czyli w obu przypadkach obraz ma się zmienić.

Listing 11.11. Zmiany w procedurze MouseFunc

C

```
1: void MouseFunc ( int button, int state, int x, int y )
2: {
3:   switch ( app_state ) {
4: case STATE_NOTHING:
5:     if ( button == GLUT_LEFT_BUTTON && state == GLUT_DOWN ) {
6:       last_xi = x,  last_eta = y;
7:       app_state = STATE_TURNING;
8:       glutPostWindowRedisplay ( WindowHandle );
9:     }
10:    break;
11: case STATE_TURNING:
12:    if ( button == GLUT_LEFT_BUTTON && state != GLUT_DOWN ) {
13:      app_state = STATE_NOTHING;
14:      glutPostWindowRedisplay ( WindowHandle );
15:    }
16:    break;
17: default:
18:    break;
19:  }
20: } /*MouseFunc*/
```

11.5. Uzupełnienia

11.5.1. Bloki i bufory magazynowe

Bloki zmiennych jednolitych zapewniają bardzo szybki dostęp do umieszczonych w nich danych, ale mają pewne ograniczenia. Po pierwsze istnieje limit wielkości takiego bloku; specyfikacja OpenGL gwarantuje możliwość używania bloków o wielkości 16 KB, przy czym implementacje mogą mieć większy limit, ale też niezbyt duży, na przykład 64 KB. Limit w konkretnym systemie możemy poznać, wywołując procedurę

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

której drugi parametr jest adresem zmiennej typu GLint.

Po drugie szadery nie mogą przypisywać wartości zmiennym jednolitym, które z tego powodu nie nadają się do składowania wyników obliczeń wykonywanych przez GPU. Jeśli pojemność bloku zmiennych jednolitych jest za mała lub wyniki obliczeń mają być zapamiętane (aby ich użyć później do wykonania obrazu lub aby CPU mogła je odczytać z pamięci

GPU), zamiast bloku zmiennych jednolitych możemy użyć **bloku magazynowego**; rozmiar takich bloków może być znacznie większy[8] i szadery mają prawo w nich pisać.

W programie w GLSL-u blok magazynowy deklarujemy podobnie jak blok zmiennych jednolitych, zastępując słowo kluczowe `uniform` słowem `buffer`. Deklaracja może wyglądać tak[9]:

```
layout(binding=1) buffer MyText {
    int  x, y, l;
    uint text[]; /* ostatnia tablica w bloku magazynowym może nie mieć
                    podanej długości */
} text;
```

Bufor dla bloku magazynowego (SSBO — *shader storage buffer object*) tworzymy tak samo jak bufory dla bloków zmiennych jednolitych i przesyłamy dane do niego w taki sam sposób.

Listing 11.12 przedstawia procedury ułatwiające korzystanie z buforów magazynowych, podobne do zamieszczonych na listingu 10.5 procedur obsługujących bufory z blokami zmiennych jednolitych.

Aby dane w buforze magazynowym były dostępne dla szadera, należy ten bufor przywiązać do celu `GL_SHADER_STORAGE_BUFFER`, który podobnie jak cel `GL_UNIFORM_BUFFER` jest indeksowany, tj. ma tablicę punktów dowiązania. Po skompilowaniu i połączeniu programu szaderów potrzebujemy znaleźć przesunięcia pól w tym bloku względem jego początku, aby móc im przypisywać wartości przy użyciu procedury `glBufferSubData`.

Dostęp do pól w bloku magazynowym daje procedura `GetAccessToStorageBlock`. Pierwszym jej parametrem jest identyfikator programu, drugim liczba n pól w bloku, a trzeci parametr to tablica wskaźników do napisów, z których pierwszy jest nazwą globalną bloku, a kolejne n to nazwy pól w tym bloku (poprzedzone nazwą bloku i kropką, np. `"MyText.text"`). Kolejne parametry wskazują zmienne, którym procedura ma przypisać wyniki: wielkość bloku w bajtach[10], tablicę przesunięć kolejnych pól w bloku i numer punktu dowiązania przydzielony dla tego bloku magazynowego w celu `GL_SHADER_STORAGE_BUFFER`. Poszczególne informacje są odczytywane z programu przez „uniwersalne" procedury `glGetProgramResourceIndex`, i `glGetProgramResourceiv`[11], z których pierwsza

[8] Bloki magazynowe są dostępne w wersji OpenGL-a i GLSL-a 4.3 i nowszych. Specyfikacja dopuszcza używanie bloków o wielkości 2^{27} bajtów, przy czym implementacje mogą podnieść ten limit nawet do $2^{31}-1$ bajtów, jeśli GPU ma dostatecznie dużo pamięci. Ceną za to może być nieco wolniejszy dostęp do danych.

[9] Podobny kwalifikator `layout` może też poprzedzać deklarację bloku zmiennych jednolitych; określa on numer punktu dowiązania (w celu `GL_UNIFORM_BUFFER`), który dotąd określaliśmy przy użyciu procedury `glUniformBlockBinding`. Jeśli użyjemy obu sposobów określania numeru punktu dowiązania, to ostatnie słowo ma procedura `glUniformBlockBinding`. Analogiczną rolę dla bloków magazynowych spełnia procedura `glShaderStorageBlockBinding`.

[10] Wielkość ta jest obliczona na podstawie deklaracji bloku; w szczególności, jeśli na jego końcu jest tablica, to brana jest jej długość podana w deklaracji, a nie faktyczna długość, jaką ta tablica będzie miała w utworzonym przez aplikację buforze.

[11] Procedur tych można używać także do odczytywania informacji na temat bloków zmiennych jednolitych, natomiast nie ma odpowiednika „wysokopoziomowej" procedury `glGetActiveUniformsiv` dla bloków magazynowych.

Listing 11.12. Procedury `GetAccessToStorageBlock` i `NewStorageBuffer`

C

```
char GetAccessToStorageBlock ( GLuint prog, int n, const GLchar **names,
                               GLint *size, GLint *ofs, GLuint *bpoint )
{
  int    i;
  GLuint prop, ind;

  ind = glGetProgramResourceIndex ( prog, GL_SHADER_STORAGE_BLOCK,
                                    names[0] );
  if ( ind == GL_INVALID_INDEX )
    return false;
  prop = GL_BUFFER_DATA_SIZE;
  glGetProgramResourceiv ( prog, GL_SHADER_STORAGE_BLOCK, ind,
                           1, &prop, 1, NULL, size );
  if ( bpoint ) {
    prop = GL_BUFFER_BINDING;
    glGetProgramResourceiv ( prog, GL_SHADER_STORAGE_BLOCK, ind,
                           1, &prop, 1, NULL, (GLint*)bpoint );
  }
  if ( n > 0 ) {
    prop = GL_OFFSET;
    for ( i = 0; i < n; i++ ) {
      ind = glGetProgramResourceIndex ( prog, GL_BUFFER_VARIABLE,
                                        names[i+1] );
      glGetProgramResourceiv ( prog, GL_BUFFER_VARIABLE, ind,
                               1, &prop, 1, NULL, &ofs[i] );
    }
  }
  ExitIfGLError ( "GetAccessToStorageBlock" );
  return true;
} /*GetAccessToStorageBlock*/

GLuint NewStorageBuffer ( GLint size, GLuint bp )
{
  GLuint buf;

  glGenBuffers ( 1, &buf );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, bp, buf );
  glBufferData ( GL_SHADER_STORAGE_BUFFER, size, NULL, GL_DYNAMIC_DRAW );
  ExitIfGLError ( "NewStorageBuffer" );
  return buf;
} /*NewStorageBuffer*/

void AttachStorageBlockToBP ( GLuint prog, const GLchar *name, GLuint bp )
{
  GLuint ind;
```

```
46:
47:   ind = glGetProgramResourceIndex ( prog, GL_SHADER_STORAGE_BLOCK, name );
48:   glShaderStorageBlockBinding ( prog, ind, bp );
49:   ExitIfGLError ( "AttachStorageBlockToBP" );
50: } /*AttachStorageBlockToBP*/
```

podaje indeks obiektu o przekazanej nazwie (np. pola w bloku) w tablicy symboli programu szaderów, a druga podaje wybraną informację liczbową (np. wielkość bloku lub przesunięcie pola względem początku bloku). Ostatni parametr wskazuje zmienną, w której ma być zapamiętany numer punktu dowiązania bloku, odczytany z programu szaderów. Parametr ten może mieć też wartość `NULL`, jeśli numer punktu dowiązania jest aplikacji znany.

Procedura `NewStorageBuffer` rezerwuje bufor magazynowy o podanej wielkości. Procedura `AttachStorageBlockToBP` umożliwia uzgodnienie numeru punktu dowiązania w kolejnych programach szaderów korzystajcych z tego samego bloku magazynowego. Procedury te są zbudowane tak samo jak procedury `NewUniformBuffer` i `AttachUniformBlockToBP` z listingu 10.5 i *w zasadzie* tak samo się ich używa.

Kontekst OpenGL-a w każdej implementacji ma ograniczenia, które mogą być luźniejsze niż to gwarantuje specyfikacja. Dotyczy to m.in. liczby bloków zmiennych jednolitych lub bloków magazynowych, do których szader określonego typu może mieć jednoczesny dostęp, a także długości tablic punktów dowiązania w celach `GL_UNIFORM_BUFFER` i `GL_SHADER_STORAGE_BUFFER`. Określone przez specyfikację [1] limity są w rozdziale 23 podane w tabelach, w których można znaleźć też nazwy symboliczne parametrów oraz nazwy procedur (np. `glGetIntegerv` lub `glGetIntegeri_v`), które aplikacja może wywołać z tymi parametrami, aby poznać ograniczenie implementacji OpenGL-a zainstalowanej na konkretnym sprzęcie, na którym została uruchomiona.

Minimalne gwarantowane przez specyfikację OpenGL 4.5 długości tablic punktów dowiązania we wspomnianych wyżej celach to odpowiednio 84 i 8. Długości tych tablic podane przez procedurę `glGetIntegerv` wywołaną przez aplikację działającą na sprzęcie użytym do napisania tej książki, to 84 i 96. Według specyfikacji szadery mogą mieć dostęp do 14 bloków zmiennych jednolitych i do 8 bloków magazynowych. Ten ostatni limit w implementacji może być większy, na przykład 16, ale pisząc aplikację przeznaczoną do rozpowszechniania, warto starać się zmieścić w ciaśniejszym gorsecie. Dlatego różne programy szaderów korzystają z tych samych punktów dowiązania, aby mieć dostęp do bloków magazynowych o różnych zawartościach. Oczywiście, przed *każdym* uruchomieniem obliczeń na GPU trzeba zadbać o to, aby *wszystkie* bufory magazynowe potrzebne programowi szaderów, który ma wykonać swoją pracę, były przywiązane do odpowiednich punktów dowiązania. Służy do tego instrukcja

```
glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, i, buf );
```

której drugi parametr jest numerem punktu dowiązania w celu `GL_SHADER_STORAGE_BUFFER`. Trzeci parametr jest podanym przez procedurę `glGenBuffers` identyfikatorem bufora, do którego zostały wpisane odpowiednie dane.

Procedura `glBindBufferRange` ma pięć parametrów, z których pierwsze trzy są takie same jak parametry procedury `glBindBufferBase`. Przywiązuje ona do wybranego punktu dowiązania *część* bufora, której początek i długość są określone przez ostatnie dwa parametry. Umożliwia to zmieszczenie w jednym buforze dwóch lub więcej tablic o nieustalonych zawczasu długościach i mieszczących dane różnych typów; przypomnijmy, że składnia deklaracji bloku magazynowego dopuszcza niepodanie długości tablicy, która jest *ostatnim* polem bloku. Części bufora z tablicami, których długości aplikacja obliczyła podczas swojego działania, można dowiązać do różnych punktów dowiązania. Początek przywiązywanej części jest n-tym bajtem od początku bufora; zależnie do celu, do którego część bufora jest przywiązywana, liczba n *musi* być całkowitą wielokrotnością stałej `GL_SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT` albo `GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT`. Ponadto suma długości podanej części bufora i liczby n nie może być większa niż długość całego bufora.

Kolejne dwie procedury umożliwiają skrócenie kodu aplikacji, przywiązując w jednym wywołaniu kilka buforów lub ich części do kolejnych punktów dowiązania w podanym celu. Ich nazwy to `glBindBuffersBase` i `glBindBuffersRange`; identyfikatory buforów oraz początki i wielkości części buforów przywiązywanych przez tę drugą procedurę podaje się w tablicach. Procedury te są dostępne w wersji OpenGL 4.4 i nowszych.

W większych aplikacjach różne programy szaderów współpracują, zapisując i odczytując zawartość tych samych buforów widocznych jako bloki magazynowe; jeśli dany blok ma ten sam numer punktu dowiązania w tych programach, to nie trzeba bufora przywiązywać przed każdym wykonaniem obliczeń, ale trzeba dbać o to, by dowiązania buforów pozostawione po wcześniejszych obliczeniach na GPU były właściwe. Pomocy w uruchamianiu aplikacji mogą dostarczyć procedury takie jak `glGetIntegerv` i `glGetIntegeri_v`, które oprócz limitów implementacji są w stanie podawać informacje o bieżącym stanie kontekstu OpenGL-a. Pierwsza z nich, wywołana z parametrem `GL_SHADER_STORAGE_BUFFER_BINDING` lub `GL_UNIFORM_BUFFER_BINDING` podaje identyfikator bufora przywiązanego do wskazanego celu. Druga podaje identyfikator bufora przywiązanego do określonego punktu dowiązania w tym celu. Czytelników chcących zgłębić temat zachęcam do przejrzenia spisu innych informacji, których procedury te mogą dostarczać, w specyfikacji lub na stronie [7].

11.5.2. *Szadery w kodzie SPIR-V

Na przykładzie aplikacji pierwszej C można prześledzić pewne problemy związane z używaniem szaderów skompilowanych do postaci SPIR-V. Bloki zmiennych jednolitych `MyFont` i `MyText` szaderów z listingów 11.1 i 11.2 używają domyślnego układu `shared`, w którym 32-bitowe liczby całkowite w tablicach są upakowane bez przerw. W takich liczbach pakujemy wzorce czcionek i kody ASCII kolejnych znaków napisu. Zewnętrzny kompilator GLSL-a, `glslangValidator`, nie obsługuje tego układu; w przypadku niepodania kwalifikatora układu samowolnie i chyłkiem (bez ostrzeżenia) zmienia układ na `std140`, a próba skompilowania szadera z deklaracją `layout(shared) uniform ...` kończy się otrzymaniem komunikatu o błędzie. W rozdziale 9 wspomniałem, że w układzie `std140` każdy element tablicy liczb zajmuje tyle miejsca, co wektor o czterech współrzędnych złożony z takich

liczb[12]. W związku z tym aplikacja, przesyłając dane do bufora, musiałaby odpowiednio „porozsuwać" elementy tablicy, co trzeba uznać za wyjątkowo niedołężny i pracochłonny sposób marnowania miejsca w pamięci GPU.

Problem właściwego wykorzystania pamięci możemy rozwiązać w ten sposób, że zamiast bloków zmiennych jednolitych użyjemy bloków magazynowych. W tym przypadku możemy użyć (niedozwolonego dla bloków zmiennych jednolitych) układu `std430`, w którym nie ma przerw między elementami tablic typu `int`, `uint` ani `float`. Ale aby aplikacja działała, potrzebny jest sprzęt i sterownik zgodny ze specyfikacją OpenGL 4.3 lub nowszą.

Zmiany szadera fragmentów umożliwiające używanie jego wersji reprezentowanej w kodzie SPIR-V są pokazane na listingu 11.13. Zmiany bloków zmiennych jednolitych na bloki magazynowe musimy dokonać także w treści szadera wierzchołków, aby w obu szaderach opisy tych bloków były identyczne. Dla każdego bloku magazynowego i bloku zmiennych jednolitych należy podać (w kwalifikatorze `layout(binding=...)`) numer punktu dowiązania tego bloku w celu `GL_SHADER_STORAGE_BUFFER` albo `GL_UNIFORM_BUFFER`.

Listing 11.13. Zmodyfikowany szader fragmentów do pisania tekstu

GLSL

```
#version 440

layout(location=0) out vec4 out_Colour;

layout(binding=0) uniform MyGC {
    mat4 pm;      /* macierz rzutowania */
    vec4 fg, bk;  /* kolory znakow i tła */
    int  h;       /* wysokość klatki */
  } gc;

layout(std430,binding=0) buffer MyFont {
    int  chw, chh;   /* szerokość i wysokość znaku */
    int  chf, chl;   /* kody pierwszego i ostatniego znaku */
    uint glyphs[];
  } font;

layout(std430,binding=1) buffer MyText {
    int  x, y;      /* pozycja pierwszego znaku */
    int  l;         /* długość napisu */
    uint text[];    /* upakowane znaki napisu */
  } text;

#define SETFRAG(C) ....       /* bez zmian */
#define EXTRACTBYTE(x,b) .... /* bez zmian */

```

[12] Również wektory o 2 lub 3 współrzędnych są wyrównywane do długości wektorów o 4 współrzędnych przez wstawienie między nie odstępów w tablicy. Alternatywne wobec opisanego tu rozwiązanie polega na umieszczeniu danych w tzw. buforze obrazu (*image buffer*, zobacz podrozdz. 26.4).

```
26: void main ( void )
27: {
28:   int   x0, y0;
29:   uint  c, r, mask, chrow;
30:
31:   x0 = int(gl_FragCoord.x) - text.x;
32:   if ( x0 < 0 || x0 >= font.chw*text.l )
33:     SETFRAG ( bk )
34:   y0 = gc.h - 2 - int(gl_FragCoord.y) - (text.y-font.chh);
35:   if ( y0 < 0 || y0 >= font.chh )
36:     SETFRAG ( bk )
37:   .... /* dalej tak, jak w liniach 36-59 na listingu 11.2 */
38: } /*main*/
```

Dla każdej zmiennej interfejsu kompilator domaga się podania położenia[13], w kwalifikatorze `layout(location=...)`. Okazuje się, że jeśli użyta wersja języka GLSL jest wcześniejsza niż 4.4, to wymaganie to uniemożliwia korzystanie z bloków interfejsu, tj. struktur z polami zawierającymi poszczególne atrybuty wierzchołków lub fragmentów. Zatem każdy atrybut musi być przekazany w osobnej „prostej" zmiennej o jawnie podanym położeniu albo na początku pliku źródłowego szadera należy podać dyrektywę co najmniej `#version 440` — i wtedy kwalifikator `layout(location=0)` powinien być podany przed całym blokiem interfejsu. W przedstawianym tu przykładzie także zmienną wyjściową szadera fragmentów, `out_Colour`, trzeba było opatrzyć kwalifikatorem położenia[14].

Jest jeszcze jeden problem. Kompilator (nie wypisując ostrzeżenia) ignoruje kwalifikator `layout(origin_upper_left)` zmiennej `gl_FragCoord`, w związku z czym trzeba wykonać obliczenia w domyślnym układzie współrzędnych, którego początek jest w dolnym lewym narożniku okna.

Deklaracja zmiennej `gl_FragCoord` z kwalifikatorem `origin_upper_left` (z linii 3 na listingu 11.2) została więc usunięta. Zamiast niej w bloku `MyGC` pojawiło się pole `h`, którego wartość będzie wysokością klatki w pikselach; w linii 34 jest ona użyta w nowym sposobie obliczania numeru wiersza we wzorcu znaku. Aplikacja musi nadać temu polu wartość równą wysokości okna w pikselach po każdej zmianie tej wysokości. Pozostałe instrukcje szaderów nie wymagały zmian.

11.5.3. *Badanie programów szaderów

Po wymianie karty graficznej w moim komputerze i zainstalowaniu nowych sterowników[15] aplikacja 1C zmodyfikowana tak, by używać szaderów reprezentowanych w kodzie SPIR-V

[13] Opis ten dotyczy kompilatora w wersji `10:11.0.0`; co ciekawe, wersja wcześniejsza, którą dysponowałem podczas pracy nad pierwszym wydaniem, nie nakładała tego wymagania.

[14] Numer położenia w tym przypadku jest numerem bufora obrazu, do którego trafi informacja wyprowadzana w danej zmiennej wyjściowej; bufor ramki związany z oknem na ekranie ma tylko jeden bufor obrazu, o numerze 0, ale pozaekranowy bufor ramki może mieć ich więcej.

[15] tego samego producenta!

przestała działać i konieczne okazało się przeprowadzenie śledztwa w tej sprawie. Błąd objawiał się w ten sposób, że procedura `glGetUniformBlockiv`, wywołana przez procedurę `GetAccessToUniformBlock` (zob. listing 10.5, linia 19), podawała wartość `GL_INVALID_INDEX`. To oznacza, że blok zmiennych jednolitych o nazwie `MyGC` w programie otrzymanym z połączenia szaderów reprezentowanych w kodzie SPIR-V był nieobecny, choć gdy aplikacja (za pomocą procedury `glCompileShader`) kompilowała *ten sam* kod źródłowy szaderów w GLSL-u, działała dalej bez zarzutu.

Przełom w śledztwie nastąpił po wywołaniu pokazanej na listingu 11.14 procedury `PrintProgramBlocks`. Jej zadaniem było odkrycie, jakie nazwy w programie mają bloki zmiennych jednolitych, bloki magazynowe i pola w tych blokach. Znając nazwę i rodzaj zmiennej lub innego obiektu w programie szaderów, możemy za pomocą procedury `glGetProgramResourceIndex` poznać jego indeks w tablicy będącej częścią programu. Z kolei, jeśli znamy indeks obiektu, to jego nazwę i inne atrybuty (np. przesunięcie pola w bloku) możemy poznać, wywołując procedurę, która udzieli potrzebnej informacji; w przypadku nazwy jest to procedura `glGetProgramResourceName`.

Wywołana w linii 9 procedura `glGetProgramInterfaceiv` przypisuje zmiennej `n` liczbę obiektów wskazanego typu: bloków zmiennych jednolitych, wszystkich zmiennych jednolitych (w tym pól w blokach), bloków magazynowych lub wszystkich pól w tych blokach. Następnie w pętli są odczytywane i wypisywane nazwy obiektów o kolejnych indeksach. Jeśli obiekty te są polami bloku zmiennych jednolitych lub bloku magazynowego, to wypisywane są też ich przesunięcia względem początku bloku albo położenia.

Co się okazało? Zamiast nazw bloków `MyGC`, `MyFont` i `MyText` pojawiły się nazwy `MyGC.gc`, `MyFont.font` i `MyText.text`, a zatem nazwy zewnętrzne bloków zostały rozszerzone o nazwy wewnętrzne tych bloków. Również nazwy pól w blokach zostały poprzedzone nazwami wewnętrznymi bloków, wskutek czego na przykład zamiast `MyGC.pm` należało szukać pola `MyGC.gc.pm`.

Poprawienie błędu polegało na dopisaniu alternatywnych tablic z napisami — nowymi nazwami bloków i pól w tych blokach, przy czym oryginalne napisy pozostały. Aplikacja najpierw wywołuje procedurę `GetAccessToUniformBlock` lub `GetAccessToStorageBlock`, podając jej oryginalne nazwy, a w razie niepowodzenia (sygnalizowanego przez wartość powrotną `false`) ponawia próbę, podając nazwy alternatywne. W ten sposób, gdybym z powrotem wymienił kartę graficzną i sterownik na te, których używałem wcześniej, lub próbował uruchomić aplikację na innym sprzęcie, ma ona szansę nadal działać.

Opisane tu procedury pomogły mi w szczególności wyjaśnić, co się stało po usunięciu wewnętrznej nazwy bloku zmiennych jednolitych z macierzami przekształceń (zobacz przypis 14 na s. 192). Blok ten ma zewnętrzną nazwę `TransBlock` i prywatną nazwę `trb`. Nazwy pól w tym bloku, które trzeba podać, aby uzyskać ich indeksy, są prefiksowane nazwą zewnętrzną bloku, na przykład `TransBlock.mm`, a w instrukcjach szadera trzeba je poprzedzać nazwą prywatną, na przykład `trb.mm`. Po usunięciu nazwy prywatnej w treści szadera pisze się już tylko same nazwy pól, na przykład `mm`. Okazało się, że wtedy również w tablicy utworzonej przez kompilator nazwy pól występowały bez poprzedzającej je nazwy

Listing 11.14. Procedury śledcze

C

```
void PrintResourceNames ( const char *txt, GLuint prog_id, GLuint interf )
{
  GLint   n, i, ofs;
  GLchar  name[256];
  GLsizei lgt;
  GLuint  prop = GL_OFFSET;

  printf ( "%s:\n", txt );
  glGetProgramInterfaceiv ( prog_id, interf, GL_ACTIVE_RESOURCES, &n );
  for ( i = 0; i < n; i++ ) {
    glGetProgramResourceName ( prog_id, interf, i, 256, &lgt, name );
    printf ( "  %3d: %s", i, name );
    switch ( interf ) {
  case GL_UNIFORM:
  case GL_BUFFER_VARIABLE:
      glGetProgramResourceiv ( prog_id, interf, i, 1, &prop, 1, NULL,
                               &ofs );
      if ( ofs >= 0 )
        printf ( "  ofs = %d\n", ofs );
      else {
        loc = glGetProgramResourceLocation ( prog, interf, name );
        printf ( "  loc = %d\n", loc );
      }
      break;
  default:
      printf ( "\n" );
      break;
    }
  }
  ExitIfGLError ( "PrintResourceNames" );
} /*PrintResourceNames*/

void PrintProgramResources ( GLuint prog_id, const char *name )
{
  if ( !glIsProgram ( prog_id ) )
    ExitOnError ( "PrintProgramResources" );
  printf ( "program %s:\n", name );
  PrintResourceNames ( "Uniform blocks", prog_id, GL_UNIFORM_BLOCK );
  PrintResourceNames ( "Uniform variables", prog_id, GL_UNIFORM );
  PrintResourceNames ( "Storage blocks", prog_id, GL_SHADER_STORAGE_BLOCK );
  PrintResourceNames ( "Buffer variables", prog_id, GL_BUFFER_VARIABLE );
} /*PrintProgramResources*/
```

zewnętrznej bloku. Co ciekawe, niestety, nazwa zewnętrzna pozostała w kodzie SPIR-V wygenerowanym przez zewnętrzny kompilator.

Można stąd wyciągnąć dwa morały. Po pierwsze, gdy coś nie działa, zamiast tracić głowę, lepiej jest starać się zrozumieć problem, co uczyniwszy, damy radę go naprawić. Są procedury w OpenGL-u, które mogą w tym pomóc. Drugi morał jest chyba trochę mniej optymistyczny: chcąc rozpowszechniać aplikację, koniecznie trzeba ją przedtem przetestować na wielu różnych i różnie wyposażonych komputerach. Mało kto ma w domu aż tyle sprzętu.

11.6. Ćwiczenia

1. Dokonaj odpowiednich przeróbek i napisz dodatkowe procedury umożliwiające osobne podawanie (i zmienianie) napisu, jego położenia i fontu.
2. Przerób aplikację (program w C i szadery) tak, aby kolory znaków i tła były związane z obiektem tekstowym (w bloku `MyText`, a nie `MyGC`).
3. Przerób aplikację tak, aby kolory znaków i tła były podawane jako atrybuty wierzchołków obszaru tekstu. Wykorzystaj to do wyświetlenia tekstu, w którym kolor znaków i kolor tła zmieniają się płynnie.
4. Wymyśl i wypróbuj najprostszy sposób, aby dwukrotnie powiększyć tekst, czyli spowodować, że każdy piksel wzorca znaku w używanym foncie będzie odwzorowany na kwadrat 2×2 piksele.
5. Napisz szader fragmentów i komplet procedur na CPU umożliwiające wyświetlanie tekstu złożonego z wielu linii (rozdzielonych znakami `\n`). Do reprezentacji tekstu trzeba wprowadzić dodatkową tablicę indeksów początków poszczególnych linii w tekście, przy czym dla uproszczenia można ustalić maksymalną liczbę (np. rzędu kilkadziesiąt, i tak więcej na ekranie się nie zmieści) linii reprezentowanych przez jeden obiekt tekstowy. Rozwiązanie ćwiczenia wymaga wprowadzenia odpowiednich modyfikacji do szaderów z listingów 11.1 i 11.2.
6. *Napisz procedurę umożliwiającą pisanie tekstu obróconego o całkowitą wielokrotność kąta prostego i/lub pisanie tekstu odbitego symetrycznie względem prostej pionowej lub poziomej.
7. *Napisz procedury umożliwiające pisanie tekstów pochylonych pod dowolnym kątem; obszar testu powinien być równoległobokiem, którego wierzchołki mają dodatkowy atrybut — wektor współrzędnych w układzie, którego początek jest jednym z wierzchołków równoległoboku, a wersory osi są równoległe do jego boków. Szader fragmentów powinien wybierać kolor na podstawie wyniku interpolacji tego atrybutu w etapie rasteryzacji zamiast współrzędnych fragmentu.
8. *Napisz procedury umożliwiające pisanie tekstów przy użyciu fontów proporcjonalnych, tj. takich, w których znaki mają różne szerokości. Znaki tego tekstu mogą być wyświetlane po kolei, tj. każdy znak osobno, co umożliwi łatwe obliczanie położeń kolejnych znaków.

9. *Dodaj do aplikacji 1C procedury podane na listingu 11.14 i za ich pomocą zbadaj bloki zmiennych jednolitych w programach szaderów używanych do rysowania dwudziestościanu. Zajrzyj na stronę [7] i przeczytaj opisy procedur `glGetProgramInterfaceiv` i `glGetProgramResourceiv`.

10. *Napisz aplikację, która przeczyta kod źródłowy wskazanych w wierszu poleceń szaderów w GLSL-u albo szadery skompilowane w formacie SPIR-V, połączy je w program i wypisze zawartość tablic z nazwami pól w blokach zmiennych jednolitych oraz indeksy tych pól. Za pomocą tej aplikacji przebadaj inne szadery, zwłaszcza napisane przez siebie.

12

Aplikacja pierwsza D

Zamienimy krawędzie i ściany dwudziestościanu w figury zakrzywione — łuki okręgów i trójkąty sferyczne położone na sferze jednostkowej, w którą jest wpisany dwudziestościan. Każda krawędź zostanie zamieniona na złożoną z krótkich odcinków łamaną o wierzchołkach na sferze. Każda trójkątna ściana zostanie zastąpiona przez wiele małych trójkątów o wierzchołkach na sferze, dzięki czemu otrzymamy obraz trudno odróżnialny od obrazu sfery. Całe to obliczenie zostanie wykonane przez szadery rozdrabniania, które wprowadzimy do aplikacji.

12.1. Szadery i programy szaderów

W dodatku do dotychczas używanych utworzymy trzy nowe programy szaderów, z szaderami rozdrabniania. Wszystkie te programy będą używać tego samego szadera wierzchołków, pokazanego na listingu 12.1. Szader ten nie wykonuje żadnych obliczeń, tylko kopiuje atrybuty wierzchołków (położenie i kolor) na wyjście.

Listing 12.1. Szader wierzchołków dla programów rozdrabniających

GLSL

```
#version 420

layout(location=0) in vec4 in_Position;
layout(location=1) in vec3 in_Colour;

out vec3 Colour;

void main ( void )
{
  gl_Position = in_Position;
  Colour = in_Colour;
} /*main*/
```

Pierwszy z nowych programów ma na celu narysowanie łuków okręgu położonych na sferze jednostkowej. Łuk będzie przybliżony łamaną złożoną z 10 odcinków. Użyty w tym programie szader sterowania rozdrabnianiem (listing 12.2) określa taki poziom podziału, a poza tym przesyła atrybuty wierzchołka (dostarczone przez szader wierzchołków) na swoje wyjście.

Prymitywy geometryczne, które mają być rozdrabniane, są nazywane **płatami** (*patches*); aby je narysować, odpowiednią procedurę (`glDrawArrays`, `glDrawElements` albo którąś z dotąd nieopisanych procedur rysujących) należy wywołać ze stałą `GL_PATCHES` podaną jako pierwszy parametr. Wcześniej trzeba poinformować OpenGL-a o liczbie wierzchołków każdego płata, wywołując procedurę `glPatchParameteri`, co będzie opisane dalej. Liczba wierzchołków płata jest też podana w kwalifikatorze `layout` w linii 5 — oczywiście każdy łuk ma dwa końce.

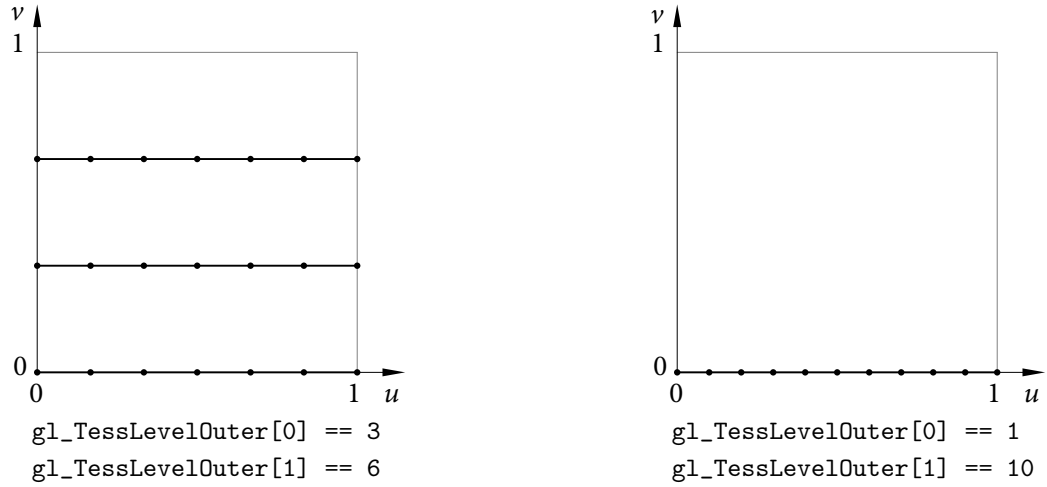
Listing 12.2. Pierwszy szader sterowania rozdrabnianiem

GLSL

```
1: #version 420
2:
3: #define MY_LEVEL 10
4:
5: layout(vertices=2) out;
6:
7: in vec3 Colour[];
8:
9: out vec3 TCColour[];
10:
11: void main ( void )
12: {
13:   if ( gl_InvocationID == 0 ) {
14:     gl_TessLevelOuter[0] = 1;          /* jedna łamana */
15:     gl_TessLevelOuter[1] = MY_LEVEL;   /* z tylu odcinków */
16:   }
17:   gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
18:   TCColour[gl_InvocationID] = Colour[gl_InvocationID];
19: } /*main*/
```

Szader sterowania rozdrabnianiem ma wejście i wyjście zadeklarowane jako tablice (dotyczy to nie tylko zmiennych wbudowanych `gl_in` i `gl_out`, ale też atrybutów dodatkowych, zobacz linie 7 i 9 na listingu 12.2), przy czym szader ma za zadanie przetworzyć *tylko jeden* wierzchołek. Numer tego wierzchołka jest podany w zmiennej wbudowanej `gl_InvocationID`, która służy do indeksowania tych tablic; w tym przypadku przyjmuje ona tylko wartości 0 lub 1.

Przetwarzając jeden (i tylko jeden) wierzchołek płata, szader sterowania rozdrabnianiem ma dostarczyć informację o tym, jak bardzo płat ma być rozdrobniony. Dlatego przypisania wartości elementom tablicy `gl_TessLevelOuter` (i `gl_TessLevelInner`, występujące w szaderach sterowania rozdrabnianiem opisanych dalej) są wykonywane w instrukcji wa-



Rysunek 12.1. Podziały dziedziny płata w trybie izolinii

runkowej — tylko wtedy, gdy zmienna `gl_InvocationID` ma wartość 0. Potok przetwarzania grafiki z wmontowanym szaderem rozdrabniania przedstawionym na listingu 12.3 pracuje w **trybie izolinii** (*isoline mode*). Oznacza to, że dziedzina płata przetwarzana przez etap rozdrabniania w potoku (znajdująca się między szaderem sterowania rozdrabnianiem a szaderem rozdrabniania) jest kwadratem jednostkowym, $[0,1] \times [0,1]$, w którym ma być wygenerowanych n linii poziomych (o współrzędnych $v = i/n$ dla $i = 0, \ldots, n-1$). Każda z tych linii ma być podzielona na m odcinków (rys. 12.1). Liczby n i m szader sterowania rozdrabnianiem ma wpisać do tablicy `gl_TessLevelOuter` i w liniach 14 i 15 to właśnie robi.

Szader rozdrabniania również ma dostęp do całej tablicy wierzchołków płata na wejściu i ma obliczyć i wyprowadzić dane dla *jednego* wierzchołka wyjściowego — odpowiadającego punktowi w dziedzinie płata wygenerowanemu przez etap rozdrabniania[1]. Zadaniem szadera na listingu 12.3 jest obliczenie wierzchołka łamanej przybliżającej łuk okręgu wielkiego sfery jednostkowej, którego końce są wierzchołkami płata podanymi na wejściu. Szader ma też obliczyć współrzędne wierzchołka w układzie kostki standardowej i podać jego kolor.

