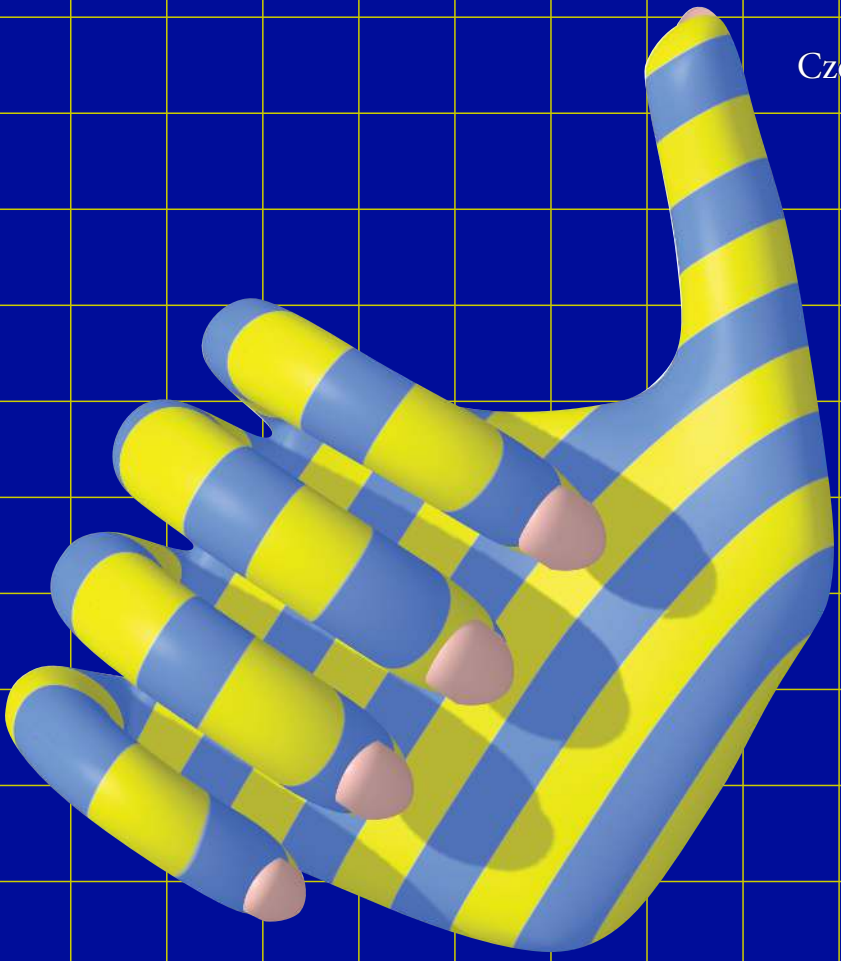
Przemysław Kiciak

Wydanie II poprawione, poszerzone i pogłębione

OpenGL i GLSL

(nie taki krótki kurs)

Część III



WSA

OpenGL i GLSL

Przemysław Kiciak

Wydanie II
poprawione,
poszerzone
i pogłębione

OpenGL i GLSL

(nie taki krótki kurs)

Część III

WSA

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

ISBN 978-83-971793-3-2 część III
ISBN 978-83-971793-0-1 części I–III

Wydanie II
Warszawa 2024

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

PDF: 19 września 2024 , 4168822 B.

Spis treści części III

30

Graficzny interfejs użytkownika

Interfejs użytkownika opisanych dotąd aplikacji, mówiąc delikatnie, pozostawia co nieco do życzenia: wszystkie polecenia oprócz zmieniania wymiarów okna i położenia obserwatora użytkownik wydaje, naciskając jakiś klawisz. Nie da się w ten sposób wygodnie wprowadzać wielkości analogowych, takich jak parametry oświetlenia lub parametry artykulacji, a zresztą klawiatura bywa potrzebna do wprowadzania napisów (liczb, nazw plików itp.), a wtedy użytkownik powinien na bieżąco widzieć, co pisze. Dlatego w bardziej skomplikowanych aplikacjach potrzebny jest **graficzny interfejs użytkownika** (*GUI, graphical user interface*), czyli rozmaite **wihajstry** (*widgets*), które użytkownik widzi w oknie i za których pośrednictwem może wprowadzać dane i wydawać polecenia. Niestety, biblioteka FreeGLUT ma tylko bardzo ograniczony i niedziałający poprawnie z nowym OpenGL-em (zobacz p. 3.1.2) zestaw procedur realizujących GUI, a w bibliotece GLFW nie ma nawet tego.