Kwalifikator `layout` w linii 3 określa, że ma być użyty tryb izolinii i każda (tj. w tym przypadku jedna) linia ma być podzielona na części o jednakowej długości. W tablicy `gl_in` są podane podstawowe atrybuty obu wierzchołków płata, w szczególności ich położenia. W tablicy `In` jest dodatkowy atrybut, tj. kolor każdego z tych wierzchołków. Wyjście szadera to zmienna wbudowana `gl_Position`, do której ma być przypisany wektor współrzędnych jednorodnych nowego wierzchołka w kostce standardowej i zmienna `TEColour`, której jest przypisywany kolor nowego wierzchołka. Oba te atrybuty zostną poddane interpolacji w celu określenia położenia (w układzie kostki standardowej) i koloru każdego fragmentu zrasteryzowanego odcinka łamanej wytworzonej przez rozdrabnianie płata.

[1]Oczywiście, etap rozdrabniania produkuje wiele punktów w dziedzinie płata; generowanie przez szader rozdrabniania wierzchołków wyjściowych dla tych punktów odbywa się równolegle.

Listing 12.3. Pierwszy szader rozdrabniania

GLSL

```
1: #version 420
2:
3: layout(isolines,equal_spacing) in;
4:
5: in vec3 TCColour[];
6:
7: out vec3 TEColour;
8:
9: uniform TransBlock {
10:     mat4 mm, mmti, vm, pm, vpm;
11:     vec4 eyepos;
12:   } trb;
13:
14: void main ( void )
15: {
16:   float t, t1;
17:   vec4  vert;
18:
19:   t = gl_TessCoord.x;  t1 = 1.0-t;
20:   vert = t1*gl_in[0].gl_Position + t*gl_in[1].gl_Position;
21:   vert.xyz = normalize ( vert.xyz );  vert.w = 1.0;
22:   gl_Position = trb.vpm * (trb.mm * vert);
23:   TEColour = t1*TCColour[0] + t*TCColour[1];
24: } /*main*/
```

Szader rozdrabniania ma też dostęp do bloku zmiennych jednolitych `TransBlock`, przechowującego macierze przekształceń, których złożenie jest przejściem od układu współrzędnych obiektu do kostki standardowej; to właśnie ten szader w naszym programie ma dokonać tego przekształcenia. Natomiast informacja wygenerowana przez etap rozdrabniania dziedziny płata jest podana w zmiennej wejściowej `gl_TessCoord` — współrzędne u, v punktu w dziedzinie są wartościami pól `x`, `y` tej zmiennej. Jest to zmienna wektorowa; w tym miejscu potrzebujemy tylko pierwszej jej współrzędnej, którą dla wygody przypisujemy (w linii 19) zmiennej `t`. W linii 20 dokonujemy interpolacji między wierzchołkami końcowymi płata (w tym miejscu można użyć funkcji `mix`, zobacz p. 9.13.1). W linii 21 punkt na odcinku jest rzutowany na sferę jednostkową. W linii 22 punkt na sferze jest przedstawiany kolejno w układzie współrzędnych świata i w układzie kostki standardowej, a w linii 23 obliczany jest jego kolor, przez interpolację kolorów końców odcinka.

Programy, które odcinki i trójkąty otrzymane w wyniku rozdrabniania po prostu rysują w kolorach otrzymanych przez interpolację kolorów wierzchołków (bez uwzględnienia oświetlenia) zawierają szader fragmentów przedstawiony na listingu 7.2. Zwróćmy uwagę, że zmienna wyjściowa szadera rozdrabniania ma inną nazwę (`TEColour`) niż zmienna wejściowa szadera fragmentów (`Colour`), ale zmienne te mają ten sam typ `vec3`, co umożliwia poprawne połączenie programów szaderów. Jeśli między szaderami trzeba przekazać więcej danych, to należy wskazać numery miejsc poszczególnych zmiennych wyjściowych

i wejściowych przy użyciu kwalifikatorów layout(location=*i*). Jeśli dane są przekazywane w zmiennej strukturalnej, to w komunikujących się szaderach opis struktury i nazwa zewnętrzna zmiennej muszą być identyczne.

Drugi program rozdrabniający ma przekształcić płaskie trójkąty, których wierzchołki leżą na sferze jednostkowej, w trójkąty sferyczne położone na tej sferze (oczywiście trójkąty te będą przybliżane przez wiele drobnych trójkątów płaskich). Szader sterowania rozdrabnianiem pokazany na listingu 12.4, podobnie jak szader z listingu 12.2, przepisuje atrybuty jednego wierzchołka trójkąta (położenie i kolor) z tablic wejściowych do tablic wyjściowych. Dla jednego z tych wierzchołków szader wpisuje do tablic gl_TessLevelOuter i gl_TessLevelInner informację o pożądanym stopniu rozdrobnienia. Kopiowanie atrybutów wierzchołka z wejścia na wyjście jest identyczne jak na listingu 12.2.

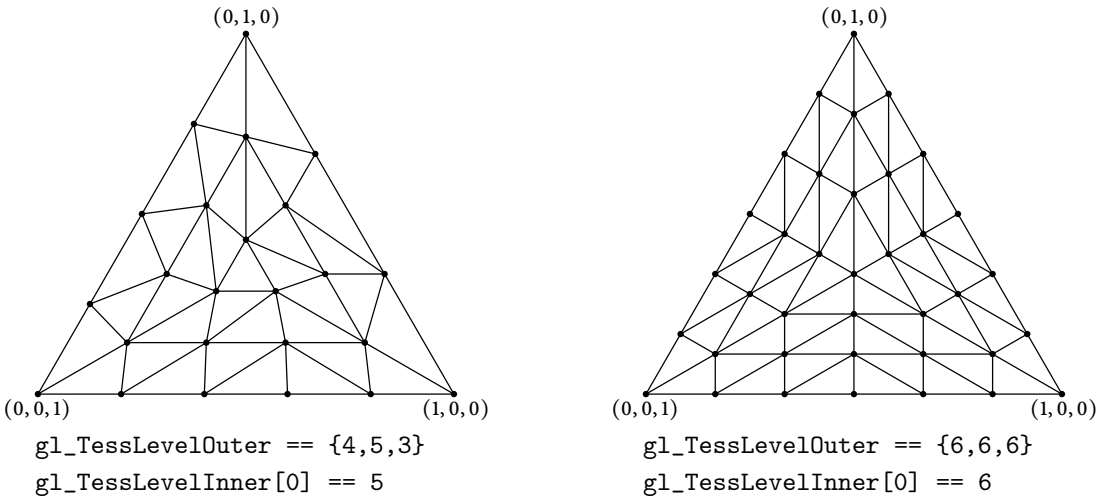
Listing 12.4. Drugi szader sterowania rozdrabnianiem

GLSL

```
#version 420

#define MY_LEVEL0 10
#define MY_LEVEL1 10

layout(vertices=3) out;

in vec3 Colour[];     /* wejście i wyjście tego szadera */
out vec3 TCColour[];  /* są takie same jak szadera z listingu 12.2 */

void main ( void )
{
  if ( gl_InvocationID == 0 ) {
    gl_TessLevelOuter[0] = MY_LEVEL0;
    gl_TessLevelOuter[1] = MY_LEVEL0;
    gl_TessLevelOuter[2] = MY_LEVEL0;
    gl_TessLevelInner[0] = MY_LEVEL1;
  }
  gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
  TCColour[gl_InvocationID] = Colour[gl_InvocationID];
} /*main*/
```

Na rysunku 12.2 są pokazane trójkąty podzielone w zależności od parametrów podanych w tablicach gl_TessLevelOuter i gl_TessLevelInner. Do pierwszej z tych tablic trzeba wpisać trzy liczby, które określają stopnie rozdrobnienia trzech boków trójkąta. Liczba podana w pierwszym elemencie drugiej tablicy określa liczbę trójkątnych „warstw cebuli" mających powstać wskutek podziału; nieco ściślej biorąc, liczba tych warstw jest połową podanej liczby.

Szader rozdrabniania na listingu 12.5 oblicza i przekształca do układu współrzędnych kostki standardowej jeden wierzchołek odpowiadający pewnemu punktowi z dziedziny płata wygenerowanemu przez etap rozdrabniania dziedziny. W przypadku płatów trójkątnych zmienna wektorowa gl_TessCoord zawiera współrzędne barycentryczne tego punktu



Rysunek 12.2. Podziały trójkątnej dziedziny płata

w układzie odniesienia wierzchołków trójkątnej dziedziny. Jak widać w liniach 15–16 i 19, interpolacja przy użyciu współrzędnych barycentrycznych jest bardzo łatwa do przeprowadzenia. Punkt płaskiego trójkąta w przestrzeni jest w linii 17 rzutowany na sferę jednostkową, a następnie przekształcany do układu kostki standardowej, identycznie jak w szaderze z listingu 12.3.

Listing 12.5. Drugi szader rozdrabniania

GLSL

```
1: #version 420
2:
3: layout(triangles,equal_spacing) in;
4:
5: in vec3 TCColour[];               /* ten fragment jest taki sam */
6: out vec3 TEColour;                /* jak w liniach 5-12 */
7: uniform TransBlock { .... } trb;  /* na listingu 12.3 */
8:
9: void main ( void )
10: {
11:   float s, t, u;
12:   vec4  vert;
13:
14:   s = gl_TessCoord.x;  t = gl_TessCoord.y;  u = gl_TessCoord.z;
15:   vert = s*gl_in[0].gl_Position + t*gl_in[1].gl_Position +
16:          u*gl_in[2].gl_Position;
17:   vert.xyz = normalize ( vert.xyz );  vert.w = 1.0;
18:   gl_Position = trb.vpm * (trb.mm * vert);
19:   TEColour = s*TCColour[0] + t*TCColour[1] + u*TCColour[2];
20: } /*main*/
```

Chcąc narysować trójkąt tak, aby otrzymać obraz części oświetlonej powierzchni gładkiej, na przykład sfery, należy dla każdego wierzchołka tego trójkąta podać wektor normalny tej powierzchni; podstawiając do modelu oświetlenia wynik interpolacji wektorów między wierzchołkami, szader fragmentów wygładzi obraz powierzchni. Obliczenia wektora normalnego mógłby dokonać szader geometrii, ale wygodniej jest użyć do tego szadera rozdrabniania[2]. Z powodów opisanych dalej szader fragmentów może też potrzebować wektora normalnego płaszczyzny trójkąta, który obliczy szader geometrii.

Dla sfery jednostkowej znamy *dokładne* rozwiązanie zadania: w dowolnym punkcie $\boldsymbol{p}$ sfery jednostkowej o środku w początku układu współrzędnych jej jednostkowy wektor normalny jest wektorem współrzędnych kartezjańskich punktu $\boldsymbol{p}$.

Podczas przekształcania figur geometrycznych wektory normalne podlegają innym regułom niż punkty tych figur i różnice punktów, tj. wektory swobodne. Zbadajmy to. Każde przekształcenie afiniczne przestrzeni trójwymiarowej jest opisane wzorem (5.5), który tu przypomnę:

$$f(\boldsymbol{p}) = L\boldsymbol{p} + \boldsymbol{t}.$$

Występuje w nim macierz L opisująca część liniową przekształcenia i wektor przesunięcia $\boldsymbol{t}$. Macierz ta jest nieosobliwa, gdy przekształcenie f jest różnowartościowe; zawsze powinniśmy dbać o to, aby w programach właśnie tak było. Różnica dowolnych punktów, $\boldsymbol{v} = \boldsymbol{p}_2 - \boldsymbol{p}_1$, jest wektorem swobodnym, którego obraz, $\boldsymbol{w} = f(\boldsymbol{p}_2) - f(\boldsymbol{p}_1)$, otrzymamy, mnożąc ten wektor przez macierz L:

$$\boldsymbol{w} = (L\boldsymbol{p}_2 + \boldsymbol{t}) - (L\boldsymbol{p}_1 + \boldsymbol{t}) = L(\boldsymbol{p}_2 - \boldsymbol{p}_1) = L\boldsymbol{v}.$$

Natomiast obliczenie wektora normalnego obrazu płaszczyzny w przekształceniu afinicznym f opiera się na następującym rachunku: równanie płaszczyzny o wektorze normalnym $\boldsymbol{n}$ przechodzącej przez punkt $\boldsymbol{p}_0$ ma postać

$$\boldsymbol{n}^T(\boldsymbol{p} - \boldsymbol{p}_0) = 0.$$

Obrazy w przekształceniu f wszystkich punktów płaszczyzny (tj. obrazy spełniających powyższe równanie punktów $\boldsymbol{p}$, w tym obraz punktu $\boldsymbol{p}_0$) spełniają równanie

$$0 = \boldsymbol{n}^T L^{-1} L(\boldsymbol{p} - \boldsymbol{p}_0) = (L^{-T}\boldsymbol{n})^T\big(f(\boldsymbol{p}) - f(\boldsymbol{p}_0)\big).$$

Wynika stąd, że wektor normalny obrazu płaszczyzny w przekształceniu f jest (z dokładnością do czynnika stałego) wartością wyrażenia $L^{-T}\boldsymbol{n}$.[3] Jeśli przekształcenie reprezentujemy w postaci jednorodnej, to macierz L jest górnym lewym blokiem 3×3 odpowiedniej macierzy 4×4, a L^{-T} jest górnym lewym blokiem transpozycji odwrotności tej macierzy.

[2] Dla wspólnego wierzchołka wielu trójkątów szader rozdrabniania wykonuje obliczenia tylko raz. Szader geometrii powtarzałby to samo obliczenie, przetwarzając każdy z tych trójkątów.

[3] Zatem wektor normalny *nie jest* wektorem swobodnym, zobacz podrozdział 5.2.

Jeśli przekształcenie f jest izometrią, to macierz L opisująca jego część liniową jest ortogonalna. W takim (i w tylko takim) przypadku $L^{-T} = L$. Mając obiekt określony w swoim lokalnym układzie, przekształcamy go najpierw do układu świata, następnie do układu obserwatora i wreszcie do układu kostki standardowej. Oświetlenie obliczamy w układzie świata, zatem musimy podać współrzędne wektora normalnego w tym układzie. W tym celu do bloku `TransBlock` zostało dodane jeszcze jedno pole, o nazwie `mmti`. Zawiera ono współczynniki transpozycji odwrotności macierzy przekształcenia modelu, podanej w polu `mm`. Za zapewnienie właściwej zawartości tych pól odpowiada aplikacja.

Uwaga: Ponieważ blok zmiennych jednolitych `TransBlock` jest używany przez wiele szaderów i przez wszystkie programy szaderów (z wyjątkiem opisanego w rozdz. 11 programu wyświetlającego tekst), trzeba zadbać o to, aby we wszystkich szaderach blok ten miał identyczny opis.

Listing 12.6. Trzeci szader rozdrabniania

GLSL

```
1: #version 420
2:
3: layout(triangles,equal_spacing,ccw) in;
4:
5: in vec3 TCColour[];  /* tak samo jak na listingu 12.3 */
6:
7: out NVertex {        /* tak samo jak na listingu 10.3 */
8:     vec3 Colour;
9:     vec3 Position;
10:    vec3 Normal;
11:  } Out;
12:
13: uniform TransBlock { .... } trb; /* blok TransBlock taki sam jak wszędzie */
14:
15: void main ( void )
16: {
17:   float s, t, u;
18:   vec4  vert, vv;
19:
20:   .... /* tu linie 14-16 z listingu 12.5 */
21:   vv = trb.mm * vert;
22:   gl_Position = trb.vpm * vv;
23:   Out.Colour = s*TCColour[0] + t*TCColour[1] + u*TCColour[2];
24:   Out.Position = vv.xyz/vv.w;
25:   Out.Normal = normalize ( mat3(trb.mmti) * vert.xyz );
26: } /*main*/
```

Szader oblicza współrzędne punktu na sferze tak samo jak szader na listingu 12.5; współrzędne te (w układzie modelu) są zapamiętane w zmiennej `vert`. W linii 21 dokonywane jest przejście do układu świata, a następnie (w linii 22) do układu kostki standardowej. Współrzędne (kartezjańskie) punktu w układzie świata, potrzebne w obliczeniach oświetlenia, są

przypisywane zmiennej wyjściowej w linii 24. W linii 25 następuje obliczenie wektora normalnego powierzchni w układzie świata: pola xyz zmiennej vert reprezentują ten wektor w układzie modelu. Wartością wyrażenia mat3(trans.mmti) jest górny lewy blok macierzy M^{-T} przechowywanej w zmiennej trans.mmti. Jednostkowym wektorem normalnym powierzchni w układzie świata jest unormowany iloczyn tego bloku (będącego macierzą L we wzorach na s. 283) i wektora normalnego w układzie modelu.

Listing 12.7. Drugi szader geometrii aplikacji 1D

GLSL

```
1: #version 420
2:
3: layout(triangles) in;
4: layout(triangle_strip,max_vertices=3) out;
5:
6: in NVertex { .... } In[];   /* blok taki jak wyjściowy na listingu 12.6 */
7:
8: out FVertex {
9:     vec3 Colour;
10:    vec3 Position;
11:    vec3 Normal, TNormal;
12:  } Out;
13:
14: void main ( void )
15: {
16:   int i;
17:   vec3 v1, v2, tnv;
18:
19:   v1 = In[1].Position - In[0].Position;
20:   v2 = In[2].Position - In[0].Position;
21:   tnv = normalize ( cross ( v1, v2 ) );
22:   for ( i = 0; i < 3; i++ ) {
23:     gl_Position = gl_in[i].gl_Position;
24:     Out.Position = In[i].Position;
25:     Out.Normal = In[i].Normal;
26:     Out.TNormal = tnv;
27:     Out.Colour = In[i].Colour;
28:     EmitVertex ();
29:   }
30:   EndPrimitive ();
31: } /*main*/
```

Zamiana wektora normalnego płaszczyzny trójkąta na wektor normalny zakrzywionej powierzchni przybliżanej przez ten trójkąt powoduje pewien problem w obliczeniach oświetlenia: podczas obliczania koloru fragmentu może się okazać, że obserwator i źródło światła znajdują się po tej samej stronie płaszczyzny trójkąta i po przeciwnych stronach płaszczyzny stycznej do powierzchni, lub na odwrót. Szader z listingu 10.4, znakomicie sprawdzający

się podczas rysowania brył wielościennych, może wtedy błędnie obliczyć kolor fragmentu, uznając, że obserwator widzi inną stronę powierzchni niż w rzeczywistości[4]. Pewnym (niedoskonałym) rozwiązaniem okazało się przekazanie szaderowi fragmentów *obu* wektorów normalnych. Wektory normalne sfery (lub innej powierzchni zakrzywionej, którą przyjdzie nam rysować) w wierzchołkach trójkątów będą obliczane przez szader rozdrabniania. Pokazany na listingu 12.7 szader geometrii do etapu rasteryzacji przekaże te wektory, $\boldsymbol{n}$, dołączając do nich (w polu `TNormal` bloku wyjściowego) obliczony przez siebie wektor normalny $\boldsymbol{m}$ trójkąta. Zwróćmy uwagę na kwalifikator `ccw` wejścia szadera na listingu 12.6: wybiera on taką orientację trójkątów otrzymanych z rozdrabnianego płata, aby obliczone przez szader geometrii wektory były zorientowane na zewnątrz sfery, tak jak podane przez szader rozdrabniania wektory normalne sfery, tj. aby iloczyn skalarny $\langle \boldsymbol{n}, \boldsymbol{m} \rangle$ był dodatni.

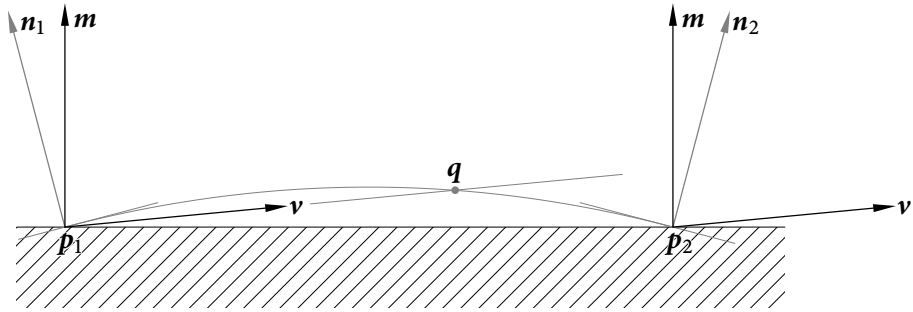
Listing 12.8 przedstawia nowy szader fragmentów, a raczej detale szadera z listingu 10.4 zmienione w celu uwzględnienia w obliczeniach oświetlenia obu wektorów normalnych. Do modelu oświetlenia, tj. wzoru (10.1), za $\boldsymbol{n}$ szader podstawia wektor normalny otrzymany w polu `In.Normal`, ale składnik bezpośredniego oświetlenia przez źródło światła jest dodawany wtedy, gdy iloczyn skalarny $\langle \boldsymbol{l}_i, \boldsymbol{n} \rangle$ ma ten sam znak co $\langle \boldsymbol{v}, \boldsymbol{m} \rangle$ (a nie $\langle \boldsymbol{v}, \boldsymbol{n} \rangle$).

Listing 12.8. Szader fragmentów do rysowania powierzchni zakrzywionych

GLSL

```
#version 420

#define MAX_NLIGHTS 8

in FVertex {
    vec3 Colour;
    vec3 Position;
    vec3 Normal, TNormal;
  } In;

.... /* tu wszystko takie, jak w liniach 11-48 na listingu 10.4 */

vec3 LambertLighting ( void )
{
  vec3  normal, lv, vv, Colour;
  float d, e, dist;
  uint  i, mask;

  normal = normalize ( In.Normal );
  vv = posDifference ( trb.eyepos, In.Position, dist );
  e = dot ( vv, In.TNormal );
  .... /* linie 57-78 szadera z listingu 10.4 bez zmian */
} /*LambertLighting*/
```

[4]Dokładniej: inną niż w rzeczywistości, której obraz chcemy otrzymać. Błąd ten objawił się podczas uruchamiania opisanej w następnym rozdziale aplikacji 1E, która wyświetla sfery na czarnym tle.



Rysunek 12.3. Test oświetlenia dla interpolowanych wektorów normalnych

Jak to działa? Rysunek 12.3 przedstawia przekrój przez płaską (trójkątną) ścianę bryły, której obraz ma (dzięki oświetleniu) wyglądać jak obraz bryły o powierzchni zakrzywionej. Wektory $\boldsymbol{n}_1$, $\boldsymbol{n}_2$ i $\boldsymbol{m}$ leżą w płaszczyźnie tego przekroju. Wektor $\boldsymbol{v}$ i (niepokazany) wektor $\boldsymbol{l}$, z którego kierunku dochodzi światło, mogą nie leżeć w tej płaszczyźnie, ale ich składowe prostopadłe do wektorów $\boldsymbol{m}$ i $\boldsymbol{n}$ nie mają wpływu na to, która strona powierzchni jest widoczna i która strona jest oświetlona.

Rozważmy dwa punkty widziane z kierunku wektora $\boldsymbol{v}$, zakładając dla ustalenia uwagi, że $\langle \boldsymbol{v}, \boldsymbol{m} \rangle > 0$,[5] przy czym kąt między wektorami $\boldsymbol{v}$ a $\boldsymbol{m}$ jest niewiele mniejszy niż kąt prosty. Kąt między otrzymanym w punkcie $\boldsymbol{p}_1$ z interpolacji wektorów normalnych powierzchni w wierzchołkach trójkąta wektorem $\boldsymbol{n}_1$ a wektorem $\boldsymbol{v}$ jest rozwarty ($\langle \boldsymbol{v}, \boldsymbol{n}_1 \rangle < 0$), co oznacza, że widoczna dla obserwatora strona powierzchni zakrzywionej jest przeciwna niż strona płaskiej ściany. Jeśli więc $\langle \boldsymbol{l}, \boldsymbol{n}_1 \rangle > 0$, to światło pada na punkt $\boldsymbol{p}_1$ po niewidocznej z kierunku $\boldsymbol{v}$ stronie powierzchni zakrzywionej. Ale punkt ten jest zasłonięty przez pewien znajdujący się bliżej obserwatora (oznaczony literą $\boldsymbol{q}$) punkt tej powierzchni, widoczny od strony, która w punkcie $\boldsymbol{p}_1$ jest oświetlona[6]. Wprawdzie szader wyprowadzi kolor obliczony przy użyciu wektora normalnego powierzchni w punkcie $\boldsymbol{p}_1$ (a nie $\boldsymbol{q}$, co byłoby właściwsze), ale ten błąd można uznać za akceptowalny. Jeśli $\langle \boldsymbol{l}, \boldsymbol{n}_1 \rangle < 0$, to obserwator widzi nieoświetloną stronę powierzchni, tak więc wynik badania widoczności oświetlonej strony powierzchni przez porównanie znaków liczb $\langle \boldsymbol{v}, \boldsymbol{m} \rangle$ i $\langle \boldsymbol{l}, \boldsymbol{n}_1 \rangle$ jest poprawny.

W punkcie $\boldsymbol{p}_2$ zarówno trójkąt, jak i powierzchnia zakrzywiona są z kierunku $\boldsymbol{v}$ widziane od zewnętrznej strony każdej z brył i rozważany tu sposób sprawdzania, czy strona ta jest oświetlona, zawsze daje poprawny wynik.

Podsumowując, dwa nowe programy składają się z czterech szaderów, a trzeci program ma ich pięć — wypełniają one wszystkie programowalne etapy potoku przetwarzania grafiki.

Pierwszy program składa się z szaderów przedstawionych na listingach 12.1, 12.2, 12.3 i 7.2. Jego przeznaczeniem jest wyświetlanie krawędzi dwudziestościanu, przerobionych na łuki okręgów.

[5] Jeśli $\langle \boldsymbol{v}, \boldsymbol{m} \rangle < 0$, to wystarczy zmienić w wyobraźni zwroty wektorów $\boldsymbol{m}$ i $\boldsymbol{n}$ na przeciwne.

[6] Ta strona w punkcie $\boldsymbol{q}$ *może* nie być oświetlona, bo wektory normalne powierzchni w punktach $\boldsymbol{p}_1$ i $\boldsymbol{q}$ mają różne kierunki. Uprzedzałem, że rozwiązanie jest niedoskonałe. Ale jest.

Drugi program jest zbudowany z szaderów pokazanych na listingach 12.1, 12.4, 12.5 i 7.2. On z kolei zamienia płaskie trójkątne ściany na trójkąty sferyczne. Wyświetlenie wszystkich ścian dwudziestościanu powoduje powstanie obrazu sfery, przy czym kolory pikseli na tym obrazie są obliczane przez interpolację kolorów wierzchołków ścian.

Trzeci program składa się z szaderów na listingach 12.1, 12.4, 12.6, 12.7 i 12.8. Szader fragmentów podstawia kolor fragmentu jako kolor farby pokrywającej powierzchnię do modelu oświetlenia lambertowskiego. Otrzymany przy jego użyciu obraz przestawia sferę oświetloną.

Uwaga: Można w programie użyć tylko szadera rozdrabniania, bez szadera sterującego rozdrabnianiem. Wyjście szadera wierzchołków jest wtedy kierowane na wejście szadera rozdrabniania. Tryb (izolinii, trójkątów, czworokątów lub punktów) jest podany w kwalifikatorze `layout` wejścia szadera rozdrabniania. Parametry, które do tablic `gl_TessLevelOuter` i `gl_TessLevelInner` wpisywałby szader sterowania rozdrabnianiem, podaje aplikacja, wykonując instrukcje

```
glPatchParameterfv ( GL_PATCH_DEFAULT_OUTER_LEVEL, otab );
glPatchParameterfv ( GL_PATCH_DEFAULT_INNER_LEVEL, itab );
```

W tablicach `otab` i `itab` należy podać odpowiednio cztery i dwie liczby typu `GLfloat`. Użycie szadera sterowania rozdrabnianiem umożliwia dostosowanie poziomu rozdrobnienia poszczególnych płatów do wielkości ich obrazów.

12.2. Aplikacja pierwsza D

Zmiany kodu aplikacji umożliwiające użycie szaderów opisanych wyżej są pokazane na listingach 12.9–12.13. Dotyczą one inicjalizacji programów szaderów, przygotowania obiektu do rysowania, odpowiedniej procedury go rysującej, interakcji z użytkownikiem, który może chcieć oglądać sferę lub dwudziestościan, i sprzątania.

Listing 12.9 przedstawia zmienione procedury przesyłania danych do bloku zmiennych jednolitych `TransBlock`, w którym jest dodane nowe pole `mmti`. Nazwa tego pola dopisana do tablicy `UTBNames` spowodowała przenumerowanie elementów (wydłużonej o 1) tablicy `trbofs`. W procedurze `LoadVPMatrix` wystarczyło tylko zmienić indeksy do tej tablicy. Natomiast procedura `LoadMMatrix` oprócz przesłania podanej jako parametr macierzy do pola `mm` wyznacza transpozycję odwrotności tej macierzy i przesyła ją do pola `mmti`. Zapewnia to spójność danych w bloku `TransBlock`.

Listing 12.9. Zmienione procedury przesyłania macierzy przekształceń

C

```
1: #define NTRUOFFS 6
2:
3: static const GLchar *UTBNames[] =
4:   { "TransBlock", "TransBlock.mm", "TransBlock.mmti", "TransBlock.vm",
5:     "TransBlock.pm", "TransBlock.vpm", "TransBlock.eyepos" };
6:
```

```
7: void LoadVPMatrix ( TransBl *trans )
8: {
9:   GLfloat vpm[16];
10:
11:   glBindBuffer ( GL_UNIFORM_BUFFER, trans->trbuf );
12:   M4x4Multf ( vpm, trans->pm, trans->vm );
13:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[2], 16*..., trans.vm );
14:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[3], 16*..., trans.pm );
15:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[4], 16*..., vpm )
16:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[5], 4*..., trans-eyepos );
17:   ExitIfGLError ( "LoadVPMatrix" );
18: } /*LoadPMatrix*/
19:
20: void LoadMMatrix ( TransBl *trans, GLfloat mm[16] )
21: {
22:   GLfloat mmti[16];
23:
24:   if ( mm ) memcpy ( trans->mm, mm, 16*sizeof(GLfloat) );
25:   glBindBuffer ( GL_UNIFORM_BUFFER, trans->trbuf );
26:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[0], 16*..., trans->mm );
27:   M4x4TInvertf ( mmti, trans->mm );
28:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1], 16*..., mmti );
29:   ExitIfGLError ( "LoadMMatrix" );
30: } /*LoadMMatrix*/
```

Procedura `LoadMyShaders` przedstawiona na listingu 12.10 czyta, kompiluje i łączy pięć programów szaderów — dwa używane przez aplikację pierwszą B i trzy nowe. Ponieważ niektóre szadery wchodzą w skład więcej niż jednego programu, wszystkie są kompilowane w pętli w liniach 20–21. Nazwy plików z szaderami są podane w tablicy `filename`; kolejno są to szadery z listingów 10.1, 10.2, 12.1, 12.2, 12.4, 12.3, 12.5, 12.6, 10.3, 12.7, 7.2, 10.4, 12.8, a typy tych szaderów są podane w tablicy `shtype`. We wszystkich szaderach, w których występuje blok zmiennych jednolitych `TransBlock`, jego definicja jest taka jak w tym rozdziale.

Listing 12.10. Procedura `LoadMyShaders`

C

```
1: GLuint program_id[5];
2:
3: void LoadMyShaders ( void )
4: {
5:   static const char *filename[13] =
6:     { "app1d0.vert.glsl", "app1d1.vert.glsl", "app1d2.vert.glsl",
7:       "app1d2.tesc.glsl", "app1d3.tesc.glsl",
8:       "app1d2.tese.glsl", "app1d3.tese.glsl", "app1d4.tese.glsl",
9:       "app1d1.geom.glsl", "app1d2.geom.glsl",
10:      "app1d0.frag.glsl", "app1d1.frag.glsl", "app1d3.frag.glsl" };
11:   static const GLuint shtype[12] =
```

```
12:     { GL_VERTEX_SHADER, GL_VERTEX_SHADER, GL_VERTEX_SHADER,
13:       GL_TESS_CONTROL_SHADER, GL_TESS_CONTROL_SHADER,
14:       GL_TESS_EVALUATION_SHADER, GL_TESS_EVALUATION_SHADER,
15:       GL_TESS_EVALUATION_SHADER, GL_GEOMETRY_SHADER, GL_GEOMETRY_SHADER,
16:       GL_FRAGMENT_SHADER, GL_FRAGMENT_SHADER, GL_FRAGMENT_SHADER };
17:   GLuint shader_id[13], sh[5];
18:   int    i;
19:
20:   for ( i = 0; i < 13; i++ )
21:     shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
22:   sh[0] = shader_id[0];  sh[1] = shader_id[10];
23:   program_id[0] = LinkShaderProgram ( 2, sh, "0" );
24:   sh[0] = shader_id[1];  sh[1] = shader_id[8];  sh[2] = shader_id[11];
25:   program_id[1] = LinkShaderProgram ( 3, sh, "1" );
26:   sh[0] = shader_id[2];  sh[1] = shader_id[3];  sh[2] = shader_id[5];
27:   sh[3] = shader_id[10];
28:   program_id[2] = LinkShaderProgram ( 4, sh, "2" );
29:   sh[0] = shader_id[2];  sh[1] = shader_id[4];  sh[2] = shader_id[6];
30:   sh[3] = shader_id[10];
31:   program_id[3] = LinkShaderProgram ( 4, sh, "3" );
32:   sh[0] = shader_id[2];  sh[1] = shader_id[4];  sh[2] = shader_id[7];
33:   sh[3] = shader_id[9];  sh[4] = shader_id[12];
34:   program_id[4] = LinkShaderProgram ( 5, sh, "4" );
35:   GetAccessToTransBlockUniform ( program_id[0] );
36:   GetAccessToLightBlockUniform ( program_id[1] );
37:   for ( i = 1; i <= 4; i++)
38:     AttachUniformTransBlockToBP ( program_id[i] );
39:   AttachUniformLightBlockToBP ( program_id[4] );
40:   for ( i = 0; i < 13; i++ )
41:     glDeleteShader ( shader_id[i] );
42:   ExitIfGLError ( "LoadMyShaders" );
43: } /*LoadMyShaders*/
```

W liniach 22–34 następuje łączenie programów szaderów; każde z pięciu wywołań procedury `LinkShaderProgram` jest poprzedzone wpisaniem do tablicy `sh` identyfikatorów szaderów składających się na dany program.

Dostęp aplikacji do bloków zmiennych jednolitych jest uzyskiwany przez wywołania procedur w liniach 35 i 36, po czym w liniach 37–39 dostęp do tych bloków uzyskują wszystkie korzystające z nich programy szaderów. Instrukcje, które tworzą bufory dla tych bloków są na listingu pominięte.

Zmiana procedury `ConstructIcosaheronVAO` pokazana na listingu 12.11 polega na *zastąpieniu* ostatnich 36 liczb w tablicy indeksów wierzchołków przez 60 nowych — dajemy po trzy indeksy na każdą trójkątną ścianę[7]. W liniach 16–17 tworzony jest odpowiednio wydłużony bufor, do którego jest przesyłany nowy ciąg liczb.

[7] Rezygnujemy z wachlarzy i taśm trójkątowych, bo i tak potrzebujemy każdą ścianę wyspecyfikować osobno, jako płat do rozdrobnienia.

Listing 12.11. Nowe procedury konstrukcji i rysowania dwudziestościanu

C

```
void ConstructIcosahedronVAO ( void )
{
  .... /* tablice vertpos i vertcol takie jak na listingu 7.5 */
  static const GLubyte vertind[96] =
     { 0, 1, 2, 0, 3, 4, 0, 5, 1, 9, 2, 8, 3, /* łamana, od 0 */
       .... /* początek tablicy vertind taki jak na listingu 7.5 */
       2, 3, 4, 5, 8, 9, 10, 11,              /* 4 odcinki, od 28 */
       0, 1, 2, 0, 2, 3, 0, 3, 4, 0, 4, 5,    /* trójkątne płaty, od 36 */
       0, 5, 1, 6, 7, 8, 6, 8, 9, 6, 9, 10,
       6, 10, 11, 6, 11, 7, 1, 9, 2, 9, 8, 2,
       2, 8, 3, 8, 7, 3, 3, 7, 4, 7, 11, 4,
       4, 11, 5, 11, 10, 5, 10, 1, 5, 10, 9, 1};

  .... /* tu wszystko tak samo, jak na listingu 7.5 */
  glBindBuffer ( GL_ELEMENT_ARRAY_BUFFER, icos_vbo[2] );
  glBufferData ( GL_ELEMENT_ARRAY_BUFFER,
                 96*sizeof(GLubyte), vertind, GL_STATIC_DRAW );
  glBindVertexArray ( 0 );
  ExitIfGLError ( "ConstructIcosahedronVAO" );
} /*ConstructIcosahedronVAO*/

void DrawIcosahedron ( int opt, char enlight )
{
  glBindVertexArray ( icos_vao );
  switch ( opt ) {
    .... /* rysowanie wierzchołków i krawędzi bez zmian */
default:   /* ściany */
    glUseProgram ( program_id[enlight ? 1 : 0] );
    glDrawElements ( GL_TRIANGLES, 60,
                     GL_UNSIGNED_BYTE, (GLvoid*)(36*sizeof(GLubyte)) );
    break;
  }
  glBindVertexArray ( 0 );
} /*DrawIcosahedron*/

void DrawTessIcos ( int opt, char enlight )
{
  glBindVertexArray ( icos_vao );
  switch ( opt ) {
case 0:
    break;
case 1:
    glUseProgram ( program_id[2] );
    glPatchParameteri ( GL_PATCH_VERTICES, 2 );
    glDrawElements ( GL_PATCHES, 24, GL_UNSIGNED_BYTE, (GLvoid*)0 );
```

```
46:     glDrawElements ( GL_PATCHES, 26, GL_UNSIGNED_BYTE,
47:                      (GLvoid*)(1*sizeof(GLubyte)) );
48:     glDrawElements ( GL_PATCHES, 10, GL_UNSIGNED_BYTE,
49:                      (GLvoid*)(26*sizeof(GLubyte)) );
50:     break;
51: default:
52:     glUseProgram ( program_id[enlight ? 4 : 3] );
53:     glPatchParameteri ( GL_PATCH_VERTICES, 3 );
54:     glDrawElements ( GL_PATCHES, 60, GL_UNSIGNED_BYTE,
55:                      (GLvoid*)(36*sizeof(GLubyte)) );
56:     break;
57:   }
58:   glBindVertexArray ( 0 );
59:   ExitIfGLError ( "DrawTessIcos" );
60: } /*DrawTessIcos*/
```

Podczas rysowania płatów trójkątnych każda trójka wierzchołków jest traktowana osobno — płatów nie łączy się w wachlarze ani taśmy. W związku ze zmianą tablicy indeksów procedura `DrawIcosahedron` została zmieniona tak, aby ściany dwudziestościanu były rysowane jako osobne trójkąty.

Nowa procedura `DrawTessIcos` wyświetla płaty — odcinki albo trójkąty, zależnie od wartości parametru `opt`. Jeśli jest równa 1, to rysowanych jest 30 płatów, z których każdy odpowiada jednej krawędzi dwudziestościanu. Do rysowania jest używany pierwszy z nowych programów o identyfikatorze zapamiętanym w `program_id[2]`. Procedura `glPatchParameteri` w linii 44 zgłasza po dwa wierzchołki na płat. Następnie są trzy wywołania procedury `glDrawElements`. Został tu użyty pewien trik. Dane w tablicy indeksów wierzchołków opisują długą łamaną (o 25 wierzchołkach, czyli o 24 odcinkach), po niej krótką łamaną (o 3 wierzchołkach) i 4 osobne odcinki. Procedura wywołana w linii 45 rysuje co drugi odcinek długiej łamanej. Następne wywołanie rysuje pozostałe odcinki długiej łamanej i pierwszy odcinek łamanej krótkiej, a w liniach 48–49 jest rysowany drugi odcinek tej łamanej i cztery osobne odcinki.

Jeśli parametr `opt` ma wartość 2, to procedura `DrawTessIcos` rysuje płaty trójkątne. Indeksy ich wierzchołków to 60 liczb dodanych na końcu tablicy (w liniach 8–12 na listingu). Zależnie od wartości parametru `enlight` jest wybierany drugi lub trzeci z programów opisanych w tym rozdziale (ich identyfikatory są zapamiętane w zmiennych `program_id[3]` i `program_id[4]`). W linii 53 OpenGL jest informowany o tym, że kolejne płaty mają po 3 wierzchołki.

Uwaga: Choć szadery sterowania rozdrabnianiem mają (w kwalifikatorze wyjścia, zobacz listingi 12.2 i 12.4) zadeklarowane liczby wierzchołków opisujących każdy płat, nie zwalnia to aplikacji z obowiązku wywołania przed rysowaniem płatów procedury `glPatchParameteri` z odpowiednią liczbą podaną jako parametr.

Procedura `RedrawMyWorld` na listingu 12.12 zależnie od wartości zmiennej `tesselate` wywołuje procedurę rysującą dwudziestościan albo sferę (otrzymaną przez rozdrobnienie

płatów). Procedura `KeyboardFunc`, w dodatku do reakcji na dotychczas obsługiwane klawisze, po napisaniu litery `T` albo `t` zmienia wartość tej zmiennej i każe wykonać nowy obraz.

Listing 12.12. Procedury `RedrawMyWorld` i `ProcessCharCommand`

C

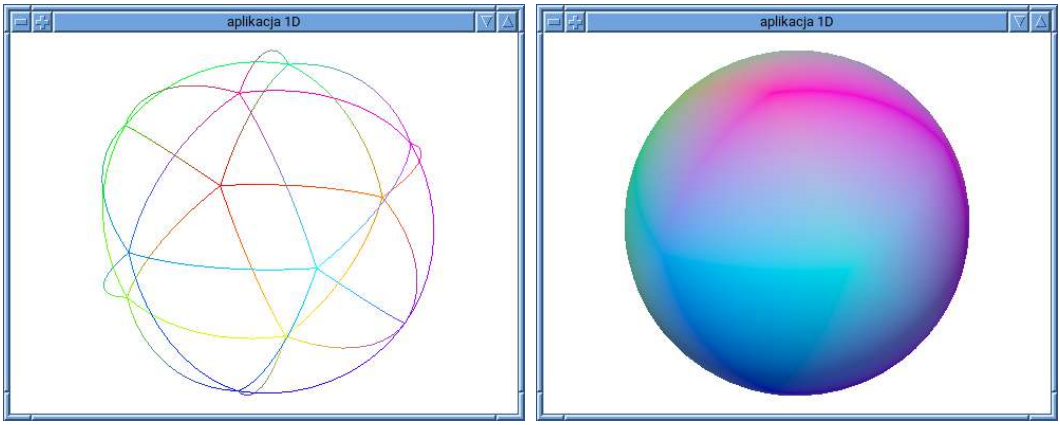
```
char tesselate = 0;

void RedrawMyWorld ( void )
{
  .... /* tu niezmienione linie 44-52 z listingu 11.10 */
    glEnable ( GL_DEPTH_TEST );
    if ( tesselate )
      DrawTessIcos ( option, enlight );
    else
      DrawIcosahedron ( option, enlight );
    glUseProgram ( 0 );
    glFlush ();
  }
} /*RedrawMyWorld*/

void ProcessCharCommand ( char charcode )
{
  int oldoption;

  oldoption = option;
  switch ( toupper ( charcode ) ) {
case 'T':
    tesselate = !tesselate;
    return true;
    .... /* reakcje na pozostałe litery bez zmian */
  }
  return option != oldoption;
} /*ProcessCharCommand*/
```

Listing 12.13. Procedura sprzątania

C

```
void DeleteMyWorld ( void )
{
  int i;

  glUseProgram ( 0 );
  for ( i = 0; i < 5; i++ )
    glDeleteProgram ( program_id[i] );
  glDeleteBuffers ( 1, &trans.trbuf );
  .... /* reszta sprzątania bez zmian */
} /*DeleteMyWorld*/
```

Procedura sprzątania w dwóch pętlach likwiduje wszystkie programy, a następnie zwalnia pozostałe zasoby zarezerwowane przez aplikację. No i tyle.



Rysunek 12.4. Okno aplikacji pierwszej D

12.3. Ćwiczenia

1. Prześledź działanie szadera z listingu 12.8 i analizę przeprowadzoną na stronie 287. Zbadaj, kiedy przybliżana przez trójkąt powierzchnia zakrzywiona jest w punktach $\boldsymbol{p}_1$ i $\boldsymbol{q}$ oświetlona po przeciwnych stronach i jaki to ma wpływ na otrzymane obrazy.

2. Wprowadź przekształcenie obiektu będące złożeniem skalowania nierównomiernego z obrotem. Zmodyfikuj odpowiednio instrukcję w linii 8 na listingu 12.12 i wstaw przed nią instrukcję obliczającą transpozycję odwrotności macierzy przekształcenia modelu. Otrzymasz program wykonujący obrazy elipsoidy.

3. Napisz szader fragmentów, który w zależności od wartości odpowiedniej zmiennej jednolitej (zmienianej po naciśnięciu jakiegoś klawisza) przyjmuje w obliczeniach oświetlenia wektor normalny sfery (obliczony przez szader rozdrabniania) albo wektor normalny płaszczyzny trójkąta (obliczony przez szader geometrii). Wypróbuj go w działaniu.

4. Wydłuż tablicę wierzchołków dwudziestościanu, dopisując do niej wierzchołki w połowie każdej krawędzi i w środkach ciężkości wszystkich ścian. Korzystając z nowych wierzchołków, podziel każdą ścianę na 6 trójkątów prostokątnych i utwórz nową tablicę trójek indeksów — ma w niej znaleźć się 180 liczb, będących numerami wierzchołków 60 trójkątów (trzech trójkątów otrzymanych z podziału każdej ściany dwudziestościanu). Wyświetl te trójkąty jako płaty, zamieniając je na trójkąty sferyczne.

5. Wprowadź przekształcenie nieliniowe obiektu, przez modyfikację szaderów z listingów 12.5 i 12.6. Po znalezieniu wierzchołka na sferze jednostkowej, a *przed* poddaniem go

Rysunek 12.5. Rozwiązania ćwiczeń 4 i 5

dalszym przekształceniom, pomnóż współrzędne x i y przez 0.75, a współrzędną z przez $e^{z/10}$ (możesz użyć funkcji `exp`).

W obliczeniach wektora normalnego przekształcanej powierzchni przesunięcie jest nieistotne, skupmy się zatem na macierzach 3×3 opisujących część liniową i „część liniową" użytych przekształceń. W naszym przypadku przekształcenie obiektu jest opisane przez iloczyn macierzy MS, gdzie M jest macierzą obrotu modelu (to jej współczynniki są w polu `TransBlock.mm`), a $S = S(x, y, z)$ jest macierzą zmieniającą się ze współrzędnymi x, y, z przekształcanego punktu (co wprowadza nieliniowość). Dla ustalonych liczb x, y, z macierz S jest macierzą skalowania osi (o współczynnikach $\frac{3}{4}, \frac{3}{4}, e^{z/10}$). Równanie płaszczyzny stycznej do powierzchni będącej obrazem w przekształceniu f płaszczyzny o wektorze normalnym $\boldsymbol{n}$ opisanemu przez te macierze dla ustalonych liczb x, y, z otrzymamy tak:

$$0 = \boldsymbol{n}^T(\boldsymbol{p} - \boldsymbol{p}_0) = \boldsymbol{n}^T S^{-1} M^{-1} MS(\boldsymbol{p} - \boldsymbol{p}_0) = (M^{-T} S^{-T} \boldsymbol{n})^T \big(f(\boldsymbol{p}) - f(\boldsymbol{p}_0)\big).$$

Stąd wynika[8], że jeśli $\boldsymbol{n}$ oznacza wektor normalny sfery jednostkowej w rozpatrywanym (przetwarzanym przez szader) punkcie, to wektor normalny płaszczyzny stycznej do otrzymanej skorupki jest równy $M^{-T} S^{-T} \boldsymbol{n}$. Mamy

$$S = S^T = \begin{bmatrix} \frac{3}{4} & 0 & 0 \\ 0 & \frac{3}{4} & 0 \\ 0 & 0 & e^{z/10} \end{bmatrix}, \quad S^{-1} = S^{-T} = \begin{bmatrix} \frac{4}{3} & 0 & 0 \\ 0 & \frac{4}{3} & 0 \\ 0 & 0 & e^{-z/10} \end{bmatrix}.$$

Ćwiczenie polega m.in. na dodaniu do szadera z listingu 12.6 instrukcji obliczających wektor normalny zgodnie z tym rachunkiem i uruchomieniu aplikacji z tym szaderem.

[8] Ten rachunek *nie jest* pełnym dowodem, ale to jest kurs OpenGL-a, nie Analizy II.

12.4. *Uzupełnienia[9]

12.4.1. Rozdrabnianie nierównomierne

Zamiast kwalifikatora układu wejścia szadera rozdrabniania `equal_spacing` można podać kwalifikator `fractional_even_spacing` albo `fractional_odd_spacing`. Skutkiem podania tych kwalifikatorów będzie podzielenie krawędzi dziedziny płata na odpowiednio parzystą (*even*) albo nieparzystą (*odd*) liczbę odcinków o jednakowej długości i dodatkowe dwa krótsze odcinki na końcach. Elementy tablic `gl_TessLevelOuter` (i `gl_TessLevelInner`), którym szader sterowania rozdrabnianiem przypisuje wartości, są typu `float`; przypisane wartości mogą być ułamkowe. Reguła podziału krawędzi o długości l, gdy odpowiedni element tablicy `gl_TessLevelOuter` ma wartość d, jest następująca: niech n oznacza największą liczbę parzystą albo nieparzystą mniejszą niż d. Wtedy „wewnętrzne" odcinki podzielonej krawędzi mają długość l/d, a długość odcinków skrajnych jest równa $l(1-n/d)/2$.

Możliwość dokonywania takiego podziału bywa użyteczna, gdy szader sterowania rozdrabnianiem dostosowuje podział płata do potrzebnego poziomu szczegółowości (*level of detail*) rysowanych obiektów. Jeśli animacja wiąże się ze stopniowym zmienianiem wielkości obrazu obiektu, a krawędzie są dzielone zawsze na odcinki o tej samej długości, to zmiana liczby tych odcinków powoduje skokową zmianę jakości obrazu. Opisane tu kwalifikatory umożliwiają znaczne osłabienie tego zjawiska.

12.4.2. Restart prymitywu

W tablicy indeksów wierzchołków przywiązanej do celu `GL_ELEMENT_ARRAY_BUFFER` może występować specjalny indeks, tj. liczba r niebędąca indeksem wierzchołka, na przykład 255, jeśli wszystkich wierzchołków jest mniej niż 255 i liczby w tablicy są ośmiobitowe. Liczba ta służy do **restartu prymitywu**. Przed rysowaniem należy poinformować OpenGL-a, jaka to liczba, podając ją jako parametr procedury `glPrimitiveRestartIndex`, oraz włączyć restart za pomocą instrukcji

```
glEnable ( GL_PRIMITIVE_RESTART );
```

Odcinki łamanej lub trójkąty, których jeden z wierzchołków ma numer r, podczas rysowania łamanej (`GL_LINE_STRIP`), taśmy trójkątowej (`GL_TRIANGLE_STRIP`) lub wachlarza trójkątów (`GL_TRIANGLE_FAN`) za pomocą procedury `glDrawElements` zostaną pominięte. Dzięki temu można narysować *wiele* łamanych, taśm trójkątowych lub wachlarzy za pomocą *jednego* wywołania procedury `glDrawElements`. Motywacją do takiego postępowania jest ograniczanie komunikacji między CPU a GPU.

Przed rysowaniem następnych obiektów trzeba pamiętać o wyłączeniu restartu prymitywu lub uaktualnieniu informacji o specjalnym indeksie.

Ćwiczenie: Zmodyfikuj aplikację tak, aby rysowała cały dwudziestościan foremny za pomocą jednego wywołania procedury `glDrawElements`. W podobny sposób rysuj sześcian,

[9] Zobacz pierwsze zdanie podrozdziału 4.5.

ośmiościan i dwunastościan foremny. **Wskazówka:** Dla dwudziestościanu utwórz 4 taśmy trójkątowe składające się odpowiednio z 8, 8, 2 i 2 trójkątów.

12.4.3. Prymitywy z przyległościami

Pierwszy parametr procedur z rodziny `glDraw*` (np. `glDrawArrays` lub `glDrawElements`) może być jedną ze wspomnianych w podrozdziale 7.6 stałych symbolicznych `GL_LINES_ADJACENCY`, `GL_LINE_STRIP_ADJACENCY`, `GL_TRIANGLES_ADJACENCY`, `GL_TRIANGLE_STRIP_ADJACENCY`. Pierwsze dwie stałe powodują wprowadzenie do potoku przetwarzania grafiki **odcinków z przyległościami**, a pozostałe dwie wprowadzają **trójkąty z przyległościami**. Wyjaśnijmy, na czym to polega.

Odcinek jest określony przez swoje dwa końce. Jego przyległości to dwa dodatkowe punkty, z których jeden „poprzedza" pierwszy koniec, a drugi „następuje po" drugim końcu. Zatem odcinek z przyległościami jest reprezentowany przez cztery punkty, które (po przejściu przez szader wierzchołków) trafiają na wejście szadera geometrii[10]. Szader ten powinien mieć kwalifikator wejścia

```
layout(lines_adjacency) in;
```

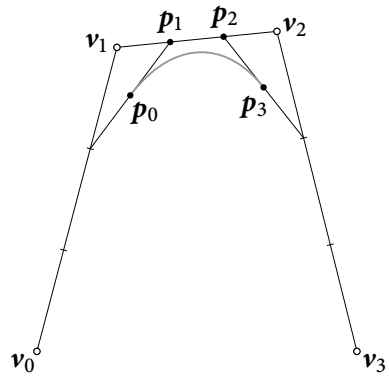
Tablice wejściowe takiego szadera (tablica `gl_in` i tablice dodatkowych atrybutów wyprowadzonych przez szader wierzchołków) mają długość 4 zamiast 2. „Właściwe" końce odcinka to punkty reprezentowane przez elementy 1 i 2 tych tablic, a elementy 0 i 3 opisują przyległości. Można to widzieć też tak, że szader geometrii otrzymuje na wejściu łamaną złożoną z trzech odcinków.

Najprostsze zastosowanie odcinka z przyległościami polega na narysowaniu krzywej wielomianowej trzeciego stopnia. Łamana otrzymana przez szader geometrii może być zinterpretowana jako reprezentacja Béziera takiej krzywej[11]. Inna możliwość to przyjęcie, że jest to reprezentacja B-sklejana, na podstawie której można znaleźć reprezentację Béziera tej krzywej. Odpowiednią konstrukcję (i wzory) dla krzywej z węzłami równoodległymi przedstawia rysunek 12.6.

Mając reprezentację Béziera, szader geometrii może obliczyć pewną liczbę punktów krzywej i wyprowadzić łamaną o wierzchołkach w tych punktach. Najprostszy taki szader (który nie przetwarza żadnych dodatkowych atrybutów wierzchołków) jest pokazany na listingu 12.14. Szader ten w liniach 19–23 znajduje reprezentację Béziera krzywej, a następnie w pętli oblicza punkty tej krzywej, tj. wierzchołki łamanej wyjściowej, i wyprowadza je do etapu obcinania. Punkty krzywej są obliczane metodą opisaną w podrozdziale 15.1. Podprogram `BCHorner4f` nie musi tu obliczać wektora pochodnej parametryzacji, zatem można go otrzymać z procedury `BCHorner2f` z listingu 15.4 przez zmianę nazwy i zmianę wszystkich wystąpień słowa kluczowego `vec2` na `vec4`.

[10] Program szaderów używany podczas rysowania w trybach z przyległościami nie powinien zawierać szaderów rozdrabniania, bo one mają zastosowanie do rozdrabniania *płatów* rysowanych w trybie `GL_PATCHES`. Jeśli szader geometrii jest nieobecny, to przyległości są pomijane i do etapu obcinania trafia tylko „właściwy" odcinek albo trójkąt.

[11] Podstawowe wiadomości o krzywych Béziera i B-sklejanych podałem dalej, w rozdziale 15 i w dodatku B, a szczegółowe ich opisy można znaleźć w książce [41].



$$p_0 = \frac{1}{6}v_0 + \frac{2}{3}v_1 + \frac{1}{6}v_2$$
$$p_1 = \frac{2}{3}v_1 + \frac{1}{3}v_2$$
$$p_2 = \frac{1}{3}v_1 + \frac{2}{3}v_2$$
$$p_3 = \frac{1}{6}v_1 + \frac{2}{3}v_2 + \frac{1}{6}v_3$$

Rysunek 12.6. Konstrukcja punktów kontrolnych reprezentacji Béziera krzywej

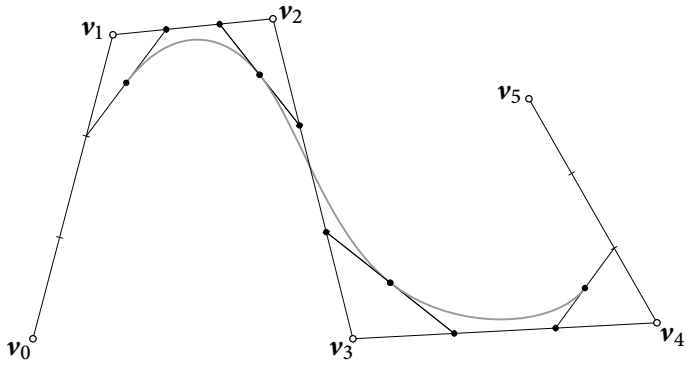
Listing 12.14. Szader geometrii zamieniający odcinek z przyległościami na krzywą

GLSL

```
#version 450 core

#define MAX_DEG  3
#define NN      21

layout(lines_adjacency) in;
layout(line_strip,max_vertices=NN) out;

void BCHorner4f ( int n, vec4 bcp[MAX_DEG+1], float t, out vec4 p )
{
  .... /* zobacz listing 15.4 */
} /*BCHorner4f*/

void main ( void )
{
  vec4 cp[MAX_DEG+1];
  int i;

  cp[0] = mix ( gl_in[0].gl_Position, gl_in[1].gl_Position, 2.0/3.0 );
  cp[1] = mix ( gl_in[1].gl_Position, gl_in[2].gl_Position, 1.0/3.0 );
  cp[2] = mix ( gl_in[1].gl_Position, gl_in[2].gl_Position, 2.0/3.0 );
  cp[3] = mix ( gl_in[2].gl_Position, gl_in[3].gl_Position, 1.0/3.0 );
  cp[0] = mix ( cp[0], cp[1], 0.5 );  cp[3] = mix ( cp[2], cp[3], 0.5 );
  for ( i = 0; i < NN; i++ ) {
    BCHorner4f ( 3, cp, float(i)/float(NN-1), gl_Position );
    EmitVertex ();
  }
  EndPrimitive ();
} /*main*/
```

Uwaga: Przejście od układu współrzędnych modelu do układu kostki standardowej może wykonać szader wierzchołków; krzywa (wielomianowa lub wymierna), zarówno w rzucie równoległym, jak i perspektywicznym, zostanie narysowana poprawnie.

Wywołanie procedury `glDraw*` z parametrem `GL_LINES_ADJACENCY` spowoduje wprowadzenie osobnego odcinka z przyległościami dla kolejnych *rozłącznych* czwórek wierzchołków w tablicy (VAO) przywiązanej za pomocą `glBindVertexArray`. Jeśli wtedy podana jako parametr liczba wierzchołków jest równa n, to do potoku przetwarzania grafiki zostanie wprowadzonych $\lfloor n/4 \rfloor$ odcinków. Jeśli pierwszy parametr procedury `glDraw*` jest równy `GL_LINE_STRIP_ADJACENCY`, to odcinek z przyległościami zostanie wprowadzony dla każdej czwórki *kolejnych* wierzchołków, co poskutkuje wprowadzeniem $n - 3$ odcinków.



Rysunek 12.7. Łamana kontrolna, krzywa B-sklejana i jej części w reprezentacji Béziera

Rysunek 12.7 przedstawia krzywą B-sklejaną, której obraz można wykonać w opisany tu sposób w drugim z przedstawionych wyżej trybów rysowania. Zaletą tej reprezentacji krzywych jest otrzymywanie gładkich połączeń łuków wielomianowych — parametryzacja krzywej B-sklejanej trzeciego stopnia z węzłami jednokrotnymi (w tym równoodległymi) ma ciągłe pochodne pierwszego i drugiego rzędu.

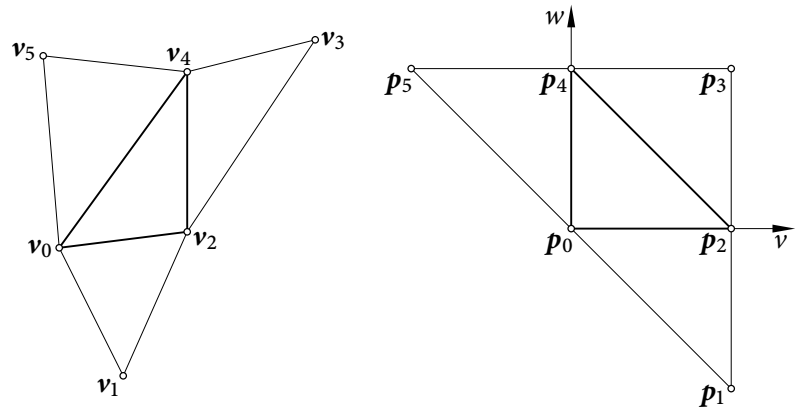
Przyległości trójkąta to trzy trójkąty „doczepione" do jego boków. Trójkąt z przyległościami jest więc reprezentowany przez 6 wierzchołków i taką długość mają tablice wejściowe szadera geometrii, w którym występuje kwalifikator

```
layout(triangles_adjacency) in;
```

Kolejność wierzchołków reprezentujących trójkąt z przyległościami w tych tablicach jest pokazana na rysunku 12.8.

Wielomian drugiego stopnia dwóch zmiennych jest jednoznacznie określony przez swoje wartości w sześciu punktach nieleżących na krzywej drugiego stopnia[12], na przykład w wierzchołkach i środkach boków dowolnego trójkąta. Szader geometrii może z jednego trójkąta

[12] czyli na elipsie, paraboli, hiperboli lub dwóch prostych



Rysunek 12.8. Wierzchołki trójkąta z przyległościami i odpowiadające im punkty na płaszczyźnie

wyprodukować wiele trójkątów (w sposób opisany w podrozdz. E.4), przy czym zarówno współrzędne wierzchołków tych trójkątów, jak i inne ich atrybuty mogą być wartościami wielomianów drugiego stopnia określonych przez dane wierzchołki trójkąta z przyległościami.

Niech f oznacza dowolny wielomian drugiego stopnia określony na płaszczyźnie (tj. wielomian dwóch zmiennych). Możemy go przedstawić jako funkcję zmiennych u, v, w, będących współrzędnymi barycentrycznymi (zobacz podrozdz. 5.3) w układzie określonym przez wierzchołki trójkąta $\boldsymbol{p}_0 = (1,0,0)$, $\boldsymbol{p}_2 = (0,1,0)$ i $\boldsymbol{p}_4 = (0,0,1)$. Wierzchołkom przyległości przyporządkujemy punkty $\boldsymbol{p}_1 = (1,1,-1)$, $\boldsymbol{p}_3 = (-1,1,1)$ i $\boldsymbol{p}_5 = (1,-1,1)$. Jeśli $f_i = f(\boldsymbol{p}_i)$ dla $i = 0,\ldots,5$, to

$$f(u,v,w) = (vw+u)f_0 + uvf_1 + (wu+v)f_2 + vwf_3 + (uv+w)f_4 + wuf_5.$$

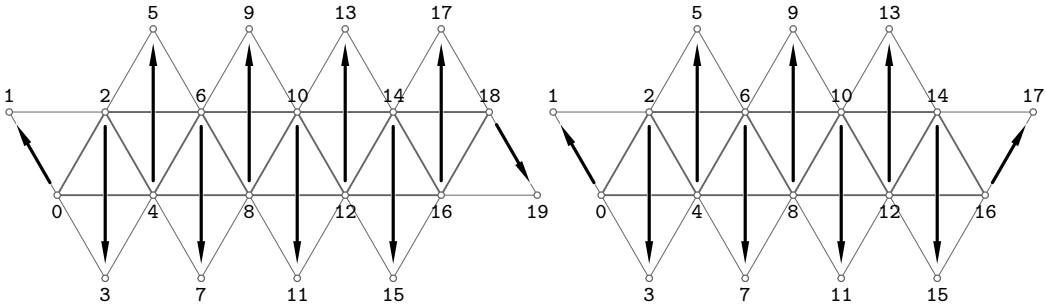
Szader geometrii lub fragmentów może użyć tego wzoru do interpolowania dowolnych atrybutów wierzchołków trójkąta z przyległościami.

Jeśli chcemy zapewnić ciągłość funkcji opisujących atrybuty punktów na wspólnych bokach trójkątów, to trzeba przyjąć, że „właściwymi" wierzchołkami trójkąta są przyległości, tj. punkty $\boldsymbol{v}_1$, $\boldsymbol{v}_3$ i $\boldsymbol{v}_5$; wierzchołkom $\boldsymbol{v}_0$, $\boldsymbol{v}_2$ i $\boldsymbol{v}_4$ odpowiadają środki boków takiego trójkąta. Możemy umieścić wszelkie atrybuty wierzchołków i środków boków trójkątów w tablicach zarejestrowanych w VAO, zapisać odpowiedni ciąg liczb w tablicy indeksów w buforze przywiązanym do celu `GL_ELEMENT_ARRAY_BUFFER` i wywołać procedurę `glDrawElements` z parametrem `GL_TRIANGLES_ADJACENCY`. Jeśli drugi parametr ma wartość n, to do potoku przetwarzania grafiki trafi $\lfloor n/6 \rfloor$ trójkątów z przyległościami, których dalszym przetwarzaniem zajmą się szadery.

Pozostał do omówienia tryb `GL_TRIANGLE_STRIP_ADJACENCY`. Wskutek jego zastosowania do potoku trafi taśma zawierająca trójkąty z przyległościami, wyznaczone przez nakładające się szóstki wierzchołków w tablicach. Z takiej taśmy zostaną wybrane i przekazane szaderowi geometrii pojedyncze trójkąty z przyległościami reprezentowane w taki sam sposób jak w trybie `GL_TRIANGLES_ADJACENCY`.

Zasady budowy taśmy trójkątowej z przyległościami można znaleźć w specyfikacji [1], w różnych książkach i na licznych stronach internetowych, ale znacznie trudniej jest znaleźć zastosowania tego trybu. Rzecz w tym, że dane opisujące trójkąty z przyległościami nie wystarczają do określenia funkcji interpolacyjnej stopnia większego niż 1 w sposób gwarantujący ciągłość na wspólnym boku „właściwych" trójkątów z taśmy[13]. Jeśli taśma ta jest przybliżeniem fragmentu powierzchni gładkiej, to aby obraz wyglądał jak obraz powierzchni gładkiej, szader geometrii powinien obliczyć wektor normalny dla każdego wierzchołka tak, aby w sąsiednim trójkącie, dla tego samego wierzchołka, otrzymać ten sam wektor normalny. Niestety położenia wierzchołków trójkąta z przyległościami nie wystarczają do rozwiązania tego zadania[14].

Przyjrzyjmy się budowie taśmy, a potem zobaczymy, do czego ona może się przydać. Jeśli podana liczba n wierzchołków taśmy jest nieparzysta, to ostatni wierzchołek jest ignorowany, założymy więc dalej, że n jest liczbą parzystą. Wtedy liczba „właściwych" trójkątów w taśmie jest równa $(n-4)/2$ — dla tylu wziętych z taśmy trójkątów z przyległościami zostanie wywołany szader geometrii.



Rysunek 12.9. Kolejność wierzchołków w taśmie trójkątowej z przyległościami

Wierzchołki „właściwych" trójkątów w taśmie mają indeksy podane na miejscach parzystych, tj. $0, 2, 4$ itd., a pozostałe indeksy identyfikują wierzchołki trójkątów będących *tylko* przyległościami[15]. Pomijając wierzchołki o numerach nieparzystych, otrzymalibyśmy zwykłą taśmę trójkątową, do narysowania w trybie `GL_TRIANGLE_STRIP`; aby zatem określić taśmę z przyległościami, trzeba po każdym wierzchołku zwykłej taśmy podać jeszcze jeden wierzchołek. Pokazuje to rysunek 12.9 przedstawiający dwie taśmy, zawierające odpowiednio parzystą i nieparzystą liczbę „właściwych" trójkątów. Kolejność wierzchołków w obu przypadkach jest taka jak kolejność strzałek na rysunku, od jego lewej do prawej strony.

Najczęściej tryb `GL_TRIANGLE_STRIP_ADJACENCY` jest wykorzystywany z użyciem bufora indeksów do tablicy wierzchołków (tj. rysowanie odbywa się przy użyciu procedury

[13] Zauważmy, że szader geometrii nie może „zajrzeć" do danych opisujących przyległości sąsiednich trójkątów w taśmie, zresztą przyległości te mogą nie istnieć.

[14] W rozdziale 33 zrealizujemy obliczanie wektora normalnego w wierzchołkach siatki nieregularnej — przez szader obliczeniowy mający dostęp do całej siatki.

[15] „Właściwe" trójkąty też są przyległościami innych „właściwych" trójkątów w taśmie, mających z nimi wspólne boki.

`glDrawElements`). Bardzo często wierzchołki przechowywane w tablicy są punktami gładkiej powierzchni, tworzącymi regularną siatkę z kolumnami i wierszami. Wtedy dla każdej pary sąsiednich wierszy (lub kolumn) może zostać utworzona taśma trójkątowa, dla której przyległości zostaną znalezione w wierszach (lub kolumnach) sąsiadujących z tymi z pary. Potrzebna jest zatem procedura wpisująca do tablicy indeksy wierzchołków, które trzeba będzie przesłać do bufora indeksów w pamięci GPU.

Listing 12.15 przedstawia dwie przykładowe procedury wykonujące tę czynność; obie wygenerują ciągi indeksów dla taśm składających się z parzystych liczb trójkątów, których wierzchołki są podane w tablicy o długości MN; wierzchołki te tworzą regularną siatkę mającą M wierszy i N kolumn — wierzchołki w tablicy są przechowywane kolumna po kolumnie, bez przerw.

Makrodefinicja `RESTART_IND` określa indeks restartu prymitywu (zobacz p. 12.4.2), wstawiany między ciągi indeksów wierzchołków poszczególnych taśm; dzięki niemu możliwe jest wyświetlenie wszystkich taśm za pomocą jednego wywołania procedury `glDrawElements`. Założenie, że liczba wierzchołków w tablicy będzie mniejsza niż 2^{16}, było podstawą do przyjęcia 16-bitowej reprezentacji indeksów, czyli typu `GLushort`.

Makrodefinicja `IND` oblicza na podstawie parametrów — numerów kolumny i wiersza — indeks odpowiedniego wierzchołka w tablicy wierzchołków. Makrodefinicja `ENTER` dopisuje indeks wierzchołka w podanym miejscu siatki na końcu ciągu w tablicy, której końcową zawartość trzeba będzie przesłać do bufora indeksów w pamięci GPU.

Pierwsza z pokazanych procedur działa przy założeniu, że pierwszy i ostatni wiersz oraz pierwsza i ostatnia kolumna stanowią „obwódkę" dla zbioru „właściwych" danych w tablicy. Procedura wpisuje do tablicy `indexb` dane opisujące $M-3$ taśmy, z których każda zawiera $2(N-3)$ „właściwe" trójkąty. Całkowita liczba wpisanych indeksów (i wartość powrotna procedury, którą trzeba zapamiętać, aby później podawać ją jako trzeci parametr procedury `glDrawElements`) jest równa $(M-3)(4N-7)-1$.

Listing 12.15. Procedury przygotowujące indeksy dla taśm trójkątowych z przyległościami

C

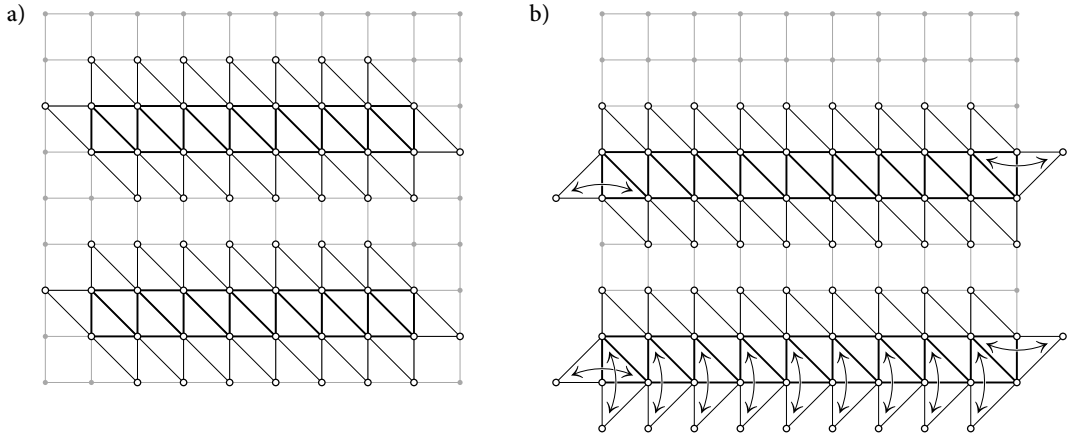
```
#define RESTART_IND 65535
#define IND(a,b) ((a)*M+(b))
#define ENTER(a,b) indexb[k++] = IND(a,b);

int GenTriangleStripAdjIndices1 ( int N, int M, GLushort *indexb )
{
  int i, j, k;

  k = 0;
  for ( j = 1; j < M-2; j++ ) {
    ENTER(1,j)  ENTER(0,j+1)
    for ( i = 2; i < N-1; i++ )
      { ENTER(i-1,j+1)  ENTER(i,j-1)  ENTER(i,j)  ENTER(i-1,j+2) }
    ENTER(N-2,j+1)  ENTER(N-1,j)
    if ( j < M-3 )
```

```
16:       indexb[k++] = RESTART_IND;
17:   }
18:   return k;
19: } /*GenTriangleStripAdjIndices1*/
20:
21: int GenTriangleStripAdjIndices2 ( int N, int M, GLushort *indexb )
22: {
23:   int i, j, k;
24:
25:   k = 0;
26:   ENTER(0,0)  ENTER(1,0)
27:   for ( i = 1; i < N; i++ )
28:     { ENTER(i-1,1)  ENTER(i-1,1)  ENTER(i,0)  ENTER(i-1,2) }
29:   ENTER(N-1,1)  ENTER(N-2,1)
30:   indexb[k++] = RESTART_IND;
31:   for ( j = 1; j < M-2; j++ ) {
32:     ENTER(0,j)  ENTER(1,j)
33:     for ( i = 1; i < N; i++ )
34:       { ENTER(i-1,j+1)  ENTER(i,j-1)  ENTER(i,j)  ENTER(i-1,j+2) }
35:     ENTER(N-1,j+1)  ENTER(N-2,j+1)
36:     indexb[k++] = RESTART_IND;
37:   }
38:   ENTER(0,M-2)  ENTER(1,M-2)
39:   for ( i = 1; i < N; i++ )
40:     { ENTER(i-1,M-1)  ENTER(i,M-3)  ENTER(i,M-2)  ENTER(i,M-2) }
41:   ENTER(N-1,M-1)  ENTER(N-2,M-1)
42:   return k;
43: } /*GenTriangleStripAdjIndices2*/
```

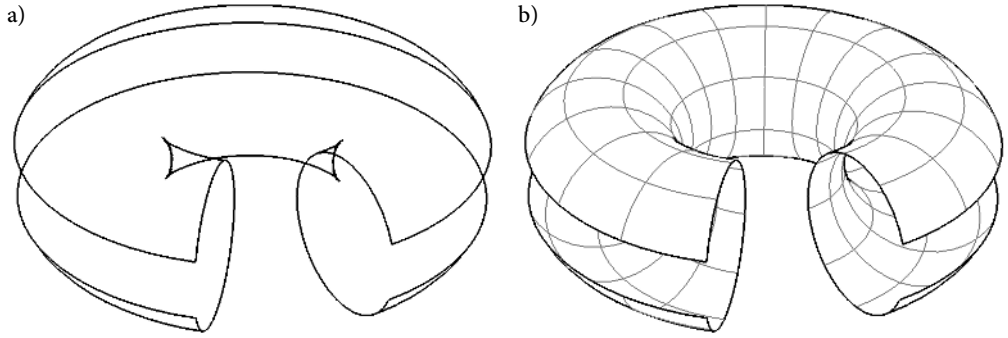
Druga procedura zakłada, że w tablicy wierzchołków nie ma obwódki, zatem każdemu czworokątowi o wierzchołkach na przecięciu pary sąsiednich kolumn z parą sąsiednich wierszy mają odpowiadać dwa „właściwe" trójkąty do umieszczenia w taśmie. Procedura wytwarza opis $M-1$ taśm, z których każda zawiera $2(N-1)$ „właściwych" trójkątów. Boki pewnych trójkątów leżą na brzegu siatki. Trójkątem przylegającym do każdego takiego boku trójkąta jest on sam. Dlatego opisy pierwszej i ostatniej taśmy są generowane inaczej niż opisy pozostałych taśm, przez instrukcje w liniach 26–29 i 38–41. Opis wszystkich taśm, z indeksami restartu we właściwych miejscach, składa się z $(M-1)(4N+1)-1$ indeksów. Przykłady taśm wybranych z tablicy przez pierwszą i drugą procedurę są pokazane na rysunkach 12.10a i b.

Jako ćwiczenie pozostawiam napisanie procedury, która wytwarza opisy taśm w taki sposób, że wiersze tablicy są indeksowane modulo M, co umożliwi otrzymanie powierzchni walcowej, czyli rurki zbudowanej z M taśm. Rozbudowując to ćwiczenie, można też otrzymać zamkniętą rurkę, czyli torus. Ale zobaczmy zastosowania.

Przyległości trójkąta umożliwiają badanie, które z jego krawędzi są sylwetkowe. **Krawędź sylwetkowa** jest wspólną krawędzią trójkątów, z których jeden jest „odwrócony przodem" do obserwatora, a drugi jest „odwrócony tyłem". Szader geometrii może przesłać na swoje wyjś-



Rysunek 12.10. Schemat wybierania z tablicy wierzchołków taśm trójkątowych z przyległościami



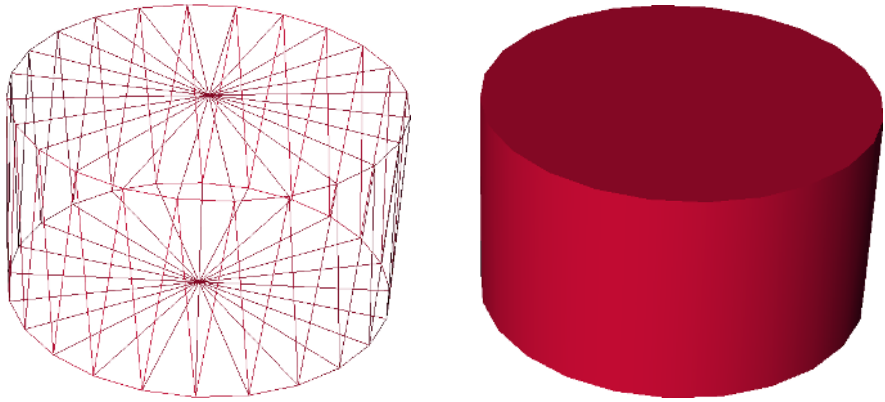
Rysunek 12.11. Sylwetka fragmentu powierzchni torusa i obraz z krawędziami sylwetkowymi

cie tylko krawędzie sylwetkowe, pokazane na rysunku 12.11a. Można nimi uzupełnić obraz „kreskowy" powierzchni, na przykład taki jak na rysunku 12.11b.

Mając „właściwy" trójkąt i trójkąty przylegające do jego boków, szader geometrii może (za pomocą funkcji `cross`[16]) obliczyć wektory normalne wszystkich czterech trójkątów, a następnie (za pomocą funkcji `dot`) iloczyny skalarne tych wektorów z wektorami różnicy położenia obserwatora i dowolnego wierzchołka każdego z tych trójkątów. Wspólna krawędź trójkątów jest sylwetkowa, jeśli odpowiednie iloczyny skalarne mają różne znaki. Zauważmy, że trójkąty, które zostały przez procedurę `GenTriangleStripAdjIndices2` doczepione jako swoje przyległości, są (w płaszczyźnie, w której leżą) zorientowane przeciwnie — wektory normalne „właściwego" trójkąta i jego samego jako przyległości będą mieć przeciwne zwroty. W rezultacie niezależnie od położenia obserwatora krawędzie, z których składa się brzeg powierzchni zbudowanej z trójkątów, zostaną rozpoznane jako krawędzie sylwetkowe.

[16] Trzeba przy tym zadbać o orientację; przy oznaczeniach z rysunku 12.8 szader fragmentów powinien obliczyć wektory $(\boldsymbol{v}_2 - \boldsymbol{v}_0) \wedge (\boldsymbol{v}_4 - \boldsymbol{v}_0)$, $(\boldsymbol{v}_1 - \boldsymbol{v}_0) \wedge (\boldsymbol{v}_2 - \boldsymbol{v}_0)$, $(\boldsymbol{v}_3 - \boldsymbol{v}_2) \wedge (\boldsymbol{v}_4 - \boldsymbol{v}_2)$ i $(\boldsymbol{v}_5 - \boldsymbol{v}_4) \wedge (\boldsymbol{v}_0 - \boldsymbol{v}_4)$.

Uwaga: Obraz na rysunku 12.11b powstał przez wyświetlenie (przy użyciu osobnych programów) krawędzi sylwetkowych, niektórych odcinków łączących wierzchołki w tablicy (tworzą one siatkę krzywych na powierzchni) i trójkątnych ścian w kolorze tła. Aby ściany nie zasłaniały swoich krawędzi (co mogłoby się zdarzyć z przyczyn rozważanych w podrozdz. 7.8), trzeba włączyć, za pomocą procedur `glPolygonOffset` i `glEnable`, odpowiednią korektę głębokości wielokątów, co jest opisane na s. 568. Zachęcam do eksperymentów, bo one dadzą Czytelnikowi więcej niż choćby najdłuższe wyjaśnienie, jakie mógłbym tu napisać.



Rysunek 12.12. Walec otrzymany z taśmy trójkątowej z przyległościami, $n = 24$

Rysunek 12.12 przedstawia obraz walca obrotowego. Można otrzymać taki obraz, rysując osobno taśmę trójkątową (powierzchnię boczną) i dwa wachlarze trójkątów (podstawy), ale dokonanie tego przez narysowanie jednej taśmy trójkątowej z przyległościami może autorowi programu sprawić dużo więcej radości. Powierzchnia boczna powstanie z „właściwych" trójkątów, a na podstawy złożą się trójkąty przylegające. Aby przykład uczynić jeszcze ciekawszym, można do wygenerowania wierzchołków użyć szaderów, co umożliwi obycie się bez zajmujących miejsce w pamięci GPU tablic ze współrzędnymi wierzchołków i dostosowywanie do wielkości obrazu walca, przez odpowiedni dobór wartości parametru procedury rysowania, poziomu szczegółowości modelu, czyli w tym przypadku liczby trójkątów.

Listing 12.16 przedstawia procedurę rysowania walca za pomocą programu zbudowanego z szaderów przedstawionych na listingach 12.17, 12.18 i 10.4. Parametr `n` określa dokładność przybliżenia walca, który zostanie otrzymany z $4n$ trójkątów — każda z jego podstaw będzie n-kątem foremnym. Parametry `h` i `r` określają wysokość i promień walca, którego środek jest początkiem układu współrzędnych modelu. Oś walca jest osią z tego układu.

Zmienne globalne `cyl_n_loc`, `cyl_r_loc` i `cyl_hh_loc` zawierają położenia zmiennych jednolitych `n`, `R` i `hh`, na podstawie których szader wierzchołków oblicza współrzędne wierzchołków trójkątów w taśmie. Zmienna `empty_vao` jest identyfikatorem obiektu tablicy wierzchołków niezawierającego opisu *żadnych* atrybutów wierzchołków — jedyny atrybut (wszystkich) wierzchołków to kolor, który można podać na przykład w zmiennej statycznej. Procedura `glDrawArrays` powoduje narysowanie taśmy z $2n$ trójkątów z przyległościami, a resztą zajmują się szadery.

Listing 12.16. Procedura rysowania walca

C

```
void DrawCylinder ( GLuint program_id, GLint n, GLfloat h, GLfloat r )
{
  glUseProgram ( program_id );
  glUniform1f ( cyl_n_loc, (GLfloat)n );
  glUniform1f ( cyl_r_loc, r );
  glUniform1f ( cyl_hh_loc, 0.5*h );
  glBindVertexArray ( empty_vao );
  glDrawArrays ( GL_TRIANGLE_STRIP_ADJACENCY, 0, 4*n+4 );
  glBindVertexArray ( 0 );
} /*DrawCylinder*/
```

Wartość zmiennej wbudowanej `gl_VertexID` szadera wierzchołków (listing 12.17) jest numerem wierzchołka w taśmie (zobacz rys. 12.9). Dwa trójkąty, które przylegają do pierwszego i ostatniego trójkąta „właściwego", zostają pominięte. Wierzchołki o numerach parzystych są położone na brzegach podstaw, przy czym jeśli numer wierzchołka jest podzielny przez 4, to wierzchołek znajdzie się na brzegu dolnej, a w przeciwnym razie na brzegu górnej podstawy. Jeśli reszta z dzielenia numeru wierzchołka przez 4 jest równa 3, to wierzchołek jest położony w środku dolnej podstawy (tj. w punkcie $(0, 0, -h/2)$), a jeśli 1, to w środku górnej podstawy (w punkcie $(0, 0, h/2)$). Współrzędne położenia wierzchołka są nadawane w liniach 27 i 30. W rezultacie z trójkątów przylegających powstają wachlarze trójkątów wypełniających podstawy walca.

Listing 12.17. Szader wierzchołków do rysowania walca

GLSL

```
#version 450 core

#define PI 3.14159265358

layout(location=0) in vec3 in_Colour;

out Vertex {
    vec3 Colour;
    vec3 MPos;
    vec3 Position;
    int  VertexID;
  } Out;

uniform TransBlock { .... } trb;  /* blok taki jak na listingu 12.3 */

uniform float n, R, hh;

void main ( void )
{
  vec4  p;
```

```
21:   int   i;
22:   float a;
23:
24:   Out.VertexID = i = gl_VertexID;
25:   if ( (i & 0x01) == 0 ) {
26:     a = float(i)*PI/(2.0*n);
27:     p = vec4 ( R*cos ( a ), R*sin ( a ), (i & 0x02) != 0 ? -hh : hh, 1.0 );
28:   }
29:   else
30:     p = vec4 ( 0.0, 0.0, (i & 0x02) != 0 ? hh : -hh, 1.0 );
31:   Out.MPos = p.xyz;
32:   Out.Position = (p = trb.mm * p).xyz;
33:   gl_Position = trb.vpm * p;
34:   Out.Colour = in_Colour;
35: } /*main*/
```

W linii 33 szader wierzchołków dokonuje przejścia od układu świata do układu kostki standardowej — dla oszczędności czasu[17]. Oprócz tego szader geometrii musi otrzymać położenie wierzchołka w układzie modelu, potrzebne do skonstruowania wektora normalnego, i położenie w układzie świata, którego dalej szader fragmentów użyje w obliczeniach oświetlenia. W konstrukcji wektorów normalnych szader geometrii potrzebuje też numeru wierzchołka w taśmie, zatem jest on przekazywany w polu `VertexID` bloku wyjściowego.

Trójkąt z przyległościami podany na wejście szadera geometrii (listing 12.18) jest opisany przez 6 wierzchołków, których położenia w układzie kostki standardowej są podane we wbudowanej tablicy `gl_in`, a dodatkowe atrybuty w tablicy `In`. W liniach 28–36 szader geometrii wyprowadza „właściwy" trójkąt, którego wierzchołki mają numery parzyste — w otrzymanych na wejściu tablicach dane opisujące te wierzchołki znajdują się na miejscach 0, 2 i 4.

Listing 12.18. Szader geometrii do rysowania walca

GLSL

```
1: #version 450 core
2:
3: layout(triangles_adjacency) in;
4: layout(triangle_strip,max_vertices=6) out;
5:
6: in Vertex {
7:     vec3 Colour;
8:     vec3 MPos;
9:     vec3 Position;
10:    int  VertexID;
11:  } In[];
12:
13: out FVertex { .... } Out;          /* blok taki jak na listingu 12.7 */
```

[17] Mógłby to zrobić szader geometrii, ale w takim rozwiązaniu wierzchołki o numerach parzystych byłyby poddawane temu przekształceniu trzykrotnie.

```
14:
15: uniform TransBlock { .... } trb;  /* blok taki jak na listingu 12.3 */
16:
17: const int aind[] = {2,3,4,0,4,5};
18:
19: void main ( void )
20: {
21:   int  i, j, k;
22:   bool b;
23:   vec3 v1, v2, tnv, nv;
24:
25:   v1 = In[2].Position - In[0].Position;
26:   v2 = In[4].Position - In[0].Position;
27:   tnv = normalize ( cross ( v1, v2 ) );
28:   for ( i = 0; i < 6; i += 2 ) {
29:     Out.Colour = In[i].Colour;
30:     Out.Position = In[i].Position;
31:     nv = mat3(trb.mmti) * vec3 ( In[i].MPos.xy, 0.0 );
32:     Out.Normal = normalize ( nv );
33:     Out.TNormal = tnv;
34:     gl_Position = gl_in[i].gl_Position;
35:     EmitVertex ();
36:   }
37:   EndPrimitive ();
38:   b = (In[4].VertexID & 0x02) != 0;
39:   nv = normalize ( mat3(trb.mmti) * vec3 ( 0.0, 0.0, b ? -1.0 : 1.0 ) );
40:   for ( i = 0, j = b ? 0 : 3;  i < 3;  i++, j++ ) {
41:     k = aind[j];
42:     Out.Colour = In[k].Colour;
43:     Out.Position = In[k].Position;
44:     Out.Normal = Out.TNormal = nv;
45:     gl_Position = gl_in[k].gl_Position;
46:     EmitVertex ();
47:   }
48:   EndPrimitive ();
49: } /*main*/
```

W liniach 25–27 szader oblicza wektor normalny płaszczyzny „właściwego” trójkąta, wyprowadzany w polu `Out.TNormal` razem z każdym wierzchołkiem. Wektor obliczany w linii 31 jest wektorem normalnym powierzchni walcowej w punkcie położenia wierzchołka trójkąta; ponieważ w układzie modelu oś walca jest osią z, za pierwsze dwie współrzędne tego wektora można przyjąć współrzędne x, y położenia wierzchołka, a trzecia jest zerem. Mnożenie przez macierz przechowywaną w zmiennej `trb.mmti`[18] jest obliczeniem wektora normalnego powierzchni walcowej w układzie świata. W linii 32 wektor ten jest przekształcany w wektor jednostkowy.

[18] Powinna to być transpozycja odwrotności macierzy przejścia od układu modelu do układu świata — odpowiada za to aplikacja.

W liniach 40–48 zostaje wyprowadzony jeden z trójkątów przylegających — ten, który na schemacie taśmy na rysunku 12.9 ma wspólny poziomy bok z trójkątem „właściwym". Do rozpoznania, czy trójkąt ten jest częścią dolnej, czy górnej podstawy walca, służy numer w taśmie wierzchołka podanego w miejscu 4 tablicy `In`. Jeśli numer ten (zawsze parzysty) jest podzielny przez 4, to właściwy trójkąt przylegający ma wierzchołki w miejscach tablicy 0, 4 i 5 i jest on częścią górnej podstawy, a zatem jego wektor normalny, skierowany na zewnątrz bryły, ma w układzie modelu współrzędne $(0, 0, 1)$. W przeciwnym razie trójkąt jest częścią dolnej podstawy; jego wierzchołki są podane na miejscach 2, 3, 4, a wektor normalny w układzie modelu otrzymuje współrzędne $(0, 0, -1)$.

Uwaga: Kolejności umieszczania na wejściu szadera geometrii „wyjętych z taśmy" wierzchołków trójkąta z przyległościami *nie znalazłem* w specyfikacjach [1] ani [3]. Szadery uruchomiłem na podstawie eksperymentów. Może się więc okazać (obym się mylił), że inna implementacja standardu (np. zrealizowana przez innego producenta sprzętu) nie wykona poprawnego obrazu. Gdyby tak było, to problem mógłby być rozwiązany przez znalezienie wierzchołka przyległości, który ma (w układzie modelu) współrzędne $x = y = 0$ i zbadanie znaku współrzędnej z tego wierzchołka.

Opisaną tu procedurę i szadery można zmodyfikować tak, aby umożliwić rysowanie stożka ściętego — trzeba w tym celu wprowadzić dwa parametry opisujące promienie podstaw i odpowiednio zmodyfikować obliczanie współrzędnych wierzchołków i wektorów normalnych. Polecam to jako ćwiczenie. Można też płaskie podstawy zamienić na powierzchnie stożkowe, ale obliczanie wektorów normalnych, jeśli ma być widoczny wierzchołek stożka, jest trochę bardziej skomplikowane. Jest ono opisane w p. 21.5.3.

12.4.4. Interpolacja atrybutów wierzchołków

Podczas rasteryzacji rysowanego odcinka lub trójkąta dla każdego fragmentu (tj. piksela) jego obrazu jest obliczana głębokość i są też interpolowane atrybuty podane dla wierzchołków tej figury. Przyjrzyjmy się dokładniej temu, co otrzymujemy.

Uwaga: Nie jest to opis algorytmu interpolacji, tylko matematyczny opis jej wyników. Różne implementacje OpenGL-a mogą te wyniki otrzymywać różnymi sposobami.

Do etapu rasteryzacji trafiają odcinki i trójkąty, które przeszły etap obcinania[19], przy czym jeśli część trójkąta pozostała po obcięciu jest wielokątem (zawsze wypukłym), to do rasteryzacji trafią trójkąty otrzymane z podziału tego wielokąta. Położenie i-tego wierzchołka jest reprezentowane przez wektor (X_i, Y_i, Z_i, W_i) współrzędnych jednorodnych w układzie kostki standardowej. Współrzędne wagowe wszystkich wierzchołków muszą mieć ten sam znak[20]. Na podstawie wymiarów klatki obliczane są współrzędne jednorodne Ξ_i, H_i wierzchołka, który w układzie okna ma współrzędne kartezjańskie $\xi_i = \Xi_i / W_i$, $\eta_i = H_i / W_i$.

[19] Interpolacja atrybutów wierzchołków podczas obcinania jest wykonywana podobnie jak opisana tu interpolacja w etapie rasteryzacji.

[20] Przypomnijmy, że nawet jeśli wszystkie wektory współrzędnych jednorodnych wierzchołków w układzie świata mają wagi równe 1, wynikiem przekształcenia perspektywicznego mogą być wektory o różnych współrzędnych wagowych. Po etapie obcinania współrzędne wagowe wszystkich wierzchołków są dodatnie.

Dla punktu $\boldsymbol{p} = (\xi, \eta)$ odcinka, którego końce są reprezentowane przez wektory współrzędnych jednorodnych (Ξ_0, H_0, Z_0, W_0) i (Ξ_1, H_1, Z_1, W_1) istnieje liczba $t \in [0,1]$, taka że

$$\frac{(1-t)\Xi_0 + t\Xi_1}{(1-t)W_0 + tW_1} = \xi, \quad \frac{(1-t)H_0 + tH_1}{(1-t)W_0 + tW_1} = \eta.$$

Daje to dwa równania liniowe:

$$\big((\Xi_1 - \Xi_0) - \xi(W_1 - W_0)\big)t = \xi W_0 - \Xi_0, \quad \big((H_1 - H_0) - \eta(W_1 - W_0)\big)t = \eta W_0 - H_0.$$

Jeśli końce odcinka mają różne obrazy, to na podstawie tych równań można obliczyć t.

Przed rasteryzacją odcinek jest zamieniany na prostokąt[21], którego dwa przeciwległe boki (o długości d określającej szerokość linii) mają środki $\boldsymbol{p}_0 = (\xi_0, \eta_0)$ i $\boldsymbol{p}_1 = (\xi_1, \eta_1)$ (rys. 12.13a). Obraz odcinka składa się z pikseli wypełniających ten prostokąt. Dla piksela, którego środek $\boldsymbol{p}$ nie jest punktem prostej $\boldsymbol{p}_0\boldsymbol{p}_1$, układ dwóch równań napisanych wyżej jest sprzeczny. Punktowi tej prostej położonemu najbliżej punktu $\boldsymbol{p}$ odpowiada parametr t otrzymany przez rozwiązanie liniowego zadania najmniejszych kwadratów postawionego dla tego układu:

$$t = \frac{\boldsymbol{a}^T\boldsymbol{b}}{\boldsymbol{a}^T\boldsymbol{a}}, \quad \text{gdzie} \quad \boldsymbol{a} = \begin{bmatrix} (\Xi_1 - \Xi_0) - \xi(W_1 - W_0) \\ (H_1 - H_0) - \eta(W_1 - W_0) \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} \xi W_0 - \Xi_0 \\ \eta W_0 - H_0 \end{bmatrix}.$$

Głębokość[22] punktu $\boldsymbol{p}$ jest równa $z = \big((1-t)Z_0 + tZ_1\big)/\big((1-t)W_0 + tW_1\big)$. Dowolny inny atrybut jest otrzymywany ze wzoru $A = (1-t)A_0 + tA_1$, gdzie A_0 i A_1 są wartościami tego atrybutu podanymi dla końców odcinka.

Wyniki interpolacji dla punktu $\boldsymbol{p} = (\xi, \eta)$ trójkąta o wierzchołkach reprezentowanych przez wektory (Ξ_0, H_0, Z_0, W_0), (Ξ_1, H_1, Z_1, W_1) i (Ξ_2, H_2, Z_2, W_2) najwygodniej jest opisać przy użyciu współrzędnych barycentrycznych t_0, t_1, t_2 określonych w układzie odniesienia tych wierzchołków. Ma być

$$\frac{\Xi_0 t_0 + \Xi_1 t_1 + \Xi_2 t_2}{W_0 t_0 + W_1 t_1 + W_2 t_2} = \xi, \quad \frac{H_0 t_0 + H_1 t_1 + H_2 t_2}{W_0 t_0 + W_1 t_1 + W_2 t_2} = \eta, \quad t_0 + t_1 + t_2 = 1,$$

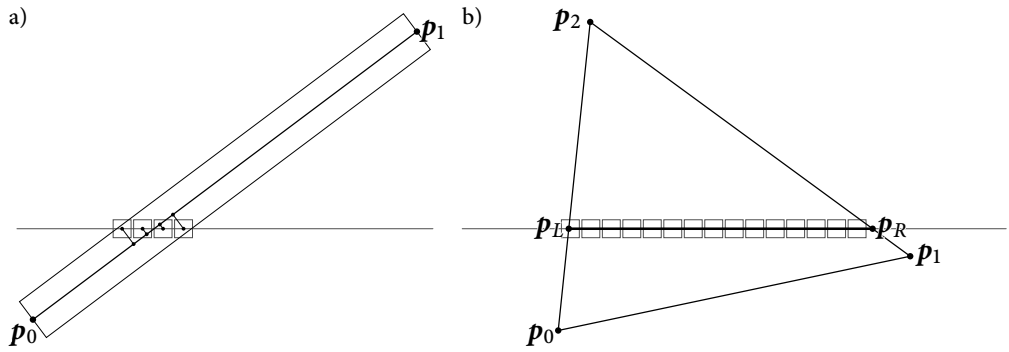
co łatwo jest przekształcić w układ trzech równań liniowych z niewiadomymi t_0, t_1, t_2. Głębokość[22] punktu $\boldsymbol{p}$ jest równa

$$z = \frac{Z_0 t_0 + Z_1 t_1 + Z_2 t_2}{W_0 t_0 + W_1 t_1 + W_2 t_2},$$

a jego atrybut A ma wartość $t_0A_0 + t_1A_1 + t_2A_2$.

[21] Może to być także równoległobok, którego dwa boki są poziome albo pionowe, zależnie od nachylenia rysowanego odcinka.

[22] Ponieważ rasteryzacji są poddawane odcinki i trójkąty obcięte do kostki standardowej, obliczona w ten sposób głębokość z jest liczbą z przedziału $[-1,1]$. Do szadera fragmentów w zmiennej `gl_FragCoord.z` trafia liczba $\zeta = (z+1)/2$.



Rysunek 12.13. Rasteryzacja odcinka i trójkąta

Obliczenia wykonywane podczas rasteryzacji trójkąta przebiegają w ten sposób, że w wierszach rastra między wierzchołkami położonymi najniżej i najwyżej są wyznaczane odcinki wypełniające trójkąt, przy czym atrybuty są interpolowane między końcami odpowiednich krawędzi trójkąta, a następnie między końcami $\boldsymbol{p}_L$ i $\boldsymbol{p}_R$ każdego takiego odcinka (rys. 12.13b).

Dowolny atrybut przekazywany do etapu obcinania można zadeklarować z kwalifikatorem `noperspective`. Wtedy jego interpolacja jest wykonywana na postawie tak przeskalowanych wektorów współrzędnych jednorodnych wierzchołków, aby ich współrzędne wagowe były równe 1. Powoduje to liniową interpolację tego atrybutu *na obrazie*, a nie w przestrzeni.

Uwaga: Jeśli jest włączony antyaliasing (zobacz rozdz. 19), to podczas rasteryzacji wyznaczanych jest wiele (kilka lub kilkanaście) punktów należących do przecięcia prymitywu (odcinka lub trójkąta) z każdym pikselem, przy czym punkty te są rozmieszczone w pikselu nieregularnie, na przykład w sposób podobny do sposobu pokazanego na rysunku 25.5. Interpolacja atrybutów wierzchołków i obliczenia głębokości dla każdego z tych punktów przebiegają tak samo, jak dla zaznaczonych na rysunku 12.13 środków pikseli.

Uwaga: Rysowane linie (tj. odcinki) mają domyślną szerokość jednego piksela. Wprawdzie istnieje procedura `glLineWidth`, której parametr może określać inną szerokość, ale w specyfikacji OpenGL 3.0 i późniejszych jest ona zdeprecjonowana. Nie ma zatem gwarancji, że procedura ta działa w oczekiwany sposób; to zależy od implementacji standardu OpenGL zainstalowanej na danym sprzęcie i od sposobu utworzenia kontekstu OpenGL-a. Procedura ta może w ogóle nie być dostępna, a jeśli jest, to jej wywołanie z parametrem większym niż 1 może spowodować błąd. Dlatego nie powinno się jej używać w aplikacjach przeznaczonych do rozpowszechniania. Jeśli aplikacja potrzebuje rysować szerokie linie, to jej autor powinien sam to zaimplementować, zamieniając odcinki na prostokąty (można do tego użyć np. szadera geometrii). Co ciekawe, specyfikacja [1] (oraz [2]) zawiera dokładny opis metody rasteryzacji szerokich linii. Wzmianka o procedurze `glLineWidth` przy tym opisie nie informuje o zdeprecjonowaniu (ta informacja jest w dodatku), co może czytelników wprowadzać w błąd.

12.4.5. Atrybuty bez interpolacji

Domyślnie atrybuty wierzchołków trójkąta (lub odcinka) w etapach obcinania i rasteryzacji są interpolowane zgodnie z opisem w poprzednim punkcie; na wejście szadera fragmentów są podawane wyniki tej interpolacji, co w większości zastosowań jest pożądane. Są jednak sytuacje, gdy szaderowi fragmentów trzeba przekazać atrybut o wartości podanej dla jednego wierzchołka — i atrybut ten ma mieć tę samą wartość dla wszystkich pikseli trójkąta. Zauważmy, że dotyczy to wszystkich atrybutów opisanych przez liczby całkowite, bo ich interpolacja albo nie ma sensu, albo wymaga przejścia do reprezentacji zmiennopozycyjnej. Przykładowym atrybutem całkowitym, który mógłby być użyty do wybrania nakładanej na obiekt tekstury, jest numer instancji rysowanego prymitywu (wiadomości na temat rysowania wielu instancji prymitywu są w rozdz. 15, a rozdz. 19 jest poświęcony teksturowaniu).

Deklaracja atrybutu, który ma nie być interpolowany, musi być poprzedzona kwalifikatorem `flat` — w dwóch miejscach: w ostatnim szaderze części przedniej potoku przetwarzania grafiki (tzn. wierzchołków, rozdrabniania albo geometrii) i w szaderze fragmentów. W szczególności kwalifikatorem tym muszą być opatrzone wszelkie podane na wejście szadera fragmentów atrybuty typu `int`, `ivec2`, `ivec3` oraz `ivec4`. Jeśli atrybuty są polami struktury (takiej jak `FVertex` w szaderach aplikacji 1B), to można tym kwalifikatorem opatrzyć indywidualne pola; pozostałe pola mają domyślny kwalifikator `smooth`, którego nie trzeba pisać.

Obiekt wyświetlany przez aplikację 1B jest zbiorem płaskich trójkątów, a wektor normalny w każdym punkcie trójkąta jest taki sam. Dlatego jednostkowy wektor normalny trójkąta, podany jako atrybut jednego wierzchołka, może być użyty bez interpolacji. W tym celu w szaderach można dokonać następujących modyfikacji: zmienić blok wyjściowy szadera geometrii i blok wejściowy szadera fragmentów odpowiednio na

```
out FVertex {
    vec3 Colour;
    vec3 Position;
    flat vec3 Normal;
  } Out;
```

```
in FVertex {
    vec3 Colour;
    vec3 Position;
    flat vec3 Normal;
  } In;
```

Ponieważ wektor normalny podany przez szader geometrii jest jednostkowy, wywołanie funkcji `normalize` można usunąć z szadera fragmentów. Odrobinę skróci to czas jego działania, ale opisane zmiany wykluczą możliwość uzyskania obrazu trójkąta wyglądającego jak fragment gładkiej powierzchni zakrzywionej.

Jak wybrać wierzchołek, którego atrybut będzie przekazany do wszystkich fragmentów trójkąta? Służy do tego procedura `glProvokingVertex`, której parametr ma dwie dopuszczalne wartości: `GL_FIRST_VERTEX_CONVENTION` albo `GL_LAST_VERTEX_CONVENTION` (która przywraca domyślny stan początkowy). Pierwsza z nich wybiera pierwszy, a ostatnia ostatni wierzchołek trójkąta lub odcinka. Reguły wybierania wierzchołków są bardziej skomplikowane, gdy szader geometrii wyprowadza wiele trójkątów lub odcinków (w postaci taśm trójkątowych lub łamanych) albo gdy jedynym szaderem w części przedniej potoku przetwarzania grafiki jest szader wierzchołków. Dokładny opis tych reguł najłatwiej jest znaleźć na stronie [7].

13

Aplikacja 1E

Wyrażając się precyzyjnie, orrerium przedstawiające jedynie Słońce, Ziemię i Księżyc nazywa się tellurium.

Rob Wilkins: *Terry Pratchett. Życie z przypisami*[1]

W kolejnej aplikacji dokonamy animacji ruchu „Ziemi" wokół „Słońca" i „Księżyca" wokół „Ziemi"[2]. Modelem każdego z tych obiektów będzie dwudziestościan przetworzony przez szader rozdrabniania na sferę. Do systematycznego opisu ruchu obiektów względem siebie użyjemy łańcucha kinematycznego; najpierw wyjaśnię, co to takiego, a potem opiszę procedury umożliwiające konstruowanie i animowanie łańcuchów w aplikacjach.

13.1. Łańcuchy kinematyczne

Bryła sztywna w przestrzeni trójwymiarowej ma 6 stopni swobody, tj. do opisu jej położenia trzeba użyć sześciu liczb (np. trzech współrzędnych kartezjańskich ustalonego punktu bryły i trzech liczb opisujących jej obrót). Jeśli dwie bryły są połączone w sposób ograniczający ich ruch względem siebie, to tworzą **parę kinematyczną**. Jeśli na przykład połączenie jest zawiasem, to układ złożony z takich brył ma 7 stopni swobody — o ile pierwszą bryłę można przesuwać i obracać dowolnie, o tyle położenie drugiej bryły po ustaleniu położenia pierwszej bryły i kąta obrotu drugiej wokół osi zawiasu jest określone jednoznacznie. Tak więc para utworzona przy użyciu zawiasu odebrała układowi 5 stopni swobody.

Łańcuch kinematyczny (*kinematic linkage*) *w mechanice* jest to układ brył sztywnych zwanych **członami** (*links*), których ruch względem siebie jest ograniczony przez pary kinematyczne (*joints*). Z każdą parą kinematyczną jest związany co najmniej jeden **parametr artykulacji** (*articulation parameter*), który opisuje na przykład kąt obrotu zawiasu lub przesunięcie pręta w prowadnicy lub tłoka w cylindrze. Pary z jednym parametrem artykulacji są zwane **prostymi**, a pary mające dwa lub więcej parametrów (takie jak przegub kulowy lub para wymuszająca pozostawanie obiektu na płaszczyźnie, ale pozwalająca go na niej dowol-

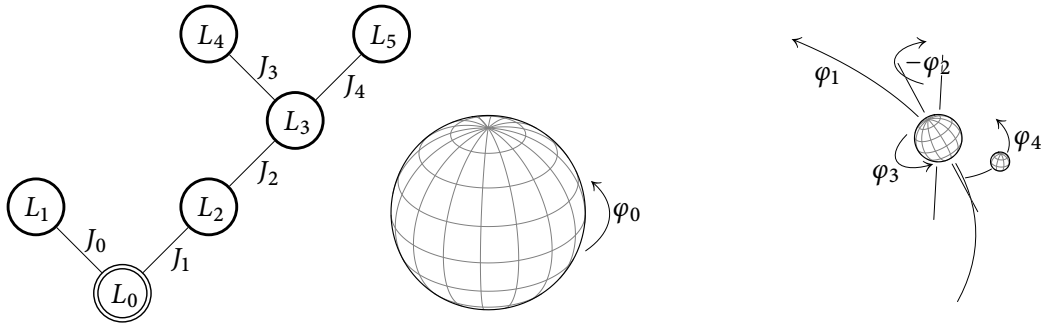
[1] Przekład Piotra W. Cholewy, Insignis Media, 2023

[2] Dalej cudzysłowy pomijam.

nie przesuwać lub obracać) są **złożone**. Łańcuch kinematyczny może być opisany za pomocą grafu, którego wierzchołkami są człony, a krawędziami — pary kinematyczne. Łańcuch jest **otwarty**, jeśli ten graf nie zawiera cyklu, w przeciwnym razie mamy łańcuch **zamknięty**. Zauważmy, że parametry artykulacji par kinematycznych tworzących cykl nie są niezależne. Z jednej strony poszczególne pary mogą odbierać te same stopnie swobody, na przykład jeśli to są zawiasy, których osie są równoległe. Z drugiej strony ustalenie parametrów dla pewnych par może jednoznacznie określić wartości parametrów wszystkich pozostałych par w cyklu, przy czym aby je znaleźć, należy na ogół rozwiązać układ równań nieliniowych. Taki układ może być sprzeczny (tj. nie mieć rozwiązań), może też zdarzyć się kolizja członów. Tymi problemami nie będziemy się tu zajmować, ograniczając się do modelowania **łańcuchów otwartych** (których grafy są drzewami) i nie zwracając uwagi na kolizje.

Artykulacja łańcucha kinematycznego polega na znalezieniu położeń wszystkich członów łańcucha po ustaleniu wartości parametrów artykulacji i położenia pewnego członu łańcucha.

W programie człony łańcucha kinematycznego będą układami współrzędnych kartezjańskich w przestrzeni. W poszczególnych układach będą określone **obiekty**, a dokładniej wierzchołki lub punkty kontrolne figur rysowanej sceny. Mogą być człony, z którymi nie związemy żadnych obiektów, mogą być też człony z wieloma przywiązanymi obiektami. Co więcej, różne części jednego obiektu mogą być związane z różnymi członami łańcucha. Artykulacja spowoduje odkształcanie takiego obiektu.



Rysunek 13.1. Łańcuch kinematyczny Słońca, Księżyca i Ziemi

Na rysunku 13.1 jest pokazany graf łańcucha tworzonego przez opisaną w tym rozdziale aplikację. Łańcuch ma 6 członów (oznaczonych symbolami $L_0, \ldots, L_5$) i 5 par kinematycznych ($J_0, \ldots, J_4$); jego graf jest drzewem. Słońce, Ziemia i Księżyc są związane odpowiednio z członami L_1, L_4 i L_5. Pozostałe człony służą do określenia par kinematycznych realizujących dopuszczalne przemieszczenia obiektów względem siebie.

Opisane dalej procedury obsługi łańcucha umożliwiają arbitralne ustalenie położenia dowolnego członu łańcucha w układzie świata, a ponadto możliwe jest zmienianie wyboru tego członu podczas działania programu[3]. W opisanej tu aplikacji członem o ustalonym położeniu

[3]Może się to przydać, jeśli na przykład łańcuch reprezentuje idącego ludzika (lub dzika), którego stopy na przemian mają kontakt z podłożem i w związku z tym ich położenia w układzie świata na przemian dają początek artykulacji jego pozostałych części ciała.

(nazwiemy go **korzeniem**) jest człon L_0. Artykulacja łańcucha jest dokonywana za pomocą przeszukiwania grafu łańcucha w głąb (DFS), zaczynając od korzenia. Dla każdego członu należy znaleźć macierz przejścia od układu współrzędnych *będącego* tym członem do układu świata; taką macierz dla członu L_i oznaczymy symbolem A_i. Z każdą parą kinematyczną, tj. krawędzią grafu, są związane trzy przekształcenia reprezentowane przez macierze 4×4, które dla j-tej krawędzi oznaczymy F_j, R_j, B_j. Macierze F_j oraz B_j są ustalone, natomiast współczynniki macierzy R_j zależą od wartości jednego parametru artykulacji[4]: $R_j = R_j(\varphi_j)$. Aby znaleźć macierz A_i, podczas przeszukiwania grafu trzeba obliczyć iloczyn macierzy związanych z kolejnymi krawędziami przebywanymi podczas przeszukiwania. Pierwszy czynnik to macierz A_k przejścia dla korzenia L_k. Macierze kolejno przebywanych krawędzi (par kinematycznych) są domnażane z prawej strony. W przykładzie z rysunku 13.1 ścieżka od korzenia L_0 do wierzchołka L_4 składa się z krawędzi J_1, J_2 i J_3, zatem macierz przejścia od układu (członu) L_4 do układu świata jest dana wzorem

$$A_4 = A_0F_1R_1(\varphi_1)B_1F_2R_2(\varphi_2)B_2F_3R_3(\varphi_3)B_3.$$

Podobnie dla członów L_1 i L_5 otrzymamy odpowiednio macierze

$$A_1 = A_0F_0R_0(\varphi_0)B_0,$$
$$A_5 = A_0F_1R_1(\varphi_1)B_1F_2R_2(\varphi_2)B_2F_4R_4(\varphi_4)B_4.$$

Symbole $\varphi_0,\ldots,\varphi_4$ oznaczają parametry artykulacji, które są kątami obrotów wokół pewnych osi. Opiszę je dalej.

Tymczasem zauważmy, że na przykład

$$A_0 = A_4B_3^{-1}R_3^{-1}(\varphi_3)F_3^{-1}B_2^{-1}R_2^{-1}(\varphi_2)F_2^{-1}B_1^{-1}R_1^{-1}(\varphi_1)F_1^{-1}.$$

Gdybyśmy zatem ustalili położenie w przestrzeni członu L_4, aby od niego zacząć przeszukiwanie grafu (czyli wybrali ten człon na korzeń), to po drodze do członu L_0 należy jako czynniki obliczanych iloczynów przyjmować odwrotności opisanych wcześniej macierzy[5], a ponadto macierze F_j i B_j dla krawędzi przebywanych w drugą stronę „zamieniają się miejscami". Tak więc krawędź reprezentująca dowolną parę kinematyczną musi być zorientowana. Rozwiązanie umożliwiające przeszukiwanie grafu łańcucha, zaczynając od dowolnego wierzchołka, polega na używaniu **pary półkrawędzi** reprezentującej każdą krawędź grafu. Półkrawędzie w parze są zorientowane przeciwnie (wierzchołek końcowy jednej z nich jest początkiem drugiej i nawzajem)[6] i jeśli z jedną z nich jest związana trójka macierzy (F_j, R_j, B_j), to z drugą półkrawędzią jest związana trójka $(B_j^{-1}, R_j^{-1}, F_j^{-1})$.

W łańcuchu z rysunku 13.1 macierze F_j reprezentują złożenia pewnych przesunięć i obrotów, a każda macierz $R_j(\varphi)$ reprezentuje obrót o kąt φ wokół osi z układu współrzędnych. Stałe przesunięcia mają na celu odsunięcie Ziemi i Księżyca od środków ich orbit (na odległość równą promieniowi orbity), a stałe obroty zapewniają odpowiednie nachylenia osi

[4] lub większej ich liczby, w innych możliwych aplikacjach

[5] Zakładamy, że wszystkie te macierze są nieosobliwe, a więc mają odwrotności.

[6] Mamy tu w istocie dwa grafy — zwykły, którego krawędzie to pary kinematyczne, i graf skierowany, którego krawędzie to nasze półkrawędzie.

obrotów, które będą animowane, względem ekliptyki (tj. płaszczyzny orbity Ziemi). W ten sposób jest otrzymane nachylenie osi obrotu Ziemi, któremu zawdzięczamy pory roku, i nachylenie orbity Księżyca względem ekliptyki. Szczegóły konstrukcji związanych z parami kinematycznymi macierzy stałych F_j i B_j opiszę w p. 13.3. Macierz A_0 w opisanej tu aplikacji jest jednostkowa.

Jeśli F_j jest macierzą przesunięcia o *wektor* $\boldsymbol{f}_j$ oraz $B_j = F_j^{-1}$ (czyli macierz B_j opisuje przesunięcie o wektor $-\boldsymbol{f}_j$), 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). Zauważmy ponadto, że $R_j^{-1}(\varphi) = R_j(-\varphi)$ oraz $\left(F_j R_j(\varphi) F_j^{-1}\right)^{-1} = F_j R_j(-\varphi) F_j^{-1}$ — przekształcenie odwrotne do obrotu o kąt φ wokół dowolnej osi jest obrotem wokół tejże osi o kąt $-\varphi$.

Rozważmy teraz złożenie przekształceń związanych z kolejnymi dwiema krawędziami w ścieżce prowadzącej od korzenia do dowolnego innego wierzchołka grafu; powiedzmy, że to są krawędzie J_j oraz J_k. Możemy napisać

$$F_j R_j(\varphi_j) B_j F_k R_k(\varphi_k) B_k = F_j R_j(\varphi_j) T_{kj} R_k(\varphi_k) B_k.$$

Jeśli macierze $B_j^{-1} = F_j$ i F_k reprezentują przesunięcia odpowiednio o wektory $\boldsymbol{f}_j$ i $\boldsymbol{f}_k$, to iloczyn $T_{kj} = F_j^{-1} F_k$ reprezentuje złożenie tych przesunięć, czyli przesunięcie o wektor $\boldsymbol{t}_{kj} = \boldsymbol{f}_k - \boldsymbol{f}_j$. Jeśli macierze R_j i R_k reprezentują obroty wokół osi równoległych, a wektor $\boldsymbol{t}_{kj}$ jest do tych osi prostopadły, to określa przesunięcie nakładające oś pierwszego na oś drugiego obrotu. W ogólności (także gdy osie obrotów przecinają się lub są skośne) wektor $\boldsymbol{t}_{kj}$ jest różnicą *pewnych* punktów, z których pierwszy leży na osi drugiego obrotu, a drugi na osi pierwszego. Przykłady przekształceń konstruowanych w przedstawiony wyżej sposób znajdziemy w aplikacji 2H (zobacz rozdz. 23).

Z każdym *obiektem* związemy jeszcze jedną macierz, którą dla l-tego obiektu oznaczymy symbolem E_l. Macierz ta reprezentuje dodatkowe, wstępne przekształcenie obiektu. Wszystkie obiekty wyświetlane przez aplikację 1E są kulami, a właściwie obrazami kuli otrzymanej przez rozdrabnianie ścian jednego dwudziestościanu. W układzie modelu wierzchołki otrzymanych podczas rozdrabniania trójkątów leżą na sferze jednostkowej. Macierze dla Słońca, Ziemi i Księżyca opisują jednokładności o współczynnikach skali równych promieniom tych ciał niebieskich.

13.2. Procedury obsługi łańcucha kinematycznego

W tym podrozdziale przedstawiam biblioteczkę procedur w języku C (działających na CPU), które służą do zbudowania łańcucha kinematycznego i umożliwiają, po ustaleniu wartości parametrów artykulacji, znalezienie macierzy A_i dla wszystkich jego członów. Można te procedury zastosować w programie, który użyje CPU do obliczenia współrzędnych punktów reprezentujących obiekty (tj. wierzchołków lub punktów kontrolnych) w układzie świata. Inna możliwość to przesłanie macierzy A_i (a raczej iloczynów $A_i E_l$) do pamięci GPU. Wtedy zadanie przekształcania punktów do układu świata może wykonać szader obliczeniowy, który,

działając równolegle na wielu procesorach GPU, wykona to obliczenie dla wszystkich punktów naraz. Z tej możliwości skorzystamy w aplikacjach 2H i 3B.

Grafy, takie jak w łańcuchu kinematycznym, można reprezentować przy użyciu wskaźnikowych struktur danych. Wybrałem jednak reprezentację składającą się z tablic; identyfikatory członów, półkrawędzi i innych elementów łańcucha są indeksami do tablic. Taka reprezentacja ma tę zaletę, że (w razie potrzeby) łatwiej jest ją zapisywać w i odczytywać z pliku.

W reprezentacji łańcucha mamy zatem tablicę **członów** (struktur typu `kl_link`), **półkrawędzi** par kinematycznych (struktur `kl_halfjoint`), **obiektów** (struktur `kl_object`) i **referencji obiektów** (`kl_obj_ref`); struktury te są zdefiniowane w pliku `linkage.h` i przedstawione na listingu 13.1. Obiekt zawiera tablicę swoich punktów i metody (procedury) ich przetwarzania. Referencja obiektu jest wykazem punktów obiektu związanych z konkretnym członem łańcucha; może zawierać informację, że *wszystkie* punkty obiektu są związane z danym członem albo tablicę indeksów (wybranych) punktów obiektu związanych z tym członem. Jednym z warunków poprawności łańcucha jest to, że każdy punkt każdego obiektu jest związany z pewnym członem, poprzez którąś z referencji.

W liniach `1–14` na listingu są definicje identyfikatorów możliwych rodzajów par kinematycznych; liczby ukryte pod tymi identyfikatorami są przypisywane polom `art` struktur `kl_halfjoint`. W każdej parze półkrawędzi reprezentujących parę kinematyczną jedna z półkrawędzi ma pole `art` o wartości `KL_ART_OTHERHALF`, która oznacza, że na podstawie odpowiednich parametrów artykulacji trzeba wyznaczyć macierz R_j dla drugiej półkrawędzi w parze i dla pierwszej półkrawędzi przyjąć odwrotność tamtej macierzy.

Kolejne identyfikatory określają sposób, w jaki macierz R_j może zależeć od parametrów artykulacji; `KL_ART_NONE` odpowiada macierzy jednostkowej (a więc nie ma tu pary kinematycznej, człony są połączone sztywno), a `KL_ART_TRANS_X`, …, `KL_ART_TRANS_XYZ` wyznaczają przesunięcia, wzdłuż osi x, y, z i dowolne (w tym ostatnim przypadku trzeba podać trzy parametry artykulacji, będące współrzędnymi wektora przesunięcia). Identyfikatory `KL_ART_SCALE_X`, …, `KL_ART_SCALE_XYZ` opisują skalowania osi x, y, z i wszystkich trzech osi naraz (skalowania nie są izometriami, zatem zmienianie parametrów artykulacji takich par kinematycznych sprawi, że obiekty nie będą sztywne), wreszcie identyfikatory `KL_ART_ROT_X`, …, `KL_ART_ROT_V` określają pary obrotowe, umożliwiające obroty wokół osi x, y, z i obroty wokół osi o dowolnym kierunku wyznaczonym przez wektor $\boldsymbol{v}$, którego współrzędne są parametrami artykulacji w dodatku do kąta obrotu (zatem są tu 4 parametry artykulacji).

Struktura `kl_object` opisuje pojedynczy obiekt, który może być na przykład bryłą wielościenną lub (w aplikacji 2H) zespołem płatów Béziera. Pola `type` i `id` zawierają typ i identyfikator obiektu, do dowolnego użycia przez aplikację. Pole `nvc` ma wartość 3 albo 4; określa ono liczbę współrzędnych każdego punktu (wierzchołka lub punktu kontrolnego) obiektu; w pierwszym przypadku to są współrzędne kartezjańskie w $\mathbb{R}^3$, a w drugim współrzędne jednorodne. Wartość pola `nvert` jest liczbą punktów. Pola `vert` i `tvert` są wskaźnikami tablic tych punktów, przy czym pierwsza tablica zawiera punkty oryginalne, dane w układzie, w którym obiekt został zdefiniowany, a do drugiej będą wpisywane współrzędne punktów w układzie świata, otrzymane jako wynik artykulacji. Jeśli aplikacja (taka jak 2H) przecho-

Listing 13.1. Struktury łańcucha kinematycznego

C

```
#define KL_ART_OTHERHALF -1
#define KL_ART_NONE       0
#define KL_ART_TRANS_X    1
#define KL_ART_TRANS_Y    2
#define KL_ART_TRANS_Z    3
#define KL_ART_TRANS_XYZ  4
#define KL_ART_SCALE_X    5
#define KL_ART_SCALE_Y    6
#define KL_ART_SCALE_Z    7
#define KL_ART_SCALE_XYZ  8
#define KL_ART_ROT_X      9
#define KL_ART_ROT_Y     10
#define KL_ART_ROT_Z     11
#define KL_ART_ROT_V     12

typedef struct kl_linkage *kl_lkgptr;
typedef struct kl_object  *kl_objptr;

typedef char (*kl_obj_init)(kl_lkgptr, kl_objptr);
typedef void (*kl_obj_transform)(kl_lkgptr,kl_objptr,int,GLfloat*,int,int*);
typedef void (*kl_obj_postprocess)(kl_lkgptr,kl_objptr);
typedef void (*kl_obj_redraw)(kl_lkgptr,kl_objptr);
typedef void (*kl_obj_destroy)(kl_lkgptr,kl_objptr);

typedef struct kl_object {
    int                type;
    int                id;
    int                nvc;
    int                nvert;
    GLfloat            *vert, *tvert;
    GLfloat            Etr[16];
    void               *usrdata;
    kl_obj_transform   transform;
    kl_obj_postprocess postprocess;
    kl_obj_redraw      redraw;
    kl_obj_destroy     destroy;
  } kl_object;

typedef struct kl_obj_ref {
    int on;
    int nextr;
    int nv;
    int *vn;
  } kl_obj_ref;

```

```
46: typedef struct kl_link {
47:     int  fref;
48:     int  fhj;
49:     char tag;
50:   } kl_link;
51:
52: typedef struct kl_halfjoint {
53:     int     l0, l1;
54:     int     otherhalf;
55:     int     nexthj;
56:     int     pnum;
57:     int     art;
58:     GLfloat Ftr[16];
59:     GLfloat Rtr[16];
60:     GLfloat Btr[16];
61:   } kl_halfjoint;
62:
63: typedef struct kl_linkage {
64:     int          maxobj,    nobj;
65:     int          maxorefs,  norefs;
66:     int          maxlinks,  nlinks;
67:     int          maxhj,     nhj;
68:     int          maxartpar, nartpar;
69:     kl_object    *obj;
70:     kl_obj_ref   *oref;
71:     kl_link      *link;
72:     kl_halfjoint *hj;
73:     double       *artp, *prevartp;
74:     int          current_root;
75:     GLfloat      current_root_tr[16];
76:     void         *usrdata;
77:   } kl_linkage;
```

wuje te punkty tylko w pamięci GPU, to pola `vert` i `tvert` mogą mieć wartość `NULL`. Pole `Etr` przechowuje macierz E_l opisaną wcześniej. Pole `usrdata` jest wskaźnikiem dowolnych danych skojarzonych z obiektem przez aplikację.

Pola `transform`, `postprocess`, `redraw` i `destroy` wskazują metody obiektu, których działanie będzie opisane dalej. Tworząc obiekt, aplikacja może podać wskaźniki procedur, które mają być tymi metodami, albo wskaźniki puste (`NULL`), co spowoduje przypisane obiektowi metod domyślnych przedstawianej tu biblioteczki.

Struktura `kl_obj_ref` opisuje referencję obiektu. Struktury te są przechowywane w tablicy, w której są łączone w listy przypisane do członów łańcucha. Pole `on` jest numerem obiektu (tj. indeksem do tablicy obiektów struktury łańcucha). Pole `nextr` jest wskaźnikiem następnego elementu listy referencji (czyli indeksem do tablicy referencji), przy czym wskaźnik pusty w ostatnim elemencie listy jest reprezentowany przez liczbę `-1`. Pole `nv` opisuje, ile punktów obiektu jest przywiązanych do członu przez daną referencję, a pole `vn` jest wskaźnikiem tablicy indeksów tych punktów (indeksy te określają punkty w tablicach `vert`

i `nvert` obiektu lub w odpowiednich tablicach w pamięci GPU). Jeśli pole `vn` ma wartość `NULL`, to z danym członem są związane wszystkie punkty obiektu.

Struktura `kl_link` opisuje człon łańcucha. Pole `fref` wskazuje pierwszy element listy referencji członu (jest to indeks do tablicy referencji). Pole `fhj` wskazuje pierwszy element listy par kinematycznych wiążących ten człon. Podobnie, jest to indeks do tablicy półkrawędzi reprezentujących krawędź grafu łańcucha, czyli parę kinematyczą. Dany człon jest początkiem tej półkrawędzi. Pole `tag` służy do zaznaczania odwiedzonych wierzchołków podczas przeszukiwania grafu metodą DFS — przeszukiwany jest graf skierowany wzdłuż półkrawędzi, a każda ich para reprezentująca parę kinematyczną tworzy cykl, dlatego takie zaznaczanie jest konieczne w celu uniknięcia wejścia procedury przeszukiwania grafu w nieskończoną pętlę. Ponadto, choć aplikacje opisane w tej książce tego nie robią, możliwe jest utworzenie łańcucha zamkniętego, którego krawędzie tworzą cykle, ale do zrealizowania tej możliwości należałoby dopisać procedury rozwiązujące równania opisujące zamykanie łańcucha[7]. Napisanie dobrze działających procedur rozwiązujących to zadanie dla szerokiej klasy przypadków nie jest łatwe.

Struktura `kl_halfjoint` reprezentuje półkrawędź grafu. Pola `l0` i `l1` zawierają numery (indeksy do tablicy) członów, z których pierwszy jest początkiem, a drugi końcem półkrawędzi. Pole `otherhalf` jest numerem drugiej półkrawędzi w parze; początkiem i końcem tej drugiej półkrawędzi są człony `l1` i `l0`. Pole `nexthj` jest wskaźnikiem następnej półkrawędzi w liście półkrawędzi wychodzących z członu `l0` (lub jest to „wskaźnik pusty", czyli liczba `-1` sygnalizująca koniec listy). Pole `pnum` jest indeksem do tablicy parametrów artykulacji; wskazuje ono pierwszy parametr pary, przy czym liczba tych parametrów jest określona przez pole `art`, mające wartość ukrytą za jednym z wcześniej opisanych identyfikatorów, takich jak `KL_ART_ROT_X`. Zawsze jedna z półkrawędzi ma w tym polu identyfikator `KL_ART_OTHERHALF`, a jej druga połowa jeden z pozostałych identyfikatorów. W polach `Ftr`, `Rtr` i `Btr` są przechowywane macierze F_j, R_j i B_j.

Struktura `kl_linkage` jest punktem wejściowym do reprezentacji całego łańcucha. Pola `maxobj`, `maxorefs`, `maxlinks`, `maxhj` i `maxartpar` określają długości tablic zarezerwowanych dla obiektów, referencji, członów, półkrawędzi i parametrów artykulacji, czyli maksymalne (zamówione przez aplikację) liczby tych elementów łańcucha. Pola `nobj`, `norefs`, `nlinks`, `nhj` i `nartpar` oznaczają bieżące (w trakcie budowania łańcucha i po jego zakończeniu) liczby obiektów, referencji itd. Pola `obj`, `oref`, `link` i `hj` wskazują tablice obiektów, referencji, członów i półkrawędzi. W tablicy wskazywanej przez pole `artp` są przechowywane bieżące wartości parametrów artykulacji, z kolei w tablicy `prevartp` są pamiętane ich wartości poprzednie. Dzięki temu po zmianie niektórych parametrów artykulacji można obliczać tylko te macierze R_j, które zależą od zmienionych parametrów.

Pole `current_root` jest numerem członu, od którego zaczyna się przeszukiwanie grafu. W tablicy `current_root_tr` jest przechowywana macierz przejścia od układu współrzędnych korzenia do układu świata. Parametr `usrdata` może wskazywać dowolną strukturę danych określoną przez aplikację.

[7] Dla każdego cyklu $(J_{i_1}, \ldots, J_{i_m})$ iloczyn macierzy $F_{i_1}R_{i_1}B_{i_1}\cdot\ldots\cdot F_{i_m}R_{i_m}B_{i_m}$ musi być macierzą jednostkową. Niewiadomymi w takim równaniu są nieokreślone „z góry" zmienne artykulacji, określające macierze R_{i_j}.

Budowę łańcucha należy zacząć od wywołania procedury `kl_NewLinkage` (listing 13.2), która rezerwuje pamięć na strukturę `kl_linkage` i wszystkie wskazywane przez nią tablice i nadaje polom tej struktury wartości początkowe, opisujące limity liczb elementów w tablicach i początkowe liczby tych elementów, równe 0. Kolejne parametry deklarują zapotrzebowanie na tablice wystarczające do pomieszczenia odpowiednich liczb obiektów, członów, referencji, par kinematycznych i parametrów artykulacji. Długość rezerwowanej tablicy półkrawędzi jest dwukrotnie większa niż deklarowana liczba par kinematycznych (tj. wartość parametru `maxj`). Ostatni parametr jest wskaźnikiem dowolnej struktury określonej w aplikacji, którą ta zamierza skojarzyć z całym łańcuchem.

Pole `current_root` otrzymuje domyślną wartość 0, a do tablicy `current_root_tr` są wpisywane współczynniki macierzy jednostkowej.

Listing 13.2. Procedury `kl_NewLinkage` i `kl_DestroyLinkage`

C

```
kl_linkage *kl_NewLinkage ( int maxo, int maxl, int maxr,
                            int maxj, int maxp, void *usrdata )
{
  kl_linkage *lkg;

  lkg = malloc ( sizeof(kl_linkage) );
  if ( lkg ) {
    memset ( lkg, 0, sizeof(kl_linkage) );
    lkg->maxobj    = maxo;
    lkg->maxlinks  = maxl;
    lkg->maxorefs  = maxr;
    lkg->maxhj     = 2*maxj;
    lkg->maxartpar = maxp;
    lkg->obj  = malloc ( maxo*sizeof(kl_object) );
    lkg->link = malloc ( maxl*sizeof(kl_link) );
    lkg->oref = malloc ( maxr*sizeof(kl_obj_ref) );
    lkg->hj   = malloc ( 2*maxj*sizeof(kl_halfjoint) );
    lkg->artp = malloc ( 2*maxp*sizeof(double) );
    if ( !lkg->obj || !lkg->link || !lkg->oref || !lkg->hj || !lkg->artp )
      goto failure;
    lkg->prevartp = &lkg->artp[maxp];
    memset ( lkg->artp, 0, maxp*sizeof(double) );
    memset ( lkg->prevartp, 0x3f, maxp*sizeof(double) );
    lkg->current_root = 0;
    M4x4Identf ( lkg->current_root_tr );
    lkg->usrdata = usrdata;
  }
  return lkg;

failure:
  if ( lkg->obj )  free ( lkg->obj );
  if ( lkg->link ) free ( lkg->link );
```

```
33:   if ( lkg->oref ) free ( lkg->oref );
34:   if ( lkg->hj )   free ( lkg->hj );
35:   if ( lkg->artp ) free ( lkg->artp );
36:   free ( lkg );
37:   return NULL;
38: } /*kl_NewLinkage*/
39:
40: void kl_DestroyLinkage ( kl_linkage *linkage )
41: {
42:   int i;
43:
44:   if ( linkage->obj ) {
45:     for ( i = 0; i < linkage->nobj; i++ ) {
46:       if ( linkage->obj[i].destroy )
47:         linkage->obj[i].destroy ( linkage, &linkage->obj[i] );
48:     }
49:     free ( linkage->obj );
50:   }
51:   if ( linkage->oref ) {
52:     for ( i = 0; i < linkage->norefs; i++ )
53:       if ( linkage->oref[i].vn )
54:         free ( linkage->oref[i].vn );
55:     free ( linkage->oref );
56:   }
57:   if ( linkage->link ) free ( linkage->link );
58:   if ( linkage->hj )   free ( linkage->hj );
59:   if ( linkage->artp ) free ( linkage->artp );
60:   free ( linkage );
61: } /*kl_DestroyLinkage*/
```

Procedura `kl_DestroyLinkage` likwiduje łańcuch, w tym zwalnia pamięć zajmowaną przez wszystkie tablice. Przedtem dla każdego obiektu wywoływana jest procedura destruktora podana przez aplikację podczas tworzenia obiektu. Procedurę `kl_DestroyLinkage` wypada wywołać podczas sprzątania przy końcu działania aplikacji.

Listing 13.3 przedstawia procedurę `kl_NewObject`, którą trzeba wywołać tyle razy, ile obiektów ma być przyczepionych do łańcucha. Parametr `linkage` jest wskaźnikiem struktury wcześniej utworzonej przez procedurę `kl_NewLinkage`, parametr `type` jest określonym przez aplikację typem obiektu, a parametry `nvc` i `nvert` określają liczbę współrzędnych (3 albo 4) każdego punktu obiektu i liczbę tych punktów. Parametr `etrans` wskazuje tablicę ze współczynnikami macierzy E_l dla obiektu; jeśli ma wartość `NULL`, to przyjęta zostanie macierz jednostkowa. Parametr `usrdata` jest wskaźnikiem, który zostanie zapamiętany w polu `usrdata` obiektu.

Ostatnie pięć parametrów to adresy procedur, które mają być dostarczonymi przez aplikację metodami obiektu. Definicje typów tych parametrów, tj. typ wyniku i typy parametrów każdej metody są podane na listingu 13.1. Pierwsza metoda, czyli konstruktor obiektu, zostaje wywołana (jednorazowo) przez procedurę `kl_NewObject`, a adresy pozostałych metod

są zapamiętywane w strukturze obiektu. Każdy z tych parametrów może mieć wartość NULL i wtedy obiekt będzie miał odpowiednią metodę domyślną.

Listing 13.3. Procedura `kl_NewObject`

C

```
int kl_NewObject ( kl_linkage *linkage, int type, int nvc, int nvert,
                   const GLfloat etrans[16], void *usrdata,
                   kl_obj_init init,
                   kl_obj_transform transform,
                   kl_obj_postprocess postprocess,
                   kl_obj_redraw redraw,
                   kl_obj_destroy destroy )
{
  int        on;
  kl_object *obj;

  if ( linkage->nobj < linkage->maxobj ) {
    on = linkage->nobj;
    obj = &linkage->obj[on];
    memset ( obj, 0, sizeof(kl_object) );
    obj->type = type;
    obj->id = on;
    if ( etrans )
      memcpy ( obj->Etr, etrans, 16*sizeof(GLfloat) );
    else
      M4x4Identf ( obj->Etr );
    obj->transform = transform != NULL ? transform : kl_DefaultTransform;
    obj->postprocess = postprocess != NULL ? postprocess : kl_obj_stub;
    obj->redraw = redraw != NULL ? redraw : kl_obj_stub;
    obj->destroy = destroy != NULL ? destroy : kl_obj_stub;
    obj->nvc = nvc;
    obj->nvert = nvert;
    obj->vert = obj->tvert = NULL;
    obj->usrdata = usrdata;
    if ( !init || init ( linkage, obj ) ) {
      linkage->nobj ++;
      return on;
    }
  }
  return -1;
} /*kl_NewObject*/
```

Parametr `init` procedury `kl_NewObject` jest adresem konstruktora obiektu. Konstruktor może zarezerwować tablice wierzchołków i przypisać ich adresy polom `vert` i `tvert`. Może on też utworzyć referencje obiektu, a także inne elementy jego reprezentacji, takie jak dodatkowe tablice lub bufory OpenGL-a w pamięci GPU, których identyfikatory zapamięta w strukturze wskazywanej przez pole `usrdata`. Konstruktor powinien zawiadomić proce-

durę `kl_NewObject` o sukcesie inicjalizacji, podając wartość niezerową (np. `true`), a o porażce, podając 0 (czyli `false`).

Domyślna metoda `transform`, czyli procedura `kl_DefaultTransform` pokazana na listingu 13.4, dokonuje przekształcenia punktów do układu świata, czytając punkty z tablicy `vert` obiektu i wpisując wyniki przekształcenia do tablicy `tvert`. Pozostałe trzy metody domyślne nic nie robią.

Listing 13.4. Procedura `kl_DefaultTransform`

C

```
void kl_DefaultTransform ( kl_linkage *linkage, kl_object *obj,
                           int refn, GLfloat tr[16], int nv, int *vn )
{
  int     i, k;
  GLfloat *vert, *tvert;

  if ( !(vert = obj->vert) || !(tvert = obj->tvert) )
    return;
  if ( vn ) {
    switch ( obj->nvc ) {
  case 3:
      for ( i = 0; i < nv; i++ )
        { k = 3*vn[i];  M4x4MultMP3f ( &tvert[k], tr, &vert[k] ); }
      break;
  case 4:
      for ( i = 0; i < nv; i++ )
        { k = 4*vn[i];  M4x4MultMVf ( &tvert[k], tr, &vert[k] ); }
      break;
  default:
      return;
    }
  }
  else {
    .... /* tu przekształcanie wszystkich punktów, od 0 do nv-1 */
  }
} /*kl_DefaultTransform*/
```

Choć aplikacja 1E (ani żadna inna opisana w tej książce) nie korzysta z procedury `kl_DefaultTransform`, przyjrzyjmy się, jak ta procedura działa. Może ona być wywoływana przez opisaną dalej procedurę `kl_Articulate` z parametrami zawierającymi następujące informacje: `linkage` i `obj` są wskaźnikami struktury łańcucha i obiektu, `refn` jest numerem referencji, `tr` jest tablicą współczynników macierzy przekształcenia (macierzy $A_i E_l$), a parametry `nv` i `vn` podają liczbę wierzchołków obiektu opisanych przez tę referencję i tablicę z numerami tych wierzchołków. Jeśli parametr `vn` ma wartość `NULL`, to należy przekształcić wszystkie punkty obiektu. W przeciwnym razie należy przekształcić `nv` punktów o indeksach podanych we wskazywanej przez ten parametr tablicy. Zależnie od liczby współrzędnych kolejne punkty są opisane przez trójki albo czwórki liczb podanych w tablicy `obj->vert`.

Zatem, po odpowiednie trójki albo czwórki liczb procedura sięga do miejsc otrzymanych przez pomnożenie numeru z tablicy `vn` przez 3 albo 4. Przypomnijmy, że procedura `M4x4MultMP3f` działa tak, jakby dołączała czwartą współrzędną jednorodną równą 1, mnożyła otrzymany w ten sposób wektor przez podaną macierz 4×4 i odrzucała ostatnią współrzędną iloczynu, która, w przypadku gdy macierz ma w ostatnim wierszu liczby $0, 0, 0, 1$ (pilnujemy, aby zawsze tak było), jest równa 1. Otrzymana trójka liczb (współrzędnych kartezjańskich punktu w układzie świata) jest zapamiętywana w odpowiednim miejscu tablicy `obj->tvert`. Jeśli punkty mają 4 współrzędne (które są współrzędnymi jednorodnymi), to wywoływana jest procedura `M4x4MultMVf`, która mnoży macierz 4×4 przez wektor w $\mathbb{R}^4$. Pominięty na listingu fragment procedury przetwarza w podobny sposób wszystkie punkty o numerach od `0` do `nv-1`.

Na listingu 13.5 są pokazane procedury budowania łańcucha. Każda z nich inicjalizuje kolejny element odpowiedniej tablicy zarezerwowanej przez procedurę `kl_NewLinkage`, przy czym procedura `kl_NewJoint` inicjalizuje dwa elementy — półkrawędzie reprezentujące jedną parę kinematyczną. Wartość powrotna każdej z tych procedur jest indeksem odpowiedniego elementu tablicy (procedura `kl_NewJoint` podaje indeks pierwszej półkrawędzi).

Procedura `kl_NewLink` rezerwuje strukturę `kl_link` i nadaje początkowe wartości domyślne wszystkim jej polom; w szczególności (w linii 10) wskaźniki, tj. indeksy, początków list półkrawędzi wychodzących z wierzchołka grafu reprezentującego człon i list referencji obiektów odpowiedniego członu otrzymują wartość `-1`, która jest interpretowana jako wskaźnik pusty.

Po utworzeniu członów i obiektów można wywoływać procedury wprowadzające referencje[8] i pary kinematyczne. Parametry procedury `kl_NewObjRef` to kolejno wskaźnik struktury łańcucha, numer członu, w którego liście ma się znaleźć tworzona referencja (numer ten musi być wartością podaną przez `kl_NewLink`), numer obiektu, którego punkty są przywiązane do członu przez tę referencję, liczba tych punktów i tablica zawierająca indeksy tych punktów w tablicy punktów obiektu. Jeśli ten ostatni parametr ma wartość `NULL`, to referencja dotyczy wszystkich punktów obiektu; w przeciwnym razie zawartość przekazanej tablicy jest przepisywana do tablicy zarezerwowanej przez procedurę `kl_NewObjRef`. Instrukcje w liniach 35–36 wstawiają nową referencję do listy referencji członu określonego przez parametr `lkn`.

Parametry procedury `kl_NewJoint` to wskaźnik struktury łańcucha, numery członów będących początkiem i końcem nowej półkrawędzi, identyfikator rodzaju pary kinematycznej (jedna ze stałych o nazwach symbolicznych `KL_ART_NONE`, ..., `KL_ART_ROT_V`) oraz indeks pierwszego parametru artykulacji wprowadzanej pary kinematycznej w tablicy parametrów łańcucha. Zgodnie z wcześniejszym stwierdzeniem procedura inicjalizuje dwa elementy w tablicy półkrawędzi łańcucha, nadając im wartości reprezentujące przeciwnie zorientowane półkrawędzie. Każda z tych półkrawędzi jest wstawiana do listy półkrawędzi wychodzących z wierzchołka będącego jej początkiem. Ponadto do tablic `Ftr`, `Rtr` i `Btr` są wpisywane współczynniki macierzy jednostkowej.

[8] Referencje dla danego obiektu może utworzyć konstruktor tego obiektu lub można je utworzyć później. Człony łańcucha muszą być utworzone przed referencjami.

Listing 13.5. Procedury `kl_NewLink`, `kl_NewObjRef` i `kl_NewJoint`

C

```
1: int kl_NewLink ( kl_linkage *linkage )
2: {
3:   int     lkn;
4:   kl_link *lk;
5:
6:   if ( linkage->nlinks < linkage->maxlinks ) {
7:     lkn = linkage->nlinks ++;
8:     lk = &linkage->link[lkn];
9:     memset ( lk, 0, sizeof(kl_link) );
10:    lk->fref = lk->fhj = -1;
11:    lk->tag = 0;
12:    return lkn;
13:  }
14:  return -1;
15: } /*kl_NewLink*/
16:
17: int kl_NewObjRef ( kl_linkage *linkage, int lkn, int on, int nv, int *vn )
18: {
19:   int        orn;
20:   kl_obj_ref *oref;
21:
22:   if ( linkage->norefs < linkage->maxorefs ) {
23:     orn = linkage->norefs ++;
24:     oref = &linkage->oref[orn];
25:     memset ( oref, 0, sizeof(kl_obj_ref) );
26:     oref->on = on;
27:     oref->nv = nv;
28:     if ( vn ) {
29:       if ( !(oref->vn = malloc ( nv*sizeof(int) )) )
30:         return -1;
31:       memcpy ( oref->vn, vn, nv*sizeof(int) );
32:     }
33:     else
34:       oref->vn = NULL;
35:     oref->nextr = linkage->link[lkn].fref;
36:     linkage->link[lkn].fref = orn;
37:     return orn;
38:   }
39:   return -1;
40: } /*kl_NewObjRef*/
41:
42: int kl_NewJoint ( kl_linkage *linkage, int l0, int l1, int art, int pnum )
43: {
44:   int          jn0, jn1;
45:   kl_halfjoint *hj0, *hj1;
```

```
46:
47:   if ( linkage->nhj < linkage->maxhj-1 &&
48:        l0 < linkage->nlinks && l1 < linkage->nlinks ) {
49:     jn0 = linkage->nhj ++;
50:     jn1 = linkage->nhj ++;
51:     hj0 = &linkage->hj[jn0];
52:     hj1 = &linkage->hj[jn1];
53:     memset ( hj0, 0, 2*sizeof(kl_halfjoint) );
54:     hj0->otherhalf = jn1;
55:     hj1->otherhalf = jn0;
56:     hj0->l0 = hj1->l1 = l0;
57:     hj0->l1 = hj1->l0 = l1;
58:     hj0->art = art;
59:     hj0->pnum = pnum;
60:     hj1->art = art == KL_ART_NONE ? KL_ART_NONE : KL_ART_OTHERHALF;
61:     hj1->pnum = -1;
62:     hj0->nexthj = linkage->link[l0].fhj;
63:     linkage->link[l0].fhj = jn0;
64:     hj1->nexthj = linkage->link[l1].fhj;
65:     linkage->link[l1].fhj = jn1;
66:     M4x4Identf ( hj0->Ftr );
67:     M4x4Identf ( hj0->Rtr );
68:     M4x4Identf ( hj0->Btr );
69:     M4x4Identf ( hj1->Ftr );
70:     M4x4Identf ( hj1->Rtr );
71:     M4x4Identf ( hj1->Btr );
72:     return jn0;
73:   }
74:   return -1;
75: } /*kl_NewJoint*/
```

W razie wywołania dowolnej z opisanych wyżej procedur zbyt wiele razy (po wywołaniu procedury `kl_NewLinkage` z parametrami o zbyt małych wartościach) procedury te przekazują wartość `-1`, co sygnalizuje niepowodzenie.

Procedury `kl_SetJointFtr` i `kl_SetJointBtr` (listing 13.6) służą do przypisania danej parze kinematycznej macierzy F_j i B_j. Parametr `jn` podaje numer j półkrawędzi, której przypisana ma być macierz o współczynnikach podanych w tablicy `tr`; numer ten był podany przez procedurę `kl_NewJoint`. Jednocześnie z przypisaniem j-tej półkrawędzi macierzy F_j, której współczynniki należy podać w tablicy `tr`, jej druga półkrawędź z pary, o numerze k, otrzymuje macierz $B_k = F_j^{-1}$. Jeśli parametr `back` nie jest zerem, to dodatkowo następują przypisania $B_j = F_j^{-1}$ i $F_k = F_j$. Służy to do osiągnięcia stanu, w którym układy członów połączonych w parę kinematyczną mają tożsame układy współrzędnych, gdy macierz $R_j = R_k^{-1}$ jest jednostkowa. Procedura `kl_SetJointBtr` działa podobnie, przypisując wskazanej przez parametr `jn` półkrawędzi macierz B_j (a także macierz $F_k = B_j^{-1}$ drugiej półkrawędzi z pary).

Listing 13.6. Procedury `kl_SetJointFtr` i `kl_SetJointBtr`

C

```
void kl_SetJointFtr ( kl_linkage *linkage, int jn,
                      GLfloat tr[16], char back )
{
  kl_halfjoint *hj, *hj1;

  if ( jn < 0 || jn >= linkage->nhj )
    return;
  hj = &linkage->hj[jn];
  hj1 = &linkage->hj[hj->otherhalf];
  memcpy ( &hj->Ftr, tr, 16*sizeof(GLfloat) );
  M4x4Invertf ( hj1->Btr, tr );
  if ( back ) {
    memcpy ( hj->Btr, hj1->Btr, 16*sizeof(GLfloat) );
    memcpy ( hj1->Ftr, hj->Ftr, 16*sizeof(GLfloat) );
  }
} /*kl_SetJointFtr*/

void kl_SetJointBtr ( kl_linkage *linkage, int jn,
                      GLfloat tr[16], char front )
{
  .... /* tu jest symetryczne odbicie treści procedury kl_SetJointFtr */
} /*kl_SetJointBtr*/
```

Procedura `kl_SetArtParam` (listing 13.7) wprowadza do tablicy `linkage->artp` parametry artykulacji. Przedstawiona w skrócie procedura pomocnicza `_kl_UpdateArtTr` oblicza na ich podstawie macierz R_j danej półkrawędzi i $R_k = R_j^{-1}$ drugiej połowy tej półkrawędzi. Przypomnę, że jedna z półkrawędzi w parze ma atrybut `art`, określający sposób artykulacji, równy `KL_ART_OTHERHALF`. Właściwy sposób artykulacji jest podany w drugiej półkrawędzi, dlatego jeśli półkrawędź określona przez parametr `jn` ma atrybut `art` równy `KL_ART_OTHERHALF`, następuje powrót — obliczenie macierzy nastąpi podczas przetwarzania drugiej półkrawędzi z pary.

Listing 13.7. Procedury `kl_SetArtParam` i `_kl_UpdateArtTr`

C

```
void kl_SetArtParam ( kl_linkage *linkage, int pno, int nump, double *par )
{
  if ( nump > 0 && pno >= 0 && pno+nump <= linkage->maxartpar ) {
    memcpy ( &linkage->artp[pno], par, nump*sizeof(double) );
    if ( pno+nump > linkage->nartpar )
      linkage->nartpar = pno+nump;
  }
} /*kl_SetArtParam*/

static char _kl_changed ( double *a, double *b, int i, int n ) { .... }

```

```
12: static void _kl_UpdateArtTr ( kl_linkage *linkage, int jn )
13: {
14:   kl_halfjoint *hj, *hj1;
15:   int          pnum;
16:   double       *artp, *prevartp;
17:
18:   artp = linkage->artp;
19:   prevartp = linkage->prevartp;
20:   hj = &linkage->hj[jn];
21:   if ( hj->art == KL_ART_OTHERHALF ) {
22:     return;
23:   hj1 = &linkage->hj[hj->otherhalf];
24:   pnum = hj->pnum;
25:   switch ( hj->art ) {
26: case KL_ART_TRANS_X:
27:     if ( _kl_changed ( artp, prevartp, pnum, 1 ) ) {
28:       M4x4Translatef ( hj->Rtr, artp[pnum], 0.0, 0.0 );
29:       M4x4Translatef ( hj1->Rtr, -artp[pnum], 0.0, 0.0 );
30:     }
31:     break;
32:     .... /* tu inne rodzaje par kinematycznych */
33: case KL_ART_ROT_V:
34:     if ( _kl_changed ( artp, prevartp, pnum, 4 ) ) {
35:       M4x4RotateVf ( hj->Rtr, artp[pnum], artp[pnum+1], artp[pnum+2],
36:                      artp[pnum+3] );
37:       M4x4Transposef ( hj1->Rtr, hj->Rtr );
38:     }
39:     break;
40: default:
41:     M4x4Identf ( hj->Rtr );
42:     M4x4Identf ( hj1->Rtr );
43:     break;
44:   }
45: } /*_kl_UpdateArtTr*/
```

W instrukcji przełącznika (linie 25–44) następuje wybór sposobu tworzenia macierzy R_j i R_k; spośród dwunastu sposobów odpowiadających identyfikatorom z listingu 13.1 są pokazane dwa: przesunięcie wzdłuż osi x i obrót wokół osi określonej przez wektor $\boldsymbol{v}$. W pierwszym przypadku przesunięcie zależy od jednego parametru, o numerze `pnum`; macierz przesunięcia i jej odwrotność są tworzone przez procedurę `M4x4Translatef` (opisaną w rozdziale 5). W drugim przypadku macierz jest określona przez cztery parametry artykulacji, zajmujące kolejne miejsca w tablicy od numeru `pnum`; są to trzy współrzędne wektora $\boldsymbol{v}$ i kąt φ_j obrotu. Odwrotności macierzy ortogonalnych (reprezentujących obroty) są transpozycjami tych macierzy, zatem użycie procedury `M4x4Transposef` jest najprostszym (i najszybszym) sposobem znalezienia odwrotności macierzy obrotu. Pomocnicza procedura `_kl_changed` porównuje zawartości wskazanych fragmentów tablic `artp` i `prevartp`

i przekazuje 0 (`false`), gdy te są identyczne; w takim przypadku obliczenie macierzy zostaje pominięte, bo zakłada się, że macierze te są niezmienione od poprzedniego obliczenia (po uaktualnieniu wszystkich macierzy R_j parametry artykulacji są kopiowane do tablicy `prevartp`).

Listing 13.8 przedstawia procedurę `kl_Articulate`, którą należy wywołać po nadaniu nowej wartości co najmniej jednemu parametrowi artykulacji, przed narysowaniem sceny (tj. obiektów łańcucha) w nowym położeniu. Celem tej procedury jest wyznaczenie, dla każdego członu łańcucha, macierzy A_i przejścia do układu świata, a następnie, dla każdej referencji obiektu w liście tego członu, obliczenie iloczynu A_iE_l (gdzie macierz E_l jest określona dla obiektu, którego punkty są wymienione w tej referencji) i wywołanie metody `transform`, której zadaniem jest przekształcenie punktów do układu świata (domyślna metoda to pokazana na listingu 13.4 procedura `kl_DefaultTransform`; w aplikacjach opisanych w tej książce użyjemy innych metod).

Pętla w liniach 36–37 wywołuje procedurę z listingu 13.7 w celu skonstruowania macierzy R_j dla wszystkich par kinematycznych na podstawie bieżących wartości parametrów artykulacji. Po zakończeniu tego obliczenia parametry te są kopiowane do tablicy `prevartp`, w opisanym wcześniej celu.

W linii 40 polom `tag` wszystkich członów przypisywana jest wartość 0, która oznacza człony — wierzchołki grafu — jako nieodwiedzone. W linii 41 jest wywoływana procedura `_kl_rArticulate`, która przeszukuje graf łańcucha metodą DFS, zaczynając od wierzchołka będącego bieżącym korzeniem. Parametr `lkn` tej procedury jest numerem przetwarzanego wierzchołka, a parametr `tr` jest tablicą zawierającą współczynniki macierzy A_i, która jest iloczynem macierzy $A_kF_{j_1}R_{j_1}B_{j_1}\cdot\ldots\cdot F_{j_m}R_{j_m}B_{j_m}$. Macierz A_k opisuje przejście od układu korzenia do układu świata, a kolejne czynniki to odpowiednie macierze krawędzi w ścieżce od korzenia do bieżącego (i-tego) wierzchołka.

W liniach 43–44 dla każdego obiektu jest wywoływana metoda `postprocess`, której zadaniem jest wykonanie dodatkowego obliczenia dla każdego obiektu, jeśli tylko takie obliczenie jest potrzebne[9].

Procedura `_kl_rArticulate` w linii 11 zaznacza, że przetwarzany przez nią wierzchołek grafu (człon łańcucha) jest już odwiedzony. W pętli w liniach 15–19 przeglądana jest lista referencji obiektów tego członu. Dla każdej referencji jest (w linii 17) obliczany iloczyn A_iE_l, po czym jest wywoływana metoda `transform` obiektu z parametrami opisującymi obiekt, numer referencji, macierz A_iE_l, liczbę wierzchołków i tablicę indeksów wierzchołków obiektu w tej referencji. W pętli w liniach 20–28 jest przeglądana lista półkrawędzi wychodzących z wierzchołka; dla każdej z tych półkrawędzi, które kończą się w nieodwiedzonym wierz-

[9]Na przykład, po przekształceniu punktów kontrolnych powierzchni B-sklejanej może być potrzebne podzielenie tej powierzchni na płaty Béziera (zobacz podrozdz. 15.1 i B.1). W aplikacji drugiej H metoda `postprocess` wywołuje szader obliczeniowy, który oblicza współrzędne punktów kontrolnych przekształconego czajnika i torusa w układzie świata na podstawie danych w pamięci GPU — szader ten przejął rolę procedury `kl_DefaultTransform` z listingu 13.4. W aplikacji trzeciej B metoda `postprocess` wywoła najpierw szader obliczeniowy, który przekształci wierzchołki siatki będącej modelem dłoni, a następnie za pomocą innego szadera obliczeniowego dokona zagęszczania tej siatki, aby otrzymać gładką powierzchnię dłoni, gotową do narysowania.

Listing 13.8. Procedura `kl_Articulate`

C

```
1: static void _kl_rArticulate ( kl_linkage *linkage, int lkn, GLfloat tr[16] )
2: {
3:   kl_link      *lk;
4:   GLfloat      t0[16], t1[16];
5:   int          r, on, j, l1;
6:   kl_object    *obj;
7:   kl_obj_ref   *oref;
8:   kl_halfjoint *hj;
9:
10:  lk = &linkage->link[lkn];
11:  lk->tag = 1;
12:  obj = linkage->obj;
13:  oref = linkage->oref;
14:  hj = linkage->hj;
15:  for ( r = lk->fref; r >= 0; r = oref[r].nextr ) {
16:    on = oref[r].on;
17:    M4x4Multf ( t1, tr, obj[on].Etr );
18:    obj[on].transform ( linkage, &obj[on], r, t1, oref[r].nv, oref[r].vn, );
19:  }
20:  for ( j = lk->fhj; j >= 0; j = hj[j].nexthj ) {
21:    l1 = hj[j].l1;
22:    if ( !linkage->link[l1].tag ) {
23:      M4x4Multf ( t0, tr, hj[j].Ftr );
24:      M4x4Multf ( t1, t0, hj[j].Rtr );
25:      M4x4Multf ( t0, t1, hj[j].Btr );
26:      _kl_rArticulate ( linkage, l1, t0 );
27:    }
28:  }
29: } /*_kl_rArticulate*/
30:
31: void kl_Articulate ( kl_linkage *linkage )
32: {
33:  int       i;
34:  kl_object *obj;
35:
36:  for ( i = 0; i < linkage->nhj; i++ )
37:    _kl_UpdateArtTr ( linkage, i );
38:  memcpy ( linkage->prevartp, linkage->artp,
39:           linkage->maxartpar*sizeof(double) );
40:  for ( i = 0; i < linkage->nlinks; i++ ) linkage->link[i].tag = 0;
41:  _kl_rArticulate ( linkage, linkage->current_root,
42:                    linkage->current_root_tr );
43:  for ( i = 0; i < linkage->nobj; i++ )
44:    linkage->obj[i].postprocess ( linkage, &linkage->obj[i] );
45: } /*kl_Articulate*/
```

chołku, macierz A_i jest mnożona z prawej strony przez macierze F_j, R_j i B_j tej półkrawędzi i następuje rekurencyjne wywołanie procedury dla wierzchołka końcowego półkrawędzi.

Z powyższego opisu wynika, że graf łańcucha kinematycznego musi być spójny. W przeciwnym razie człony, tj. wierzchołki, do których nie da się dojść z korzenia, nie będą miały obliczonej macierzy A_i.

Procedura `kl_Redraw` pokazana na listingu 13.9 kolejno dla każdego obiektu wywołuje jego metodę `redraw`.

Listing 13.9. Procedura `kl_Redraw`

C

```
1: void kl_Redraw ( kl_linkage *linkage )
2: {
3:   int i;
4:
5:   for ( i = 0; i < linkage->nobj; i++ )
6:     linkage->obj[i].redraw ( linkage, &linkage->obj[i] );
7: } /*kl_Redraw*/
```

13.3. Konstruowanie łańcucha kinematycznego aplikacji

Pokazane na listingu 13.10 makrodefinicje określają liczby elementów tworzonego przez aplikację 1E łańcucha kinematycznego: liczbę członów, liczbę par kinematycznych i liczbę obiektów i referencji obiektów. Kolejne makrodefinicje opisują promienie Słońca, Ziemi i Księżyca oraz promienie orbit Ziemi wokół Słońca i Księżyca wokół Ziemi. Oczywiście, proporcje zapisanych liczb nie odpowiadają rzeczywistości, ale nie byłoby praktyczne dokładne odtwarzanie wymiarów wziętych z Kosmosu, gdzie na przykład promień pierwszej orbity jest około 390 razy większy niż promień tej drugiej.

W ostatnich czterech makrodefinicjach są zapisane stałe czasowe: okresy obiegu orbit lub okresy obrotów własnych, mierzone w dobach ziemskich. W szczególności jeden pełny obrót Słońca wokół własnej osi zajmuje 27 dób.

Listing 13.10. Makrodefinicje i definicja typu struktury pomocniczej

C

```
1: #define NLINKS   6
2: #define NJOINTS  5
3: #define NOBJECTS 3
4: #define NOBJREFS 3
5:
6: #define RS  0.2              /* promień Słońca */
7: #define RE  0.05             /* promień Ziemi */
8: #define RM  0.02             /* promień Księżyca */
9: #define ROE 1.0              /* promień orbity Ziemi */
10: #define ROM 0.15            /* promień orbity Księżyca */
```

```
11:
12: #define TS   27.0            /* czas jednego obrotu Słońca */
13: #define TEY 365.25           /* rok ziemski */
14: #define TE  ((TEY-1.0)/TEY)  /* czas jednego obrotu Ziemi */
15: #define TM  (28.0*TE)        /* czas obiegu Ziemi przez Księżyc */
16:
17: typedef struct my_object {
18:     GLfloat mtr[16];
19:     GLfloat colour[3], tl;
20:     GLuint  cs;
21:     int     prog;
22:   } my_object;
```

Struktury typu `my_object` są używane do przechowywania informacji potrzebnych podczas wykonywania obrazów. Dla każdego obiektu jest używana jedna taka struktura, przechowująca m.in. obliczoną przez procedurę `kl_Articulate` macierz przejścia od układu modelu do układu świata.

Listing 13.11 przedstawia procedury — metody wirtualne obiektów łańcucha i procedurę, która ten łańcuch buduje. Zadaniem procedury `KLTransform`, która jest metodą wirtualną `transform` obiektu, jest zapamiętanie macierzy przejścia do układu świata w polu struktury typu `my_object` wskazywanej przez pole `usrdata` obiektu. Wierzchołki obiektu nie są tu przekształcane, bo to nastąpi podczas rysowania obiektu. Do rysowania służy metoda `KLRedraw`, która przesyła macierz przekształcenia modelu do pamięci GPU, a następnie wywołuje procedurę rysowania dwudziestościanu z parametrami wziętymi ze struktury `my_object`: numerem programu szaderów, który ma być użyty, kolorem, sposobem wybierania koloru i stopniem rozdrobnienia trójkątnych ścian dwudziestościanu.

Budowanie łańcucha wykonuje procedura `ConstructMyLinkage`, zaczynając od wywołania procedury `kl_NewLinkage`. Jej parametry określają liczby elementów łańcucha, przy czym ostatni parametr wskazuje, że aplikacja nie przywiązuje swoich danych do całego łańcucha. Pętla w liniach 40–41 tworzy człony łańcucha. W liniach 42–50 tworzone są obiekty; każde wywołanie procedury `kl_NewObject` jest poprzedzone utworzeniem odpowiedniej macierzy wstępnego przekształcenia, która skaluje kulę do odpowiedniej wielkości. Macierze te oraz wskaźniki struktur `my_object` znajdujących się w zadeklarowanej w linii 1 tablicy są zapamiętywane w strukturach obiektów, razem z dwiema metodami wirtualnymi dostarczonymi przez aplikację i dwiema metodami (`postprocess` i `destroy`) domyślnymi.

Konstruktor obiektu, tj. procedura `KLInit`, ma za zadanie utworzyć referencję dla tego obiektu. Numer obiektu, 0, 1 lub 2, jest obliczany w linii 8. Referencja jest tworzona w linii 9, przy czym numer członu, do którego obiekt jest dołączany, jest brany z tablicy `lkn`. Referencja ma 0 wierzchołków, bo przekształcanie obiektów będzie wykonywać GPU.

Aby obiekt był poddany właściwemu przekształceniu, metoda `transform` obiektu, czyli procedura `KLTransform`, zapamiętuje w strukturze `my_object` macierz przekształcenia modelu obliczoną przez procedurę artykulacji. Metoda `redraw`, tj. procedura `KLRedraw`, przesyła tę macierz do bloku zmiennych jednolitych `TransBlock`, po czym wywołuje procedurę rysowania obiektu, tj. dwudziestościanu, który szadery „przerobią” na sferę.

Listing 13.11. Procedury konstrukcji łańcucha kinematycznego

C

```
my_object sunplanete[NOBJECTS];

static char KLInit ( kl_linkage *lkg, kl_object *obj )
{
  static const int lkn[3] = {1, 4, 5};
  int on;

  on = obj - lkg->obj;
  kl_NewObjRef ( lkg, lkn[on], on, 0, NULL );
  return true;
} /*KLInit*/

static void KLTransform ( kl_linkage *lkg, kl_object *obj,
                          int refn, GLfloat tr[16], int nv, int *vn )
{
  my_object *sp;

  sp = (my_object*)obj->usrdata;
  memcpy ( sp->mtr, tr, 16*sizeof(GLfloat) );
} /*KLTransform*/

static void KLRedraw ( kl_linkage *lkg, kl_object *obj )
{
  my_object *sp;

  sp = (my_object*)obj->usrdata;
  LoadMMatrix ( &trans, sp->mtr );
  DrawTessIcos ( sp->prog, sp->tl, sp->cs, sp->colour );
} /*KLRedraw*/

kl_linkage *ConstructMyLinkage ( void )
{
  kl_linkage *lkg;
  int        l[NLINKS], j[NJOINTS];
  GLfloat    tr[16];
  int        i;

  if ( (lkg = kl_NewLinkage ( NOBJECTS, NLINKS, NOBJREFS,
                              NJOINTS, NJOINTS, NULL )) ) {
    for ( i = 0; i < NLINKS; i++ )
      l[i] = kl_NewLink ( lkg );
    M4x4Scalef ( tr, RS, RS, RS );
    kl_NewObject ( lkg, 0, 3, 0, tr, (void*)&sunplanete[0],
                   KLInit, KLTransform, NULL, KLRedraw, NULL );
    M4x4Scalef ( tr, RE, RE, RE );
```

```
46:     kl_NewObject ( lkg, 0, 3, 0, tr, (void*)&sunplanete[1],
47:                    KLInit, KLTransform, NULL, KLRedraw, NULL );
48:     M4x4Scalef ( tr, RM, RM, RM );
49:     kl_NewObject ( lkg, 0, 3, 0, tr, (void*)&sunplanete[2],
50:                    KLInit, KLTransform, NULL, KLRedraw, NULL );
51:     j[0] = kl_NewJoint ( lkg, l[0], l[1], KL_ART_ROT_Z, 0 );
52:     j[1] = kl_NewJoint ( lkg, l[0], l[2], KL_ART_ROT_Z, 1 );
53:     j[2] = kl_NewJoint ( lkg, l[2], l[3], KL_ART_ROT_Z, 2 );
54:     j[3] = kl_NewJoint ( lkg, l[3], l[4], KL_ART_ROT_Z, 3 );
55:     j[4] = kl_NewJoint ( lkg, l[3], l[5], KL_ART_ROT_Z, 4 );
56:     M4x4Translatef ( tr, ROE, 0.0, 0.0 );
57:     kl_SetJointBtr ( lkg, j[1], tr, false );
58:     M4x4RotateXf ( tr, -23.5*PI/180.0 );
59:     kl_SetJointFtr ( lkg, j[3], tr, true );
60:     M4x4RotateYf ( tr, 5.15*PI/180.0 );
61:     kl_SetJointFtr ( lkg, j[4], tr, false );
62:     M4x4Translatef ( tr, ROM, 0.0, 0.0 );
63:     kl_SetJointBtr ( lkg, j[4], tr, false );
64:   }
65:   return lkg;
66: } /*ConstructMyLinkage*/
```

Wszystkie pięć par kinematycznych, tworzonych w liniach 51–55, opisuje obroty wokół osi z lokalnych układów współrzędnych. Początki tych układów i kierunki osi, wokół których następuje obracanie, są wyznaczone przez macierze F_j i B_j, konstruowane i przypisywane parom kinematycznym w liniach 56–63. Macierze F_0 i B_0 (domyślnie przypisane przez procedurę `kl_NewJoint`) opisują przekształcenie tożsamościowe, wskutek czego Słońce obraca się wokół osi z układu świata. Macierz F_1 też jest jednostkowa, ale macierz B_1 opisuje przesunięcie Ziemi na odległość promienia ziemskiej orbity. Dzięki temu animacja parametru φ_1, który jest kątem obrotu środka Ziemi wokół Słońca, powoduje ruch tego środka po orbicie.

Parametr φ_2 opisuje obrót, którego celem jest zachowanie kierunku w przestrzeni osi ruchu obrotowego Ziemi (tj. prostej przechodzącej przez bieguny). Jak wiadomo, oś ta jest nachylona względem prostej prostopadłej do płaszczyzny ekliptyki pod kątem $23^\circ 30'$, przy czym oś obrotu Ziemi stale celuje w okolice Gwiazdy Polarnej. Bez wprowadzenia pary kinematycznej J_2, realizującej obracanie członu L_3 względem L_2 z prędkością kątową, z jaką Ziemia obiega Słońce, ale w przeciwną stronę, prosta przechodząca przez bieguny zakreślałaby w przestrzeni cały stożek w czasie jednego roku[10].

Parametr φ_3 odpowiada za obracanie się Ziemi wokół osi przechodzącej przez bieguny. Kąt nachylenia tej osi (tzw. **kąt nutacji**) jest ustalany w linii 58, w której jest konstruowana macierz F_3. Wywołana w linii 59 procedura zapamiętuje tę macierz dla pary kinematycznej J_3, jednocześnie przyjmując dla tej pary macierz $B_3 = F_3^{-1}$.

Parametr φ_4 służy do realizacji ruchu Księżyca wokół Ziemi. Płaszczyzna orbity Księżyca jest nachylona względem płaszczyzny ekliptyki pod kątem $5^\circ 9'$; odpowiednie przekształce-

[10] W istocie oś obrotu Ziemi podlega precesji, czyli zakreśla taki stożek, co zajmuje jeden **rok platoński**, trwający około 25700 lat. Tego ruchu w aplikacji 1E nie uwzględniłem, choć byłoby to łatwe.

nie (macierz F_4) jest konstruowane w linii 60. Macierz B_4 konstruowana w linii 62 opisuje przekształcenie odsuwające Księżyc od Ziemi na odległość promienia jego orbity. Ponieważ Księżyc, obiegając Ziemię, obraca się tak, że tylko jedna jego strona jest z Ziemi widoczna, wprowadzenie do łańcucha dodatkowej pary kinematycznej realizującej własny ruch obrotowy Księżyca było zbędne.

Listing 13.12 przedstawia procedurę, która na podstawie odczytu zegara, a dokładniej czasu, który upłynął od poprzedniego wywołania procedury `TimerTocTic` i wartości zmiennej `speed` określającej szybkość animacji, oblicza fazę ruchu i nadaje wartości parametrom artykulacji, a na końcu wywołuje procedurę obliczającą macierze przekształceń (przejść do układu świata) dla wszystkich obiektów. Makrodefinicja `Mod2PI` odejmuje od swojego argumentu taką całkowitą wielokrotność liczby π, aby otrzymać liczbę z przedziału $[0, 2\pi)$.

Listing 13.12. Procedura artykulacji

C

```
1: #define MAX_SPEED 100.0  /* prędkość maksymalna */
2: #define NOM_SPEED  10.0  /* prędkość początkowa */
3: #define MIN_SPEED   1.0  /* prędkość minimalna */
4:
5: double speed = NOM_SPEED, phase = 0.0;
6:
7: void ArticulateMyLinkage ( void )
8: {
9:   phase += speed * TimerTocTic ();
10:  mylinkage->artp[0] = Mod2PI ( phase/TS );
11:  mylinkage->artp[2] = -(mylinkage->artp[1] = Mod2PI ( phase/TEY ));
12:  mylinkage->artp[3] = Mod2PI ( phase/TE );
13:  mylinkage->artp[4] = Mod2PI ( phase/TM );
14:  kl_Articulate ( mylinkage );
15: } /*ArticulateMyLinkage*/
```

Okresy poszczególnych obrotów są podane w makrodefinicjach na listingu 13.10. Zwróćmy uwagę, że choć doba trwa jedną dobę, tj. 24 godziny, okres obrotu Ziemi wokół własnej osi jest trochę krótszy, bo z powodu ruchu Ziemi na orbicie wokół Słońca w ciągu jednego roku Słońce o jeden raz mniej niż inne gwiazdy okrąża obserwatora stojącego na Ziemi.

13.4. Szadery

Do wykonywania obrazów aplikacja używa dwóch programów szaderów, z których pierwszy służy do rysowania Słońca, a drugi do rysowania Ziemi i Księżyca. Oba programy składają się z pięciu szaderów, przy czym pierwsze cztery są w obu programach te same.

Programy zawierają szader wierzchołków z listingu 12.1. Szader sterowania rozdrabnianiem (listing 13.13) powstał przez modyfikację szadera z listingu 12.4, polegającą na wprowadzeniu zmiennej jednolitej `tl`, której wartość zostaje przypisana elementom tablic `gl_TessLevelOuter` i `gl_TessLevelInner`. Z uwagi na wielkość, przerabiając dwudziestościan na

Słońce, będziemy rozdrabniać trójkątne ściany na większą liczbę kawałków, niż tego wymaga otrzymanie Ziemi i Księżyca.

Listing 13.13. Szader sterowania rozdrabnianiem aplikacji 1E

GLSL

```
#version 420

layout(vertices=3) out;

in vec3 Colour[];
out vec3 TCColour[];

uniform float tl;

void main ( void )
{
  if ( gl_InvocationID == 0 ) {
    gl_TessLevelOuter[0] = gl_TessLevelOuter[1] = gl_TessLevelOuter[2] = tl;
    gl_TessLevelInner[0] = tl;
  }
  gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
  TCColour[gl_InvocationID] = Colour[gl_InvocationID];
} /*main*/
```

Szader rozdrabniania (listing 13.14) działa podobnie do tego z listingu 12.6, ale przekazywany przez niego na wyjście kolor wierzchołka zależy od wartości zmiennej jednolitej `cs`. Wartość 1 powoduje przekazaanie koloru otrzymanego przez interpolację kolorów wierzchołków trójkąta, a każda inna wartość wybiera kolor podany w zmiennej jednolitej `colour`. W ten sposób Ziemia będzie pstrokatą kulą, taką jak w aplikacji 1D, a Księżyc będzie odpowiednio srebrny.

W obu programach jest używany szader geometrii z listingu 12.7. Szader fragmentów w programie używanym do rysowania Ziemi i Księżyca jest wzięty bez zmian z aplikacji 1D (listing 12.8).

Listing 13.14. Szader rozdrabniania aplikacji 1E

GLSL

```
#version 420

layout(triangles,equal_spacing,ccw) in;

in vec3 TCColour[];
out FVertex { .... } Out;

uniform TransBlock { .... } trb;

uniform uint cs;
uniform vec3 colour;
```

```
12:
13: void main ( void )
14: {
15:   float s, t, u;
16:   vec4  vert, vv;
17:
18:   .... /* linie 14-17 szadera z listingu 12.5 */
19:   vv = trb.mm * vert;
20:   gl_Position = trb.vpm * vv;
21:   switch ( cs ) {
22: case 1:
23:     Out.Colour = colour;
24:     break;
25: default:
26:     Out.Colour = s*TCColour[0] + t*TCColour[1] + u*TCColour[2];
27:     break;
28:   }
29:   Out.Position = vv.xyz/vv.w;
30:   Out.Normal = normalize ( mat3(trb.mmti) * vert.xyz );
31: } /*main*/
```

Szader fragmentów pokazany na listingu 13.15 przetwarza kolor otrzymany na wejściu na kolor bliższy żółtego, czyli koloru światła słonecznego. Samo będąc źródłem światła, Słońce nie powinno być traktowane jak obiekt odbijający światło pochodzące z innych źródeł. Czynnik `f`, obliczony w linii 16, powoduje nadanie trochę ciemniejszych barw pikselom w pobliżu brzegu tarczy słonecznej, dzięki czemu obraz Słońca nabiera odrobinę plastyczności.

Listing 13.15. Szader fragmentów do rysowania Słońca

GLSL

```
1: #version 420
2:
3: in FVertex { .... } In;
4: out vec4 out_Colour;
5:
6: uniform TransBlock { .... } trb;
7:
8: #define AGamma(colour) pow ( colour, vec3(256.0/563.0) )
9:
10: void main ( void )
11: {
12:   vec3  d;
13:   float r, g, b, f;
14:
15:   d = normalize ( trb.eyepos.xyz - In.Position );
16:   f = 0.25+0.75*dot ( d, normalize ( In.Normal ) );
17:   r = 0.9+0.1*sin ( 100.0*In.Colour.r*In.Colour.g );
18:   g = 1.5*r-0.5;
```

```
  b = 0.5+0.1*cos ( 20.0*(In.Colour.b+In.Colour.r) );
  r = min ( 2.0*r, 1.0 );  g = min ( 2.0*g, 1.0 );
  out_Colour = vec4 ( AGamma ( f*vec3(r, g, b) ), 1.0 );
} /*main*/
```

13.5. Pozostałe zmiany w aplikacji

Procedura kompilacji szaderów pokazana na listingu 13.16 posługuje się opisanymi wcześniej procedurami OpenGL-a i procedurami pomocniczymi, uzyskującymi dostęp do zmiennych jednolitych w blokach opisujących przekształcenia i źródła światła, a także zmiennych jednolitych w domyślnych blokach obu programów. Pierwszy z tych programów, przeznaczony do rysowania Słońca, nie korzysta z opisów źródeł światła. Zmienna jednolita `tl` szadera sterowania rozdrabnianiem jest obecna w obu programach, dlatego jej położenie jest odczytywane z obu programów i zapamiętywane w osobnych zmiennych `tl0loc` i `tl1loc`. Zmienne jednolite `cs` i `colour` też występują w obu programach, ale ich wartości są istotne tylko w drugim programie, zatem ich położenia są odczytywane tylko z niego.

Listing 13.16. Procedura kompilacji szaderów

C

```
GLuint  program_id[2];
GLint   tl0loc, tl1loc, csloc, colourloc;
TransBl trans;
LightBl light;

void LoadMyShaders ( void )
{
  static const char *filename[6] =
    { "app1e.vert.glsl", "app1e.tesc.glsl", "app1e.tese.glsl",
      "app1e.geom.glsl", "app1e1.frag.glsl", "app1e2.frag.glsl" };
  static const GLuint shtype[6] =
    { GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER,
      GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER, GL_FRAGMENT_SHADER };
  static const GLchar *uvnames[3] = { "tl", "cs", "colour" };
  GLuint  shader_id[6], sh[5];
  int    i;

  for ( i = 0; i < 6; i++ )
    shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
  program_id[0] = LinkShaderProgram ( 5, shader_id, "0" );
  memcpy ( sh, shader_id, 4*sizeof(GLuint) );
  sh[4] = shader_id[5];
  program_id[1] = LinkShaderProgram ( 5, sh, "1" );
  tl0loc = glGetUniformLocation ( program_id[0], uvnames[0] );
  tl1loc = glGetUniformLocation ( program_id[1], uvnames[0] );
```

```
26:   csloc = glGetUniformLocation ( program_id[1], uvnames[1] );
27:   colourloc = glGetUniformLocation ( program_id[1], uvnames[2] );
28:   GetAccessToTransBlockUniform ( program_id[1] );
29:   GetAccessToLightBlockUniform ( program_id[1] );
30:   AttachUniformTransBlockToBP ( program_id[0] );
31:   for ( i = 0; i < 6; i++ )
32:     glDeleteShader ( shader_id[i] );
33:   ExitIfGLError ( "LoadMyShaders" );
34: } /*LoadMyShaders*/
```

Sposób inicjalizacji źródła światła jest taki sam jak w aplikacji 1B, inne są tylko parametry tego źródła. Do procedury inicjalizacji obiektów, pokazanej na listingu 13.17 zostało dodane wywołanie procedury konstrukcji łańcucha kinematycznego, a po nim instrukcje przypisujące odpowiednie wartości polom struktur typu `my_object` dla poszczególnych obiektów. Drugi parametr pomocniczej procedury `SetSunPlanete` jest indeksem do tablicy, w której są zapamiętane identyfikatory programów szaderów; Słońce ma być rysowane przy użyciu programu `program_id[0]`, a Ziemia i Księżyc przy użyciu programu `program_id[1]`. Stopnie rozdrobnienia trójkątnych płatów w celu narysowania Słońca, Ziemi i Księżyca ustaliłem odpowiednio na 10, 5 i 4. Dla Słońca i Ziemi kolor wierzchołków ma być brany z wejścia szadera wierzchołków, a dla Księżyca ze zmiennej jednolitej `colour`. Na zakończenie jest wywoływana procedura `ArticulateMyLinkage` z listingu 13.12 i łańcuch jest gotowy do pierwszego rysowania.

Listing 13.17. Inicjalizacja obiektów

C

```
1: kl_linkage *mylinkage;
2:
3: void InitLights ( void )
4: {
5:   GLfloat amb0[3] = { 0.005, 0.005, 0.005 };
6:   GLfloat dir0[3] = { 1.0, 1.0, 1.0 };
7:   GLfloat pos0[4] = { 0.0, 0.0, 0.0, 1.0 };
8:   GLfloat atn0[3] = { 1.0, 0.0, 0.0 };
9:
10:  .... /* tu niezmienione linie 19-23 z listingu 10.11 */
11: } /*InitLights*/
12:
13: void SetSunPlanete ( my_object *sp, GLuint prog_id, GLfloat tl,
14:                       GLfloat colour[3], GLuint cs )
15: {
16:   sp->prog = prog_id;
17:   sp->tl = tl;
18:   memcpy ( sp->colour, colour, 3*sizeof(GLfloat) );
19:   sp->cs = cs;
20: } /*SetSunPlanete*/
21:
```

```
22: void InitMyWorld ( int argc, char *argv[], int width, int height )
23: {
24:   static GLfloat fg[4] = { 0.0, 0.0, 1.0, 1.0 };
25:   static GLfloat bk[4] = { 0.0, 0.0, 0.0, 0.0 };
26:   static GLfloat colour[3] = { 1.0, 1.0, 1.0 };
27:
28:   LoadMyShaders ();
29:   .... /* tworzenie buforów dla bloków zmiennych jednolitych bez zmian */
30:   LoadTextShaders ();
31:   TimerInit ();
32:   .... /* utworzenie fontów i dwudziestościanu bez zmian */
33:   InitLights ();
34:   ResizeMyWorld ( width, height );
35:   if ( (mylinkage = ConstructMyLinkage ()) ) {
36:     SetSunPlanete ( &sunplanete[0], 0, 10, colour, 0 );
37:     SetSunPlanete ( &sunplanete[1], 1, 5, colour, 0 );
38:     SetSunPlanete ( &sunplanete[2], 1, 4, colour, 1 );
39:     ArticulateMyLinkage ();
40:   }
41:   else
42:     ExitOnError ( "InitMyWorld" );
43: } /*InitMyWorld*/
```

Listing 13.18. Procedura rysująca obiekty

C

```
1: void DrawTessIcos ( int prog, GLfloat tl, GLuint cs, GLfloat colour[3] )
2: {
3:   glBindVertexArray ( icos_vao );
4:   glUseProgram ( program_id[prog] );
5:   switch ( prog ) {
6: case 0:
7:     glUniform1f ( tl0loc, tl );
8:     break;
9: case 1:
10:    glUniform1f ( tl1loc, tl );
11:    glUniform1ui ( csloc, cs );
12:    glUniform3fv ( colourloc, 1, colour );
13:    break;
14:  }
15:  glPatchParameteri ( GL_PATCH_VERTICES, 3 );
16:  glDrawElements ( GL_PATCHES, 60, GL_UNSIGNED_BYTE,
17:                   (GLvoid*)(36*sizeof(GLubyte)) );
18:  glBindVertexArray ( 0 );
19:  ExitIfGLError ( "DrawTessIcos" );
20: } /*DrawTessIcos*/
```

Dostosowanie procedury `DrawTessIcos`, wyświetlającej każde z ciał niebieskich, polegało na dodaniu parametru sterującego wyborem programu szaderów i parametrów, których wartości są przypisywane zmiennym jednolitym w wybranym programie. Procedura `RedrawMyWorld`, której nie zamieściłem na listingu, po skasowaniu tła (w kolorze starannie dobranym do tematu tej aplikacji) i pozostałych przygotowaniach takich jak we wcześniejszych wersjach wywołuje procedurę `kl_Redraw`, aby narysować wszystkie obiekty.

Do interakcji z aplikacją 1E została dodana jeszcze jedna możliwość: wykonywania najazdów i odjazdów kamery, przez zmienianie kąta między górną a dolną ścianą ostrosłupa widzenia. Odpowiada to zmienianiu długości ogniskowej „obiektywu", przy użyciu którego powstają obrazy. Użytkownik może używać do tego rolki myszy; obracanie jej do siebie powoduje zmniejszane tego kąta, co powiększa obraz obiektów, obracanie w drugą stronę ma działanie przeciwne.

Listing 13.19 przedstawia zmienione i nowe procedury „graficzne". Nowa zmienna `zoom` o początkowej wartości 1, jest dodatkowym czynnikiem we wzorach używanych przez procedurę `ResizeMyWorld` do obliczenia parametrów l, r, b i t ostrosłupa widzenia (zobacz podrozdz. 6.2). Procedura `ChangeZoom` sygnalizuje, że dokonała zmiany ostrosłupa widzenia (tj. przypisała nową wartość zmiennej `zoom`), przekazując wartość powrotną `true`, aby spowodować wykonanie nowego obrazu.

Najazdy i odjazdy kamery powodują odpowiednio zwiększanie i zmniejszanie wartości zmiennej `zoom` o ustalony czynnik, 1.05. Najprościej byłoby zmienną `zoom` mnożyć lub dzielić przez ten czynnik, ale kumulacja błędów zaokrągleń może spowodować brak możliwości powrotu do wartości początkowej[11]. Problem jest rozwiązany za pomocą zmiennej `zoomexp` typu `int`, która jest licznikiem wykonanych kroków najazdu i odjazdu; zmiany jej wartości są wolne od błędów zaokrągleń, a wartość zmiennej zoom jest obliczana za pomocą funkcji `pow`, co gwarantuje doskonałą powtarzalność nastawień długości ogniskowej kamery.

Listing 13.19. Obsługa najazdów i odjazdów kamery

C

```
#define MIN_ZOOM 0.2
#define MAX_ZOOM 5.0
#define ZOOM_FCT 1.05

int    zoomexp = 0;
double zoom = 1.0;

void ResizeMyWorld ( int width, int height )
{
  float lr, zz;

  glViewport ( 0, 0, win_width = width, win_height = height );
  SetupTextFrame ( width, height );
  zz = 0.5533*zoom;  lr = zz*(float)width/(float)height;
```

[11] W tym zastosowaniu byłoby to całkowicie nieszkodliwe, ale jeśli taki błąd jest łatwy do usunięcia, to ja nie mam żadnych wątpliwości, co należy zrobić.

```
15:   M4x4Frustumf ( trans.pm, NULL, left = -lr, right = lr,
16:                  bottom = -zz, top = zz, near = 5.0, far = 15.0 );
17:   LoadVPMatrix ( &trans );
18:   NotifyViewerPos ();
19: } /*ResizeMyWorld*/
20:
21: char ChangeZoom ( char inczoom )
22: {
23:   int   e;
24:   double z;
25:
26:   e = zoomexp;
27:   if ( inczoom ) {
28:     if ( (z = pow ( ZOOM_FCT, (double)(++e) )) > MAX_ZOOM )
29:       return false;
30:   }
31:   else {
32:     if ( (z = pow ( ZOOM_FCT, (double)(--e) )) < MIN_ZOOM )
33:       return false;
34:   }
35:   zoomexp = e;
36:   zoom = z;
37:   ResizeMyWorld ( win_width, win_height );
38:   return true;
39: } /*ChangeZoom*/
```

Procedura ChangeZoom jest wywoływana przez pokazaną na listingu 13.20 procedurę MouseFunc, która poza tym wciąż spełnia swoją rolę w obracaniu obserwatora wokół sceny. Procedura ta jest wywoływana po obróceniu rolki myszy. Podawane wtedy numery przycisków nie mają w pliku nagłówkowym freeglut.h (nadanych w makrodefinicjach) nazw symbolicznych, ale to są numery 3 (przy obracaniu rolki od siebie) i 4 (przy obracaniu do siebie). Dla lepszej czytelności kodu aplikacji w roli identyfikatorów tych przycisków można pisać wyrażenia stałe GLUT_LEFT_BUTTON+3 i GLUT_LEFT_BUTTON+4.

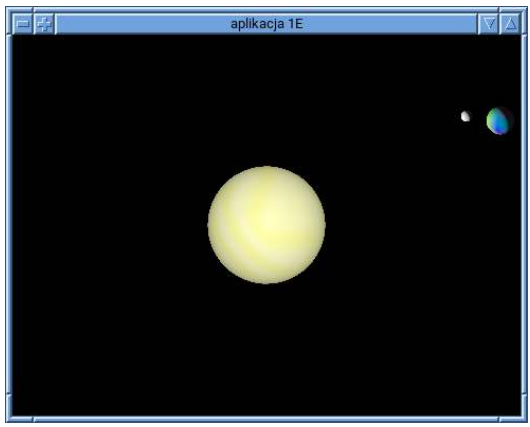
Listing 13.20. Nowa procedura MouseFunc

C

```
1: void MouseFunc ( int button, int state, int x, int y )
2: {
3:   switch ( button ) {
4: case GLUT_LEFT_BUTTON:
5:     switch ( app_state ) {
6:   case STATE_NOTHING:
7:       if ( state == GLUT_DOWN ) {
8:         last_xi = x,  last_eta = y;
9:         app_state = STATE_TURNING;
10:        glutPostWindowRedisplay ( WindowHandle );
```

```
11:       }
12:       break;
13:   case STATE_TURNING:
14:       if ( state != GLUT_DOWN ) {
15:         app_state = STATE_NOTHING;
16:         glutPostWindowRedisplay ( WindowHandle );
17:       }
18:       break;
19:   default:
20:       break;
21:     }
22:     break;
23: case GLUT_LEFT_BUTTON+3:
24:     if ( state == GLUT_DOWN && ChangeZoom ( true ) )
25:       glutPostWindowRedisplay ( WindowHandle );
26:     break;
27: case GLUT_LEFT_BUTTON+4:
28:     if ( state == GLUT_DOWN && ChangeZoom ( false ) )
29:       glutPostWindowRedisplay ( WindowHandle );
30:     break;
31: default:
32:     break;
33:   }
34: } /*MouseFunc*/
```

Obrót rolki o jedną podziałkę powoduje wysłanie kolejno dwóch komunikatów — o naciśnięciu przycisku i o jego zwolnieniu. Procedura `ChangeZoom` jest wywoływana po otrzymaniu pierwszego z nich. Jeśli procedura ta przekaże wartość `true`, to dodatkowo FreeGLUT jest proszony o wykonanie nowego obrazu w oknie (ta prośba jest potrzebna, gdy animacja jest wyłączona, a jeśli jest włączona, to nie szkodzi).



Rysunek 13.2. Okno aplikacji pierwszej E

Listing 13.21. Procedury animacji oraz przyspieszania i spowalniania ruchu

C

```
#define SPEED_FCT   1.2  /* czynnik przyspieszania */

int speedexp = 0;

char ChangeSpeed ( char incspeed )
{
  int    e;
  double s;

  e = speedexp;
  if ( incspeed ) {
    if ( (s = NOM_SPEED*pow ( SPEED_FCT, (double)(++e) )) > MAX_SPEED )
      return false;
  }
  else {
    if ( (s = NOM_SPEED*pow ( SPEED_FCT, (double)(--e) )) < MIN_SPEED )
      return false;
  }
  speedexp = e;  speed = s;
  return true;
} /*ChangeSpeed*/

char ProcessCharCommand ( char charcode )
{
  switch ( toupper ( charcode ) ) {
case 'M':
    font = font == fonts[0] ? fonts[1] : fonts[0];
    NotifyViewerPos ();
    return true;
case '+':
    ChangeSpeed ( true );
    return false;
case '-':
    ChangeSpeed ( false );
    return false;
default:
    return false;
  }
} /*ProcessCharCommand*/

char MoveOn ( void )
{
  ArticulateMyLinkage ( mylinkage );
  return true;
} /*MoveOn*/
```

Listing 13.21 przedstawia procedury, które wykonują polecenia wydawane za pomocą klawiatury. Oprócz włączania i wyłączania animacji (przez naciśnięcie klawisza spacji, czym zajmuje się procedura `KeyFunc`) są trzy takie polecenia. Napisanie litery `M` albo `m` powoduje zmianę wielkości fontu używanego do napisania położenia obserwatora. Znak `+` powoduje zwiększenie, a znak `-` zmniejszenie wartości zmiennej `speed` (listing 13.12), która określa prędkość ruchu. Prędkość ta jest obliczana na podstawie wartości zmiennej `speedexp`, która jest licznikiem wydanych poleceń przyspieszania i spowalniania ruchu, podobnie jak parametr kamery podczas najazdów i odjazdów. Procedura `MoveOn` dokonuje artykulacji łańcucha kinematycznego i zawiadamia, że obraz w oknie trzeba wykonać na nowo.

13.6. *Uzupełnienia — zmniejszanie błędu reprezentacji głębokości

Odległości od obserwatora różnych obiektów w Kosmosie zazwyczaj różnią się o wiele rzędów wielkości; prawie to samo dotyczy elementów krajobrazu i obiektów oglądanych z bliska na jego tle. Tymczasem dokładność podstawowego w OpenGL-u algorytmu widoczności z buforem głębokości maleje ze wzrostem ilorazu parametrów f = `far` i n = `near` określających ostrosłup widzenia.

Zobaczmy wpływ, jaki ograniczona dokładność liczb w buforze głębokości ma na rzetelność algorytmu widoczności. Złożenie przejścia od układu współrzędnych obserwatora do układu kostki standardowej, dokonanego przy użyciu macierzy P skonstruowanej zgodnie z opisem w podrozdziale 6.2, z dalszym przejściem z przedziału $[-1,1]$ do $[0,1]$ opisuje wzór

$$\zeta(z) = \frac{1}{2}\left(1 + \frac{(f+n)z + 2fn}{(f-n)z}\right) = \frac{f}{f-n}\frac{z+n}{z},$$

który wiąże współrzędną z punktu w układzie obserwatora z liczbą ζ reprezentującą głębokość tego punktu porównywaną z głębokościami innych punktów i przechowywaną w buforze głębokości. Odwrotność powyższej funkcji i jej pochodna wyrażają się wzorami

$$z(\zeta) = \frac{fn}{(f-n)\zeta - f}, \quad z'(\zeta) = \frac{-fn(f-n)}{\big((f-n)\zeta - f\big)^2}.$$

Największą wartość bezwzględną pochodna przyjmuje dla $\zeta = 1$, czyli dla punktów tylnej ściany ostrosłupa widzenia:

$$z'(1) = \frac{-f(f-n)}{n}.$$

Najmniej znaczący bit liczby zmiennopozycyjnej $\zeta \in [0.5, 1)$, $\operatorname{ulp}\zeta$, ma wartość 2^{-24}. Jego zmiana odpowiada zmianie współrzędnej z o wielkość $\varepsilon \approx \pm z'(\zeta)\operatorname{ulp}\zeta$. Zobaczmy, jak to wygląda w praktyce: niech $f = 10$ i $\zeta \approx 1$. Jeśli $n = 1$, to $\varepsilon \approx 5.36 \cdot 10^{-6}$, a więc zaburzenie jest całkiem małe. Jeśli $n = 0.1$, to $\varepsilon \approx 5.9 \cdot 10^{-5}$, też niewiele, ale dla $n = 0.001$ mamy już $\varepsilon \approx 0.00596$, czyli zaburzenie ponad pięciokrotnie większe niż odległość przedniej

ściany ostrosłupa widzenia od środka rzutowania. Pamiętajmy też, że podczas przekształcania współrzędnych (zaczynając od układu modelu) powstają błędy zaokrągleń. Mogą one sprawić, że liczby w buforze głębokości będą zaburzone bardziej niż tylko o wartość najmniej znaczącego bitu, co może doprowadzić do wielu błędnych wyników testów widoczności.

Jeśli maksymalna odległość obiektów przewyższa odległość minimalną o więcej niż dwa rzędy wielkości, to można sobie z tym poradzić, wykonując obraz w dwóch lub większej liczbie etapów. Przyjmijmy w każdym etapie $f/n = 64 = 2^6$. Wtedy k etapów umożliwi dosyć dokładne rozstrzyganie widoczności między obiektami znajdującymi się w odległościach pozostających w proporcji nawet $1 : 64^k = 1 : 2^{6k}$; już dla $k = 2$ to jest $1 : 4096$ i dla wielu scen w plenerze to wystarczy. Pętla realizująca kolejne etapy rysowania obrazu może mieć taką oto postać:

```
glViewport ( .... );
glClearColor ( .... );
glClear ( GL_COLOR_BUFFER_BIT );
glEnable ( GL_DEPTH_TEST );
for ( i = k-1; i >= 0; i-- ) {
  glClear ( GL_DEPTH_BUFFER_BIT );
  s = (float)(1 << 6*i);
  M4x4Frustumf ( pm, NULL, s*left, s*right, s*bottom, s*top,
                 s*near, s*64.0*near );
  ....  /* prześlij macierz rzutowania pm do pamięci GPU */
  ....  /* rysuj scenę */
}
```

W powyższym kodzie przyjąłem założenie, że wartości zmiennych `left`, `right`, `bottom`, `top` i `near` (oraz parametru `far=64*near`) opisują ostrosłup ścięty będący częścią bryły widzenia znajdującą się *najbliżej* obserwatora. W wywołaniu procedury konstruującej macierz rzutowania zmienne te są mnożone przez czynnik $s = 64^i$, ostrosłup jest więc poddawany jednokładności o tej skali. Przyjęty w danym etapie parametr n ostrosłupa widzenia staje się w następnym etapie parametrem f, dzięki czemu wybrany zakres odległości obiektów od obserwatora kolejne ostrosłupy pokrywają bez przerw. Istotna jest kolejność etapów: najpierw widoczność jest rozstrzygana między obiektami położonymi najdalej, a obiekty blisko obserwatora zostają pominięte w wyniku obcinania. W kolejnym etapie obcinanie odrzuci (już narysowane) obiekty położone daleko, a te znajdujące się bliżej zasłonią je. Piksele obrazu otrzymują kolor tła tylko na początku, a przed każdym następnym etapem rysowania trzeba skasować tylko bufor głębokości.

Dygresja: Rzut oka na wzór, na podstawie którego procedura `M4x4Frustumf` konstruuje macierz rzutowania (zobacz s. 134), pozwala zauważyć, że macierze otrzymane w kolejnych etapach opisanego tu algorytmu mają identyczne wszystkie współczynniki oprócz jednego. Byłaby to okazja do znakomitej optymalizacji kodu, ale zysk z niej (tj. skrócenie czasu obliczeń i przesyłania danych do pamięci GPU) wydaje się znikomy. Mimo to, moim zdaniem, warto zwracać uwagę na takie rzeczy.

Procedura rysowania sceny może po prostu wyświetlić po kolei wszystkie obiekty. Sceny bardziej skomplikowane warto podzielić na grupy obiektów znajdujących się w różnych odległościach i podczas rysowania, w celu zaoszczędzenia czasu, pomijać całe grupy obiektów znajdujących się poza bieżącym zakresem odległości. Jeśli dla grupy obiektów wyświetlanej w danym etapie zakres odległości od obserwatora jest wąski (w Kosmosie to jest częsta sytuacja, może dotyczyć planety z księżycami lub z pierścieniami), to można macierz rzutowania dostosować do tego zakresu, aby zwiększyć dokładność algorytmu widoczności.

13.7. Ćwiczenia

1. Dodaj do aplikacji planety Wenus, Mars i dalsze.
2. Zastanów się, jak zmodyfikować łańcuch, aby planety i inne obiekty mogły poruszać się wokół Słońca po orbitach eliptycznych zgodnie z prawami Keplera.
3. Rozszerz łańcuch kinematyczny tak, aby umożliwić podążanie obserwatora za Ziemią na orbicie wokółsłonecznej lub umieszczenie go na orbicie wokółziemskiej.
4. Dobierz początkowe położenia Ziemi i Księżyca tak, aby były zgodne z pewną datą i godziną w świecie fizycznym, utwórz zegar (kalendarz) symulacji ruchu (który odmierza rok podczas jednego obiegu Ziemi wokół Słońca) i wyświetlaj w oknie tekst opisujący datę odpowiadającą bieżącemu położeniu Ziemi na orbicie.
5. Po rozbudowaniu Układu Słonecznego o dodatkowe planety i wprowadzeniu możliwości swobodnego przemieszczania obserwatora w nim, zrealizuj procedurę rysowania sceny zawierającej obiekty położone w znacznie różniących się odległościach, zgodnie z opisem w podrozdziale 13.6.
6. *Po przestudiowaniu rozdz. 19 nałóż na Ziemię teksturę — siatkę geograficzną składającą się z południków i równoleżników (podobną do tych na rys. 13.1) albo obraz oceanów, kontynentów i ewentualnie chmur nad nimi. Nałóż też na Słońce i Księżyc tekstury przedstawiające granule i plamy słoneczne oraz morza i kratery.
7. *Po przestudiowaniu rozdz. 22 dodaj do aplikacji algorytm wyznaczania cieni, aby otrzymać „zaćmienia Słońca" i „zaćmienia Księżyca".
8. *Po przestudiowaniu rozdz. 24 dodaj do układu kometę, której orbita jest elipsą i która w pobliżu Słońca wypuszcza warkocz zrealizowany za pomocą układu cząsteczek.
9. *Po przestudiowaniu p. 26.4.2 wprowadź „Kosmos", tj. obraz gwiazd wokół „Układu Słonecznego". Nie ma potrzeby dokładnego przedstawiania wszystkich gwiazd w Galaktyce.
10. *Powierzchnia Księżyca (tego prawdziwego) odbija światło w sposób dość znacznie różniący się od lambertowskiego. Po przestudiowaniu podrozdziału 28.4 napisz szader realizujący model odbicia światła Orena i Nayara i wypróbuj go na Księżycu.

14

Aplikacja pierwsza F

… there's nothing fishy about carrying a Galleon, is there?
J.K. Rowling: *Harry Potter and the Order of the Phoenix*

Czytelnikom zapewne zdążył się już opatrzyć dwudziestościan i inne proste bryły otrzymane po rozwiązaniu ćwiczeń na końcu rozdziału 7. Dlatego teraz proponuję zmienić aplikację tak, aby mogła wyświetlać obiekty opisane przez dane przeczytane z pliku. Wiele plików z takimi danymi można znaleźć w Internecie. Naszą aplikację dostosujemy do czytania plików umieszczonych na stronie [58]. Są tu dwa zadania dla programisty: implementacja czytania danych z pliku i wyświetlania ich.

14.1. Czytanie pliku SMF

Opis formatu SMF można znaleźć w dokumencie [57]. Dane w tym formacie są zapisywane w postaci tekstowej. Procedura czytająca taki plik musi wyodrębnić występujące w nim symbole leksykalne i dokonać rozbioru gramatycznego, którego celem jest zidentyfikowanie poszczególnych danych (np. wierzchołków, ścian i przekształceń) zapisanych w pliku. Trochę to przypomina działanie kompilatora, który dokonuje analizy składniowej (czyli rozbioru gramatycznego) tekstu źródłowego programu napisanego w jakimś języku. Jest tu jednak pewna różnica: kompilator powinien rygorystycznie sprawdzić poprawność programu i odmówić wyprodukowania wyniku, jeśli w programie źródłowym są błędy[1]. Jeśli natomiast plik z danymi opisującymi model zawiera błędy, to warto przeczytać z tego pliku „co się da" i wyświetlić nawet niekompletny obiekt, co ułatwi znalezienie i ewentualne poprawienie błędu. Ponadto pewne dane, potrzebne w aplikacji, mogą być w pliku nieobecne (i aplikacja musi je uzupełnić), plik może też zawierać dane nieużywane przez aplikację i procedura czytająca musi je pominąć.

Opisana tu procedura czytania pliku SMF składa się z dwóch elementów: skanera i parsera. **Skaner**, czyli analizator leksykalny, ma za zadanie wyodrębnić z pliku symbole takie jak identyfikatory, liczby i operatory i pominąć komentarze. **Parser**, czyli analizator składni,

[1] i wypisywać ostrzeżenia, które autor programu zawsze powinien traktować poważnie

Listing 14.1. Plik nagłówkowy skanera

C

```
#define SYMB_ERROR      -2
#define SYMB_EOF        -1
#define SYMB_NONE        0
#define SYMB_PLUS        1
#define SYMB_MINUS       2
....
#define SYMB_INTEGER    14
#define SYMB_FLOAT      15
....
#define SYMB_PERCENT    22
....
#define SYMB_IDENT      33
#define SYMB_FIRSTOTHER 34

#define SCANNER_BUFLENGTH 1024

typedef struct scanner {
    int    inbuflength, namebuflength;
    char   *inbuffer;
    char   *nextname;
    char   alloc_inb, alloc_nb;
    int    bcomment, ecomment;
    int    inbufpos, inbufcount, namebufcount;
    int    linenum, colnum;
    int    nextchar;
    int    nextsymbol;
    int    nextinteger;
    double nextfloat;
    int    (*InputData)( void *usrdata, int buflength, char *buffer );
    void   *userdata;
  } scanner;

char InitScanner ( scanner *sc,
                   int inbuflength, char *inbuffer,
                   int maxnamelength, char *namebuffer,
                   int bcomment, int ecomment,
                   int (*InputData)(void *userdata,
                                    int buflength,char *buffer),
                   void *userdata );
void GetNextChar ( scanner *sc );
int GetNextSymbol ( scanner *sc );
void ShutDownScanner ( scanner *sc );

int BinSearchNameTable ( int nnames, int firstname, const char **names,
                         const char *name );
```

wywołuje w pętli skaner, otrzymując za każdym razem jeden, kolejny symbol, i zapamiętuje w tablicach przeczytane przez skaner dane.

Listing 14.1 przedstawia plik nagłówkowy prostego skanera, zawierający opis struktury danych skanera, nagłówki procedur i makrodefinicje identyfikatorów symboli rozpoznawanych przez skaner. Po namyśle nie zdecydowałem się zamieszczać listingu z treścią procedur skanera, która jest dosyć mało skomplikowana, dosyć długa i zbyt odległa tematycznie od OpenGL-a. Zamieszczam tylko opis sposobu ich używania.

Makrodefinicje w liniach 1–12 opisują symbole przetwarzane przez opisany dalej parser plików SMF; pojawienie się dowolnego innego symbolu (np. nawiasu, przecinka), którego nazwa została na listingu pominięta, oznacza błąd w czytanym pliku.

Pola struktury `scanner` przeznaczone do użytku wewnętrznego to długości bufora wejściowego i bufora bieżącej nazwy (`inbuflength`, `namebuflength`), wskaźniki tych buforów (`inbuffer`, `nextname`), informacja, czy to skaner dokonał ich rezerwacji (`alloc_inb`, `alloc_nb`), kody ASCII znaków rozpoczynających i kończących komentarze (`bcomment`, `ecomment`), indeksy do buforów (`inbufpos`, `inbufcount`, `namebufcount`) i numery bieżącej linii (`linenum`) oraz znaku w linii (`colnum`) czytanego pliku. Ponadto skaner ma wskaźnik procedury, która wprowadza dane do bufora wejściowego (`InputData` — może to być procedura czytająca plik lub dostarczająca dane z dowolnego innego źródła) i wskaźnik dowolnej struktury danych określonej przez aplikację (`usrdata` — procedura wprowadzająca dane ma w razie potrzeby dostęp do tej struktury).

Parser (opisany dalej lub dowolny inny, korzystający z opisanego tu skanera) powinien wywołać procedurę `InitScanner`, której parametry to wskaźnik zmiennej typu `scanner`, długość i wskaźnik bufora wejściowego (jeśli wskaźnik ten jest pusty, to procedura `InitScanner` dokona rezerwacji bufora, a po dojściu do końca czytanego pliku odwoła ją), długość i wskaźnik bufora bieżącej nazwy, kody ASCII znaków początku i końca komentarza (ten skaner nie obsługuje wieloznakowych ograniczników komentarzy), wskaźnik procedury wprowadzającej dane do bufora wejściowego i wskaźnik dowolnych danych aplikacji. Procedura `InitScanner` nadaje wartości początkowe wszystkim polom struktury skanera i rozpoczyna proces czytania.

Po jego rozpoczęciu należy wywoływać procedurę `GetNextSymbol`, która za każdym razem przypisze polu `nextsymbol` skanera jedną z wartości nazwanych makrodefinicjami opisanymi w pliku nagłówkowym. Jeśli przypisana wartość ma nazwę `SYMB_INTEGER`, to z pliku została przeczytana liczba całkowita (bez znaku), która została przypisana polom `nextinteger` i `nextfloat`. Jeśli przeczytany symbol to `SYMB_FLOAT`, to została przeczytana liczba rzeczywista (z częścią ułamkową lub wykładnikiem), przypisana polu `nextfloat`. Jeśli pole `nextsymbol` ma wartość `SYMB_IDENT`, to została przeczytana nazwa (tj. ciąg liter i cyfr zaczynający się od litery), zapamiętana w buforze `nextname`.

Jeśli pole `nextsymbol` otrzymało wartość `SYMB_ERROR` lub `SYMB_EOF`, to odpowiednio został wykryty błąd uniemożliwiający dalsze czytanie lub proces czytania doszedł do końca pliku. Dalsze wywołania procedury `GetNextSymbol` są wtedy bezprzedmiotowe.

Procedura `BinSearchNameTable` służy do wyszukiwania binarnego napisu wskazywanego przez ostatni parametr w (posortowanej leksykograficznie) tablicy napisów o długości

nnames. Wartość powrotna jest sumą indeksu odnalezionego w tablicy napisu i parametru firstname, albo SYMB_ERROR, jeśli taki napis w tablicy się nie znalazł.

Listing 14.2 przedstawia tablicę identyfikatorów, które mogą występować w plikach SMF; część z nich nie jest opisana w dokumencie [57], który jest trochę niekompletny. Tablica jest posortowana, dzięki czemu napotkane w pliku identyfikatory można w niej wyszukiwać metodą binarną. Makrodefinicje w liniach 2–30 wprowadzają nazwy symboliczne tych liczb przyporządkowanych tym identyfikatorom; muszą to być kolejne liczby całkowite.

Listing 14.2. Słowa kluczowe parsera plików SMF

C

```
1: #define SMF_FIRST_KEY              100
2: #define SMF_KEY_BEGIN              100
3: #define SMF_KEY_BIND               101
4: #define SMF_KEY_C                  102
5: #define SMF_KEY_E                  103
6: #define SMF_KEY_END                104
7: #define SMF_KEY_F                  105
8: #define SMF_KEY_FACE               106
9: #define SMF_KEY_FN                 107
10: #define SMF_KEY_FT                108
11: #define SMF_KEY_G                 109
12: #define SMF_KEY_N                 110
13: #define SMF_KEY_Q                 111
14: #define SMF_KEY_R                 112
15: #define SMF_KEY_ROT               113
16: #define SMF_KEY_SCALE             114
17: #define SMF_KEY_SET               115
18: #define SMF_KEY_T                 116
19: #define SMF_KEY_T_SCALE           117
20: #define SMF_KEY_T_TRANS           118
21: #define SMF_KEY_TEX               119
22: #define SMF_KEY_TRANS             120
23: #define SMF_KEY_V                 121
24: #define SMF_KEY_VERTEX            122
25: #define SMF_KEY_VERTEX_CORRECTION  123
26: #define SMF_KEY_VN                124
27: #define SMF_KEY_VT                125
28: #define SMF_KEY_X                 126
29: #define SMF_KEY_Y                 127
30: #define SMF_KEY_Z                 128
31: #define SMF_NUM_KEYWORDS (SMF_KEY_Z-SMF_FIRST_KEY+1)
32:
33: const char *smf_keyword[SMF_NUM_KEYWORDS] =
34:   { "begin", "bind", "c", "e", "end", "f", "face", "fn", "ft", "g", "n",
35:     "q", "r", "rot", "scale", "set", "t", "t_scale", "t_trans", "tex",
36:     "trans", "v", "vertex", "vertex_correction", "vn", "vt", "x", "y", "z" };
```

Listing 14.3 przedstawia procedurę czytania pliku SMF za pomocą parsera przedstawionego dalej. Pierwszy parametr jest nazwą pliku do przeczytania, a kolejne cztery wskazują zmienne, którym procedura nada wartości: liczbę przeczytanych wierzchołków, wskaźnik tablicy ze współrzędnymi tych wierzchołków, liczbę przeczytanych trójkątów i wskaźnik tab-

Listing 14.3. Procedura `ReadSMFFile`

C

```
typedef struct {
    FILE    *f;
    int     nvert, ntr, ntrv, vertc;
    GLfloat *vc;
    GLuint  *trv;
  } smf_data;

char ReadSMFFile ( char *fn, int *nv, GLfloat **vc, int *ntr, GLuint **trv )
{
  scanner  sc;
  smf_data data;
  GLfloat  tr[16];
  int      i;

  *nv = *ntr = data.nvert = data.vertc = data.ntr = data.ntrv = 0;
  *vc = NULL;  *trv = NULL;
  if ( !smf_OpenInputFile ( &sc, &data, fn ) )
    return false;
  data.vc = malloc ( MAXVERT*3*sizeof(GLfloat) );
  data.trv = malloc ( MAXFAC*3*sizeof(GLuint) );
  if ( data.vc && data.trv ) {
    M4x4Identf ( tr );
    smf_CmdSequence ( &sc, tr );
    ShutDownScanner ( &sc );
    fclose ( data.f );
    printf ( "nvert = %d, ntr = %d\n", data.nvert, data.ntr );
    if ( data.ntr > 0 ) {
      for ( i = 0; i < 3*data.ntr; i++ )
        if ( data.trv[i] >= data.nvert )
          goto failure;
      *vc = data.vc;    *nv = data.nvert;
      *trv = data.trv;  *ntr = data.ntr;
      return true;
    }
  }
failure:
  if ( data.vc ) free ( data.vc );
  if ( data.trv ) free ( data.trv );
  return false;
} /*ReadSMFFile*/
```

licy z indeksami wierzchołków tych trójkątów. Procedura zarezerwuje tablice za pomocą procedury `malloc`; gdy przestaną być potrzebne, *aplikacja* powinna je zwolnić przy użyciu procedury `free`. Jeśli nie udało się przeczytać żadnych danych (bo plik o podanej nazwie nie istnieje lub jego błędna składnia to uniemożliwiła), to zmiennym wskazywanym przez parametry zostaną przypisane wartości `0` i `NULL`. Ale jeśli choć część danych udało się przeczytać przed wykryciem błędu w pliku, to aplikacja otrzyma do nich dostęp.

Listing 14.4 przedstawia procedury pomocnicze. Procedura `GetNextSymbol` skanera jest wywoływana za pośrednictwem procedury `smf_GetNextSymbol`, która po stwierdzeniu, że kolejny symbol jest identyfikatorem, wyszukuje ten identyfikator w tablicy z listingu 14.2 i przypisuje zmiennej `sc->nextsymbol` odpowiednią wartość liczbową (która może być użyta w instrukcji przełącznika)[2].

Format SMF zakłada, że poszczególne dane zajmują osobne linie tekstu, co bardzo ułatwia czytanie, a w szczególności pomijanie tych danych, których aplikacja „nie rozumie" i „nie potrzebuje". Linia może być pusta, może być komentarzem (wtedy zaczyna się od znaku # albo %) lub ma na początku identyfikator, po którym są dane składające się z liczb lub dalszych identyfikatorów. Jeśli identyfikator określa, że dane za nim nie są aplikacji potrzebne, to wystarczy pominąć w czytanym pliku wszystkie znaki do końca linii. Temu służy procedura `smf_SkipToLineEnd`.

Procedura `smf_OpenInputFile` (wywołana w linii 17 na listingu 14.3) otwiera plik o podanej nazwie i przygotowuje skaner (wskazywany przez parametr — jest on zmienną lokalną procedury `ReadSMFFile`) do działania. Wskaźnik struktury opisującej otwarty plik jest przypisywany polu `f` struktury `*data`, a adres tej struktury jest zapamiętywany w polu `userdata` skanera. Parametry `inbuffer` i `namebuffer` procedury `InitScanner` mają wartość `NULL`, co spowoduje, że procedura ta zarezerwuje bufory, które zostaną zwolnione przez procedurę `ShutDownScanner`. Parametr `inbuflength` ma wartość `0`; bufor wejścia dla skanera otrzyma wielkość domyślną 1 KB.

Parametry `bcomment` i `ecomment` mają wartości `'#'` i `'\n'`; jeśli zatem skaner trafi na znak #, to pominie czytany tekst do końca linii, uznając go za komentarz. Komentarze o drugiej dopuszczalnej postaci obsługuje (tj. pomija) parser.

Parametr `InputData` wskazuje procedurę `_smf_ReadInputFile`, którą skaner wywoła na początku działania i za każdym razem, gdy pobierając znaki, dojdzie do końca bufora. Procedura ta czyta surowe dane z otwartego pliku, do którego ma dostęp za pomocą pola `f` struktury typu `smf_data` przekazanej przez skaner. Podana wartość jest liczbą przeczytanych z pliku bajtów; 0, jeśli proces czytania dotarł do końca pliku.

Procedury na listingu 14.5 służą do rozwiązania następującego problemu: liczby rozpoznawane przez skaner są nieujemne, a znaki + i –, które mogą je poprzedzać, są rozpoznawane jako osobne symbole `SYMB_PLUS` i `SYMB_MINUS`. Z kolei parser w pewnych miejscach spodziewa się danych liczbowych ze znakiem. Parser wywołuje wtedy procedurę `smf_GetSignedInt` lub `smf_GetSignedFloat`; zadaniem tych procedur jest przeczytanie znaku (jeśli jest obecny) i liczby, a następnie podanie liczby z właściwym znakiem. W obu przypad-

[2] Aby to działało poprawnie, liczby w makrodefinicjach na listingach 14.1 i 14.2 muszą być różne. Dlatego liczby na listingu 14.2 zaczynają się od `100`.

kach rozpoznanie, czy w tekście jest znak, a jeśli tak, to jaki, wykonuje procedura `smf_GetSign`, której wartość to `+1` albo `-1`.

Listing 14.4. Procedury pomocnicze parsera

C

```
#define MAX_NAME_LENGTH 32

void smf_GetNextSymbol ( scanner *sc )
{
  int ns;

  ns = GetNextSymbol ( sc );
  if ( ns == SYMB_IDENT )
    sc->nextsymbol = BinSearchNameTable ( SMF_NUM_KEYWORDS,
                       SMF_FIRST_KEY, smf_keyword, sc->nextname );
} /*smf_GetNextSymbol*/

void smf_SkipToLineEnd ( scanner *sc )
{
  while ( sc->nextchar != '\n' && sc->nextchar != EOF )
    GetNextChar ( sc );
  if ( sc->nextchar == '\n' ) {
    GetNextChar ( sc );
    smf_GetNextSymbol ( sc );
  }
} /*smf_SkipToLineEnd*/

int _smf_ReadInputFile ( void *userdata, int buflength, char *buffer )
{
  return fread ( buffer, 1, buflength, ((smf_data*)userdata)->f );
} /*_smf_ReadInputFile*/

char smf_OpenInputFile ( scanner *sc, smf_data *data, const char *filename )
{
  if ( !(data->f = fopen ( filename, "r" )) )
    return false;
  if ( InitScanner ( sc, 0, NULL, MAX_NAME_LENGTH, NULL, '#', '\n',
                     _smf_ReadInputFile, (void*)data ) ) {
    smf_GetNextSymbol ( sc );
    return true;
  }
  else {
    sc->userdata = (void*)data;
    fclose ( data->f );
    return false;
  }
} /*smf_OpenInputFile*/
```

Listing 14.5. Procedury czytania liczb ze znakiem

C

```
1: int smf_GetSign ( scanner *sc )
2: {
3:   int sign;
4:
5:   sign = 1;
6:   if ( sc->nextsymbol == SYMB_MINUS ) {
7:     smf_GetNextSymbol ( sc );
8:     sign = -1;
9:   }
10:  else if ( sc->nextsymbol == SYMB_PLUS )
11:    smf_GetNextSymbol ( sc );
12:  return sign;
13: } /*smf_GetSign*/
14:
15: int smf_GetSignedInt ( scanner *sc )
16: {
17:  int sign, num;
18:
19:  sign = smf_GetSign ( sc );
20:  if ( sc->nextsymbol == SYMB_INTEGER ) {
21:    num = sc->nextinteger;
22:    smf_GetNextSymbol ( sc );
23:    return sign*num;
24:  }
25:  else {
26:    sc->nextsymbol = SYMB_ERROR;
27:    return 0;
28:  }
29: } /*smf_GetSignedInt*/
30:
31: double smf_GetSignedFloat ( scanner *sc )
32: {
33:  double sign, num;
34:
35:  sign = (double)smf_GetSign ( sc );
36:  if ( sc->nextsymbol == SYMB_INTEGER || sc->nextsymbol == SYMB_FLOAT ) {
37:    num = sc->nextfloat;
38:    smf_GetNextSymbol ( sc );
39:    return sign*num;
40:  }
41:  else {
42:    sc->nextsymbol = SYMB_ERROR;
43:    return 0.0;
44:  }
45: } /*smf_GetSignedFloat*/
```

W linii 23 (na listingu 14.3) procedura ReadSMFFile wywołuje przedstawioną na listingu 14.6 procedurę smf_CmdSequence, której zadaniem jest dokonanie analizy składni tekstu w czytanym pliku i zapamiętanie w tablicach przeczytanych danych. Figury geometryczne w pliku SMF to tylko punkty (wierzchołki), trójkąty i czworokąty, ale mogą one być zebrane w **bloki** rozpoczynające się identyfikatorem begin (SMF_KEY_BEGIN) i zakończone identyfikatorem end (SMF_KEY_END). Choć dokument [57] o tym nie wspomina, przyjąłem, że bloki mogą być zagnieżdżane; stąd analiza składni wymaga użycia procedury rekurencyjnej.

Parametrami procedury są skaner, który dostarcza kolejne symbole z pliku i macierz przekształcenia afinicznego, któremu mają być poddane przeczytane wierzchołki. Początkowo jest to przekszałcenie tożsamościowe (reprezentowane przez macierz jednostkową), ale w pliku mogą być opisy przekształceń; kolejne przekształcenia należy składać i poddawać im przeczytane wierzchołki. Przekształcenia określone w bloku powinny być złożone z przekształceniem określonym przy wejściu do bloku, ale zakończenie bloku ma spowodować przywrócenie przekształcenia danego na jego początku. W linii 10 macierz przekształcenia przekazana przez procedurę wywołującą (jednostkowa, jeśli wywołanie nastąpiło w procedurze ReadSMFFile, zobacz listing 14.3, linia 22) jest kopiowana do lokalnej tablicy.

Prawie cała treść procedury smf_CmdSequence to nieskończona pętla, w której jest instrukcja przełącznika; wybór instrukcji w jej wnętrzu jest dokonywany na podstawie ostatniego rozpoznanego symbolu. Większość dopuszczalnych w tym miejscu symboli to identyfikatory z listingu 14.2. Jeśli jest to identyfikator SMF_KEY_BEGIN, to w linii 14 skaner dostarcza następny symbol, po czym procedura smf_CmdSequence jest wywoływana rekurencyjnie. Gdy powróci, ostatnim przeczytanym (i nieprzetworzonym) symbolem powinien być identyfikator SMF_KEY_END; jeśli nie, to plik wejściowy ma niepoprawną składnię. Identyfikator końca bloku jest w linii 17 pomijany, po czym następuje powrót.

W linii 24 następuje powrót po natrafieniu na identyfikator SMF_KEY_END lub na symbol końca pliku. Zwróćmy uwagę, że symbole te nie są pomijane; identyfikator końca bloku jest pomijany (przez wywołanie procedury smf_GetNextSymbol) po powrocie z wywołania rekurencyjnego w linii 17[3].

Etykiety wyboru w liniach 29–34 poprzedzają instrukcję, która pomija wszystkie znaki do końca bieżącej linii. Symbol SYMB_PERCENT jest traktowany jak początek komentarza. Pozostałe symbole są identyfikatorami danych pomijanych przez opisaną tu procedurę czytania plików SMF. Znaczenie tych danych i składnia linii tekstu, w której są one zapisane, są opisane w dokumencie [57]; jednym z proponowanych Czytelnikom na końcu rozdziału ćwiczeń jest przeczytanie tego dokumentu i rozszerzenie procedury czytania o te dane.

Identyfikator v (czyli SMF_KEY_V) rozpoczyna linię, w której są podane współrzędne kartezjańskie wierzchołka. Po tym identyfikatorze w linii powinny być trzy liczby rzeczywiste (ze znakami), które procedura czyta w pętli w liniach 41–44. Liczby są wpisywane do lokalnej tablicy va, po czym w linii 45 wierzchołek jest poddawany przekształceniu, którego macierz jest w tablicy tr. Wynik tego przekształcenia jest zapamiętywany w tablicy, która zostanie przekazana aplikacji.

[3] Jeśli identyfikator SMF_KEY_END nie zamyka bloku, to nastąpi powrót do procedury ReadSMFFile i koniec czytania.

Listing 14.6. Procedura czytania danych z pliku SMF

C

```
char smf_CmdSequence ( scanner *sc, GLfloat mm[16] )
{
  int      i, j, fvn;
  GLfloat  tm[16], tr[16], tt[16];
  GLfloat  va[3], phi;
  smf_data *data;

  data = (smf_data*)sc->userdata;
  fvn = data->nvert;
  memcpy ( tr, mm, 16*sizeof(GLfloat) );
  for (;;) {
    switch ( sc->nextsymbol ) {
case SMF_KEY_BEGIN:
      smf_GetNextSymbol ( sc );
      if ( smf_CmdSequence ( sc, tr ) ) {
        if ( sc->nextsymbol == SMF_KEY_END ) {
          smf_GetNextSymbol ( sc );
          return true;
        }
      }
      return false;

case SMF_KEY_END:  case SYMB_EOF:
      return true;

case SYMB_ERROR:
      return false;

case SYMB_PERCENT:  /* komentarz */
case SMF_KEY_BIND:  case SMF_KEY_C:  case SMF_KEY_E:
case SMF_KEY_FACE:  case SMF_KEY_FN:  case SMF_KEY_FT:  case SMF_KEY_G:
case SMF_KEY_N:  case SMF_KEY_R:  case SMF_KEY_TEX:  case SMF_KEY_T_SCALE:
case SMF_KEY_T_TRANS:  case SMF_KEY_VN:  case SMF_KEY_VT:
case SMF_KEY_VERTEX:
      smf_SkipToLineEnd ( sc );
      continue;

case SMF_KEY_V:
      if ( data->nvert >= MAXVERT ) return false; /* za dużo wierzchołków */
      smf_GetNextSymbol ( sc );
      for ( i = 0; i < 3; i++ ) {
        va[i] = smf_GetSignedFloat ( sc );
        if ( sc->nextsymbol == SYMB_ERROR )  return false;
      }
      M4x4MultMP3f ( &data->vc[3*data->nvert], tr, va );
```

```
46:       data->nvert ++;
47:       continue;
48:
49: case SMF_KEY_F:  case SMF_KEY_T:  /* ściana trójkątna */
50:       if ( data->ntr >= MAXFAC ) return false;  /* za dużo ścian */
51:       smf_GetNextSymbol ( sc );
52:       for ( i = 0; i < 3; i++ ) {
53:         if ( sc->nextsymbol == SYMB_INTEGER ) {
54:           data->trv[data->ntrv++] = sc->nextinteger+fvn+data->vertc-1;
55:           smf_GetNextSymbol ( sc );
56:         }
57:         else return false;
58:       }
59:       data->ntr ++;
60:       continue;
61:
62: case SMF_KEY_Q:  /* ściana czworokątna - zamieniam na 2 trójkąty */
63:       if ( data->ntr-1 >= MAXFAC ) return false;  /* za dużo ścian */
64:       smf_GetNextSymbol ( sc );
65:       for ( i = 0, j = data->ntrv;  i < 4;  i++ ) {
66:         if ( sc->nextsymbol == SYMB_INTEGER ) {
67:           data->trv[data->ntrv++] = sc->nextinteger+fvn+data->vertc-1;
68:           smf_GetNextSymbol ( sc );
69:         }
70:         return false;
71:       }
72:       data->trv[data->ntrv++] = data->trv[j++];
73:       data->trv[data->ntrv++] = data->trv[j++];
74:       data->ntr += 2;
75:       continue;
76:
77: case SMF_KEY_ROT:
78:       smf_GetNextSymbol ( sc );
79:       i = sc->nextsymbol;  smf_GetNextSymbol ( sc );
80:       phi = smf_GetSignedFloat ( sc );
81:       if ( sc->nextsymbol != SYMB_ERROR ) phi *= PI/180.0;
82:       else return false;
83:       switch ( i ) {
84:     case SMF_KEY_X:  M4x4RotateXf ( tt, phi );  goto comp_trans;
85:     case SMF_KEY_Y:  M4x4RotateYf ( tt, phi );  goto comp_trans;
86:     case SMF_KEY_Z:  M4x4RotateZf ( tt, phi );  goto comp_trans;
87:     default:
88:         return false;
89:       }
90:       continue;
91:
92: case SMF_KEY_SCALE:
```

```
93:       smf_GetNextSymbol ( sc );
94:       for ( i = 0; i < 3; i++ ) {
95:         va[i] = smf_GetSignedFloat ( sc );
96:         if ( sc->nextsymbol == SYMB_ERROR )  return false;
97:       }
98:       M4x4Scalef ( tt, va[0], va[1], va[2] );
99:       goto comp_trans;
100:
101: case SMF_KEY_TRANS:
102:       smf_GetNextSymbol ( sc );
103:       for ( i = 0; i < 3; i++ ) {
104:         va[i] = smf_GetSignedFloat ( sc );
105:         if ( sc->nextsymbol == SYMB_ERROR )  return false;
106:       }
107:       M4x4Translatef ( tt, va[0], va[1], va[2] );
108: comp_trans:
109:       M4x4Multf ( tm, tr, tt );
110:       memcpy ( tr, tm, 16*sizeof(GLfloat) );
111:       continue;
112:
113: case SMF_KEY_SET:
114:       smf_GetNextSymbol ( sc );
115:       if ( sc->nextsymbol == SMF_KEY_VERTEX_CORRECTION ) {
116:         smf_GetNextSymbol ( sc );
117:         data->vertc = smf_GetSignedInt ( sc );
118:         if ( sc->nextsymbol == SYMB_ERROR )  return false;
119:       }
120:       else smf_SkipToLineEnd ( sc );
121:       continue;
122:
123: default:
124:       printf ( "SYMBOL %d (%c) at line %d, column %d\n",
125:                sc->nextsymbol, sc->nextsymbol,
126:                sc->linenum+1, sc->colnum+1 );
127:       sc->nextsymbol = SYMB_ERROR;
128:       return false;
129:     }
130:   }
131: } /*smf_CmdSequence*/
```

Identyfikator `t` (`SMF_KEY_T`) lub `f` (`SMF_KEY_F`) rozpoczyna linię z opisem jednego trójkąta. Po nim powinny być podane trzy liczby całkowite, które są indeksami wierzchołków. Domyślnie wierzchołki są numerowane od 1, a w programie w C oraz w etapie pobierania wierzchołków OpenGL-a chcemy mieć tablice z indeksami od 0. Aby format był bardziej skomplikowany (i elastyczny), istnieje możliwość numerowania wierzchołków od dowolnej liczby; przesunięcie numeracji względem domyślnego sposobu jest zapamiętane w zmien-

nej `data->vertc`. Numeracja wierzchołków w każdym bloku zaczyna się od nowa. Dlatego do tablicy numerów wierzchołków (która będzie przekazana aplikacji) zostają (w linii 54, a także 67) zapisane wartości wyrażenia, które jest sumą numeru podanego w pliku (`sc->nextinteger`), numeru pierwszego wierzchołka w danym bloku (`fvn`, numer ten został zapamiętany w linii 9), przesunięcia numeracji (`data->vertc`) i liczby `-1`, która zmienia domyślną numerację wierzchołków zdefiniowaną w formacie pliku (od 1) na numerację od zera.

Identyfikator `q` (`SMF_KEY_Q`) rozpoczyna opis czworokąta, który procedura czytająca od razu zamieni na dwa trójkąty. Do tablicy numerów wierzchołków są czytane cztery liczby całkowite, po czym dwie z nich są dopisywane do tablicy.

Instrukcje w liniach 77–111 konstruują przekształcenia wierzchołków, odpowiednio obroty, skalowania i przesunięcia. Po identyfikatorze `rot` (`SMF_KEY_ROT`) powinien wystąpić jeden z identyfikatorów `x`, `y` albo `z`, określający oś obrotu, a po nim liczba będąca miarą kąta obrotu *w stopniach*. Miara ta jest w linii 81 przeliczana na radiany, po czym następuje konstrukcja macierzy obrotu wokół wskazanej osi i (w linii 109) złożenie tego obrotu z dotychczasowym przekształceniem wierzchołków. Iloczyn macierzy, który reprezentuje przekształcenie złożone, jest w linii 110 kopiowany do tablicy `tr`.

Podobnie wyglądają konstrukcje skalowań i przesunięć. Po identyfikatorze `scale` (`SMF_KEY_SCALE`) lub `trans` (`SMF_KEY_TRANS`) w pliku powinny być trzy liczby rzeczywiste, które są współczynnikami skalowani osi x, y, z albo współrzędnymi wektora przesunięcia. Po skonstruowaniu odpowiedniej macierzy skalowania lub przesunięcia następuje składanie tego przekształcenia z dotychczasowym i zapamiętywanie macierzy przekształcenia złożonego. Zwróćmy uwagę, że w obliczanym iloczynie macierz przekształcenia dotychczasowego jest pierwszym, a nowego drugim czynnikiem.

Po identyfikatorze `set` (`SMF_KEY_SET`) powinien wystąpić identyfikator `vertex_correction` (`SMF_KEY_VERTEX_CORRECTION`), a po nim liczba całkowita (ze znakiem), która określa opisane wyżej przesunięcie numeracji wierzchołków. Jeśli jest to liczba `+1`, to wierzchołki w pliku są numerowane od 0.

Ostatnią czynnością wykonywaną przez procedurę czytania pliku SMF jest proste sprawdzenie poprawności danych (listing 14.3, linie 28–30). Procedura sprawdza, czy wszystkie numery wierzchołków są we właściwym zakresie, tj. są mniejsze niż liczba przeczytanych wierzchołków. Nie ma prostego sposobu dokładniejszego sprawdzenia poprawności danych, ale ten test jest konieczny, aby uniknąć sięgania poza tablicę wierzchołków podczas rysowania obiektu.

14.2. Zmiany w aplikacji

Aplikacja pierwsza F powstała przez przerobienie aplikacji pierwszej C, nie D, bo tu nie korzystamy z rozdrabniania. Szadery aplikacji 1C są pozostawione bez żadnych zmian.

Pierwszą czynnością jest pobranie nazwy pliku SMF z polecenia wywołania aplikacji, który ma zostać przeczytany; nazwa ta jest napisem w tablicy `argv` będącej parametrem procedury `main`, przekazanej dalej procedurom `InitMyWorld` i `InputFilename` (listing 14.7).

Listing 14.7. Procedura `InitMyWorld` aplikacji 1F

C

```
int     smf_nv, smf_ntr;
GLfloat smf_mm[16];
GLuint  smf_vao, smf_vbo[2];

char *InputFilename ( int argc, char *argv[] )
{
  int i;

  for ( i = 1; i < argc-1; i++ )
    if ( !strcmp ( argv[i], "-i" ) )
      return argv[i+1];
  return NULL;
} /*InputFilename*/

void InitMyWorld ( int argc, char *argv[], int  width, int height )
{
  char   *fn;
  static GLfloat fg[4] =  0.0, 0.0, 1.0, 1.0 ;
  static GLfloat bk[4] =  0.0, 0.0, 0.0, 0.0 ;

  if ( !(fn = InputFilename ( argc, argv )) ) {
    printf ( "wywolaj tak: app1f -i plik.smf\n" );
    exit ( 0 );
  }
  LoadMyShaders ();
  .... /* tworzenie buforów dla bloków zmiennych jednolitych bez zmian */
  LoadTextShaders ();
  TimerInit ();
  font = fonts[0] = NewFont18x10 ();
  fonts[1] = NewFont12x6 ();
  vptext = NewTextObject ( 60 );
  SetTextForeground ( fg );
  SetTextBackground ( bk );
  ConstructSMFObject ( fn );
  SetupModelMatrix ( model_rot_axis, model_rot_angle );
  InitViewMatrix ();
  NotifyViewerPos ();
  InitLights ();
  ResizeMyWorld ( width, height );
  SetWindowTitle ( fn );
} /*InitMyWorld*/
```

Ponieważ polecenie wywołania aplikacji może mieć wiele argumentów (które mogą mieć istotne znaczenie dla procedury `glutInit`, wywoływanej w celu utworzenia okna), uznałem, że nazwa pliku powinna być poprzedzona argumentem `-i`. Zatem procedura `Input-`

`Filename` wyszukuje taki napis w tablicy `argv` i podaje adres następnego napisu, ale jeśli go nie znajdzie, to podaje wskaźnik pusty. Wtedy aplikacja wypisuje wskazówkę dla użytownika (linia 22) i zatrzymuje się.

Pozostałe zmiany dokonane w procedurze `InitMyWorld` są takie: usunąłem zmienne `icos_vao` i `icos_vbo`; w ich miejsce wprowadziłem zmienne pokazane na początku listingu 14.7. Zmienne `smf_nv` i `smf_ntr` służą do przechowania liczb wierzchołków i trójkątów obiektu, w zmiennych `smf_vao` i `smf_vbo` będą pamiętane identyfikatory obiektu tablicy wierzchołków (VAO) i buforów z wierzchołkami i ich numerami, a w tablicy `smf_mm` znajdzie się macierz przekształcenia modelu, której rola i konstrukcja są opisane dalej. Jedyną zmianą w procedurze `InitMyWorld` jest zastąpienie wywołania (usuniętej) procedury `ConstructIcosahedronVAO` wywołaniem nowej procedury `ConstructSMFObject`, przy czym (z powodów wyjaśnionych dalej) musi ono nastąpić *przed* wywołaniem procedury `SetupModelMatrix`.

Wywołana w linii 40 procedura powoduje umieszczenie na ramce okna aplikacji napisu składającego się z nazwy aplikacji i nazwy pliku, którego zawartość widnieje w oknie. Sposób, w jaki to jest robione, jest opisany dalej.

Listing 14.8 przedstawia procedurę konstrukcji VAO dla obiektu przeczytanego z pliku i procedurę rysowania tego obiektu. Jeśli procedura `ReadSMFFile` przeczytała z pliku jakieś dane (czyli co najmniej jeden trójkąt), o czym zawiadamia podając niezerową wartość, to w liniach 7–19 jest w znany nam już sposób tworzona reprezentacja tego obiektu w pamięci GPU. Bufor, którego identyfikator jest pamiętany w zmiennej `smf_vbo[0]`, zawiera współrzędne wierzchołków (trójki liczb typu `GLfloat`), a bufor `smf_vbo[1]` przechowuje indeksy wierzchołków kolejnych trójkątów (trójki liczb typu `GLuint`). Nie tworzymy trzeciego bufora z kolorami wierzchołków, zamiast tego w linii 16 przypisujemy zmiennej statycznej wartość, która określa kolor wszystkich wierzchołków.

W linii 20 jest wywoływana procedura `FindSMFMMatrix`, której zadaniem jest skonstruowanie odpowiedniej macierzy przekształcenia modelu, a w kolejnych dwóch liniach otrzymane od procedury `ReadSMFFile` tablice z danymi są zwalniane — po przesłaniu do pamięci GPU dane te w RAM CPU są niepotrzebne.

Procedura `DrawSMFObject` przywiązuje do odpowiednich celów VAO z tablicą wierzchołków. Następnie, zależnie od wartości parametru `option`, wybiera odpowiedni program szaderów (przystosowany do rysowania z oświetleniem lub bez, przy czym do wyświetlania punktów jest używany tylko program bez oświetlenia). Następnie procedura wydaje polecenie narysowania wierzchołków albo trójkątów, przy czym po wywołaniu procedury `glPolygonMode` z parametrem `GL_LINE` będą rysowane krawędzie trójkątów[4]. Opcję — co ma być rysowane — użytkownik może przełączać, naciskając klawisze z literami `W`, `K` i `S`.

Zanim jednak będzie można przystąpić do rysowania, trzeba zadbać o odpowiednie przekształcenie obiektu, który może być bardzo mały lub bardzo duży. Można postąpić na dwa sposoby: przystosowć obiekt do ustalonego rzutowania lub przystosować rzutowanie do obiektu. Pokazana na listingu 14.9 procedura `FindSMFMMatrix` służy do realizacji pierwszego sposobu; w tablicy wierzchołków znajduje minimalne i maksymalne współrzędne x,

[4] W ten sposób każda wspólna krawędź trójkątów zostanie narysowana więcej niż raz.

Listing 14.8. Procedury konstrukcji i rysowania obiektu

C

```
void ConstructSMFObject ( char *fn )
{
  GLfloat *vc;
  GLuint  *trv;

  if ( ReadSMFFile ( fn, &smf_nv, &vc, &smf_ntr, &trv ) ) {
    glGenVertexArrays ( 1, &smf_vao );
    glBindVertexArray ( smf_vao );
    glGenBuffers ( 2, smf_vbo );
    glBindBuffer ( GL_ARRAY_BUFFER, smf_vbo[0] );
    glBufferData ( GL_ARRAY_BUFFER, smf_nv*3*sizeof(GLfloat),
                   vc, GL_STATIC_DRAW );
    glEnableVertexAttribArray ( 0 );
    glVertexAttribPointer ( 0, 3, GL_FLOAT, GL_FALSE,
                            3*sizeof(GLfloat), (GLvoid*)0 );
    glVertexAttrib4f ( 1, 0.5, 0.8, 1.0, 1.0 );
    glBindBuffer ( GL_ELEMENT_ARRAY_BUFFER, smf_vbo[1] );
    glBufferData ( GL_ELEMENT_ARRAY_BUFFER, 3*smf_ntr*sizeof(GLuint),
                   trv, GL_STATIC_DRAW );
    FindSMFMMatrix ( smf_nv, vc, smf_mm );
    free ( vc );
    free ( trv );
    glBindVertexArray ( 0 );
    ExitIfGLError ( "ConstructSMFObject" );
  }
  else
    ExitOnError ( "Pliku nie udalo sie przeczytac" );
} /*ConstructSMFObject*/

void DrawSMFObject ( int option )
{
  glBindVertexArray ( smf_vao );
  switch ( option ) {
case 0:
    glPointSize ( 5.0 );
    glUseProgram ( program_id[0] );
    glDrawArrays ( GL_POINTS, 0, smf_nv );
    break;
default:
    if ( option == 1 )
      glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
    else
      glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
    glUseProgram ( program_id[enlight ? 1 : 0] );
    glDrawElements ( GL_TRIANGLES, 3*smf_ntr, GL_UNSIGNED_INT, (GLvoid*)0 );
```

```
46:     break;
47:   }
48:   glBindVertexArray ( 0 );
49:   ExitIfGLError ( "DrawSMFObject" );
50: } /*DrawSMFObject*/
```

y i z, czyli reprezentację prostopadłościennej kostki opisanej na obiekcie. Na jej podstawie procedura konstruuje przekształcenie afiniczne będące jednokładnością, która skaluje kostkę tak, aby otrzymać prostopadłościan o najdłuższym boku długości 2 i przesuwa ją tak, aby środek kostki znalazł się w początku układu współrzędnych. Macierz tego przekształcenia zostaje zapamiętana w zmiennej `smf_mm` (listing 14.7, linia 2). Procedura `SetupModelMatrix` ma teraz dodatkowe zadanie. Przekształcenie obiektu, które opisuje przejście od układu modelu do układu świata, jest złożeniem tej jednokładności z obrotem określonym w animacji. Dlatego do procedury `SetupModelMatrix` została dopisana instrukcja mnożenia macierzy tych przekształceń, a do dalszych działań jest przekazywany ich iloczyn.

Listing 14.9. Procedury przekształceń modelu

C

```
1: void SetupModelMatrix ( float axis[3], double angle )
2: {
3:   GLfloat m[16], mm[16];
4:
5:   M4x4RotateVf ( m, axis[0], axis[1], axis[2], angle );
6:   M4x4Multf ( mm, m, smf_mm );
7:   LoadMMatrix ( &trans, mm );
8: } /*SetupModelMatrix*/
9:
10: void FindSMFMMatrix ( int smf_nv, GLfloat *vc, GLfloat smf_mm[16] )
11: {
12:   GLfloat xmin, xmax, ymin, ymax, zmin, zmax, dx, dy, dz, s, sm[16], tm[16];
13:   int     i, j;
14:
15:   if ( smf_nv > 1 ) {
16:     xmin = xmax = vc[0];
17:     ymin = ymax = vc[1];
18:     zmin = zmax = vc[2];
19:     for ( i = 1, j = 3;  i < smf_nv;  i++, j += 3 ) {
20:       if ( vc[j] < xmin )        xmin = vc[j];
21:       else if ( vc[j] > xmax )   xmax = vc[j];
22:       if ( vc[j+1] < ymin )      ymin = vc[j+1];
23:       else if ( vc[j+1] > ymax ) ymax = vc[j+1];
24:       if ( vc[j+2] < zmin )      zmin = vc[j+2];
25:       else if ( vc[j+2] > zmax ) zmax = vc[j+2];
26:     }
27:     dx = 0.5*(xmax-xmin);
28:     dy = 0.5*(ymax-ymin);
```

```
29:     dz = 0.5*(zmax-zmin);
30:     s = dx > dy ? dx : dy;
31:     s = dz > s  ? dz : s;
32:     if ( s > 0.0 ) {
33:       s = 1.0/s;
34:       M4x4Scalef ( smf_mm, s, s, s );
35:       M4x4MTranslatef ( smf_mm, -0.5*(xmin+xmax), -0.5*(ymin+ymax),
36:                         -0.5*(zmin+zmax) );
37:       return;
38:     }
39:   }
40:   M4x4Identf ( smf_mm );
41: } /*FindSMFMMatrix*/
```

Procedura `SetWindowTitle`, wywołana w ostatniej instrukcji procedury `InitMyWorld`, ma na celu utworzenie odpowiedniego napisu i spowodowanie umieszczenia go na ramce okna, przy czym ta procedura, należąca do części graficznej aplikacji, nie powinna być związana z systemem okien (czyli nie powinna bezpośrednio wywoływać procedury `glutSetWindowTitle`). Rozwiązaniem jest rozszerzenie interfejsu mięczy częściami graficzną i okienkową. Przedstawiona na listingu 14.10 procedura ma obsługiwać wszelkie komunikaty od części graficznej wymagajace zrobienia czegoś z oknem aplikacji lub podjęcia innych działań zależnych od środowiska. Repertuar tych komunikatów można rozszerzyć stosownie do potrzeb.

Listing 14.10. Procedura `ProcessWorldRequest`

C

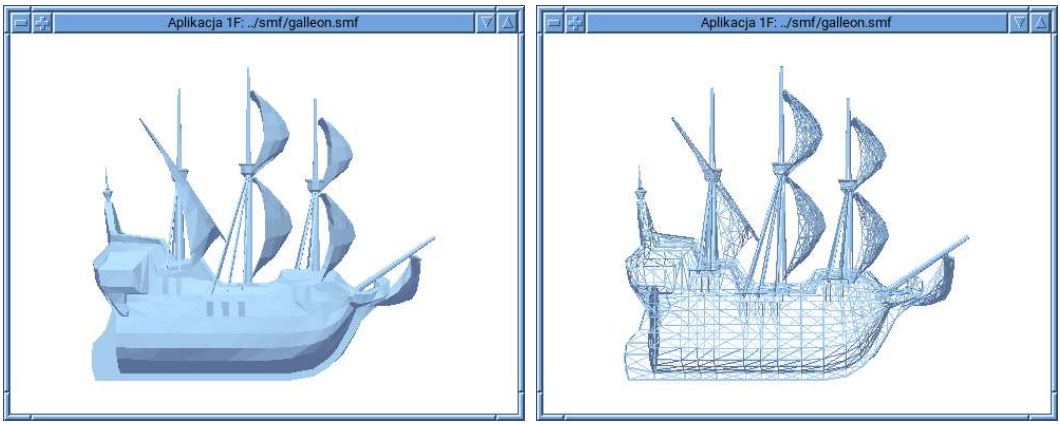
```
1: #define WMSG_SET_TITLE 1
2:
3: char ProcessWorldRequest ( int msg, void *data, void *reply )
4: {
5:   switch ( msg ) {
6: case WMSG_SET_TITLE:
7:     glutSetWindowTitle ( (char*)data );
8:     return true;
9: default:
10:    return false;
11:  }
12: } /*ProcessWorldRequest*/
```

Ostatnie dwie zmiany, których zamieszczanie tu na listingu byłoby przesadą, to zastąpienie w procedurze `RedrawMyWorld` wywołania procedury `DrawIcosahedron` wywołaniem procedury `DrawSMFObject` i stosowna modyfikacja procedury `DeleteMyWorld`.

Listing 14.11. Procedura `SetWindowTitle`

C

```
1: void SetWindowTitle ( char *fn )
2: {
3:   char s[80];
4:   int  l;
5:
6:   if ( (l = strlen ( fn )) > 65 )
7:     fn = &fn[l-65];
8:   sprintf ( s, "Aplikacja 1F: %s", fn );
9:   ProcessWorldRequest ( WMSG_SET_TITLE, (void*)s, NULL );
10: } /*SetWindowTitle*/
```



Rysunek 14.1. Okno aplikacji pierwszej F

14.3. Ćwiczenia

1. Przeczytaj opis formatu plików SMF [57] i rozbuduj procedurę czytającą te pliki o wprowadzanie dodatkowych danych: kolorów wierzchołków i wektorów normalnych. Następnie rozszerz aplikację tak, aby te dane były używane podczas wyświetlania obiektów i wypróbuj ją w działaniu.
2. Utwórz tablicę indeksów wierzchołków, w której kolejne pary liczb określają końce krawędzi trójkątów przeczytanych z pliku SMF, przy czym każda krawędź powinna mieć tylko jedną reprezentację w tej tablicy. Użyj tej tablicy do narysowania krawędzi.

   Wskazówka: W każdej parze pierwsza liczba powinna być mniejsza, po posortowaniu par w tablicy można wyeliminować powtarzające się pary.
3. Jeśli liczba wierzchołków obiektu nie przekracza 2^{16}, to można oszczędzić połowę miejsca w pamięci GPU zajmowanego przez indeksy tych wierzchołków. W tym celu trzeba

przepisać indeksy do tablicy, której elementy są typu `GLushort`, a następnie przesłać do bufora zawartość tej tablicy i w wywołaniu procedury `glDrawElements` podawać parametr `GL_UNSIGNED_SHORT`. Zmień aplikację tak, aby dokonywała tej oszczędności.

4. *Napisz procedurę (w języku C), która przeczytane z pliku SMF trójkąty o wspólnych krawędziach połączy w możliwie długie taśmy trójkątowe, tj. utworzy tablicę indeksów umożliwiającą narysowanie możliwie wielu trójkątów w trybie `GL_TRIANGLE_STRIP` zamiast `GL_TRIANGLES`. Teoretycznie można w ten sposób tablicę indeksów skrócić nawet prawie o 2/3, ale i kilkuprocentowa oszczędność miejsca jest nie do pogardzenia[5].

5. *Połącz znalezione w drugim ćwiczeniu krawędzie w możliwie długie łamane, aby jak najbardziej skrócić tablicę indeksów wierzchołków używaną do rysowania krawędzi. Korzystając z restartu prymitywu (zobacz p. 12.4.2), rysuj wszystkie krawędzie za pomocą jednego wywołania procedury `glDrawElements`.

6. *Aby narysować krawędzie z oświetleniem, trzeba dla każdego wierzchołka podać wektor normalny. Jeśli wektorów normalnych nie ma w pliku SMF, to dobierz do każdego wierzchołka wektor normalny, obliczając średni wektor normalny trójkątów, których to jest wierzchołek.

7. *Oprogramuj możliwość przeczytania opisów obiektów z kilku plików SMF i przedstawienia ich na jednym obrazie — w położeniach i wielkościach określonych indywidualnie; zaprojektuj w tym celu format (tj. składnię języka) skryptu, który będzie zawierał nazwy plików SMF do przeczytania i reprezentacje przekształceń obiektów opisanych w tych plikach. Napisz interpreter skryptu w takim formacie i zastosuj go w odpowiednio rozbudowanej aplikacji.

8. *Oprogramuj możliwość przeczytania ze skryptu opisu łańcucha kinematycznego (zobacz rozdz. 13), do którego członów mają być doczepione obiekty przeczytane z plików SMF. Zrealizuj animację łańcucha.

[5] W „dużych" aplikacjach, wyświetlających sceny zbudowane z wielu skomplikownych obiektów, takie oszczędności są istotne: nie każda karta graficzna jest wyposażona w 48 GB pamięci. Na przykład moja ma tylko 24 GB.

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 I daje podstawy, w tym:

- opis najprostszej aplikacji rysującej jeden trójkąt,
- opis potoku przetwarzania grafiki i zadań wykonywanych przez jego kolejne etapy,
- opisy wszystkich prymitywów geometrycznych wprowadzanych do potoku,
- opis bibliotek realizujących standard OpenGL i zapewniających współpracę aplikacji interakcyjnych ze środowiskiem,
- opis sposobów kompilowania szaderów i przygotowania do pracy programów działających na karcie graficznej,
- opis przekształceń geometrycznych, którym są poddawane rysowane obiekty,
- opis prostej aplikacji rysującej bryłę wielościenną i opis wielu sposobów zmieniania takich brył w inne obiekty,
- opis języka GLSL, w tym konstrukcji językowych, wbudowanych zmiennych interfejsu i podprogramów z biblioteki standardowej GLSL-a,
- opis modelu oświetlenia Lamberta,
- opis interpolacji atrybutów wierzchołków dokonywanej podczas rasteryzacji,
- opis najprostszego sposobu wyświetlania tekstu,
- opis sposobu realizacji łańcuchów kinematycznych z prostym zastosowaniem,
- opis sposobu czytania opisu obiektów z plików i rysowania tych obiektów,
- opis sposobów znajdowania błędów w tworzonej aplikacji za pomocą kontekstów uruchomieniowych i za pomocą przechwytywania wywołań procedur OpenGL-a.

Cz. I–III
ISBN 978-83-971793-0-1

Cz. I
ISBN 978-83-971793-1-8