Mój kłopot polega na tym, że nie chcę zbytnio oddalać się od kursu OpenGL-a, a jednocześnie nie chcę narażać Czytelników na studiowanie kiepskiego opisu jakiejś biblioteki GUI, której akurat nie mają i z rozmaitych powodów nie mogą sobie zainstalować. Oczywiście, można stworzyć znakomity GUI w aplikacji FreeGLUT-a lub GLFW, w którym wihajstry rysuje OpenGL, ale (wobec konieczności dostarczenia odpowiednich szaderów i utworzenia buforów z danymi opisującymi wihajstry) jest to dużo bardziej pracochłonne niż pouczające. Jeśli więc obrazy wihajstrów nie przedstawiają skomplikowanych obiektów trójwymiarowych, to łatwiej jest użyć jakichś procedur grafiki dwuwymiarowej i rysować wihajstry w (znacznie prostszym do użycia) trybie natychmiastowym. Obrazy większości wihajstrów są na tyle nieskomplikowane, że czas ich rysowania będzie niezauważalny.

Opisana w rozdziałach 32–36 aplikacja ma dwa warianty, natywne dla systemów X Window oraz Windows i korzystające z GUI zrealizowanego przy użyciu procedur dostępnych w danym systemie. W pierwszym wariancie wihajstry są rysowane za pomocą procedur z biblioteki X11 [11], a w drugim przy użyciu biblioteki GDI [19]. Sporo wysiłku włożyłem w to, aby API obu wersji GUI był taki sam[1]. Dzięki temu, choć sposoby tworzenia okien i obsługi komunikatów X Window i Windows są inne, części graficzne wariantów aplikacji dla obu systemów są identyczne. Temu służy „tłumaczenie" komunikatów otrzymanych od

[1] co prawie mi się udało

systemu na komunikaty zdefiniowane w pliku nagłówkowym `xwidgets.h` i w szczególności zamienianie kodów klawiszy specjalnych `Home`, `Delete`, `F1` itd. na odpowiednie stałe symboliczne. Ponieważ jednak tematyka współpracy aplikacji z systemem okien jest odległa od OpenGL-a, w książce zamieściłem tylko opis implementacji GUI dla X Window.

30.1. Struktury danych i procedury podstawowe

Do zrealizowania wihajstra potrzebne są dwie procedury[2]. Pierwsza z nich przetwarza wejście (tj. reaguje na komunikaty o działaniach użytkownika), a druga wyświetla odpowiedni obraz w oknie, aby użytkownik widział, gdzie ma umieścić kursor przed naciśnięciem przycisku myszy albo w jakim wihajster jest stanie (np. czy wihajster — przełącznik — jest w danej chwili włączony).

Listing 30.1 przedstawia typy danych zdefiniowane w celu zaimplementowania GUI. Każdy wihajster jest opisany przez strukturę typu `xwidget`, której pola to: `id` — identyfikator wihajstra, `r` — opis prostokąta zajmowanego przez wihajster w oknie, `state` — stan wihajstra, `input` i `redraw` — wskaźniki procedury przetwarzającej komunikaty wejściowe i procedury rysującej wihajster, `wm` — wskaźnik struktury menu okna, w którym wihajster ma się pojawić, `link` — para wskaźników tworzących listę wihajstrów tego menu, oraz `data0`, `data1` — wskaźniki danych specyficznych dla wihajstra konkretnego rodzaju.

Struktura typu `xwinmenu` reprezentuje zbiór wihajstrów należących do danego okna (lub podokna) utworzonego przez system X Window. Pole `window` jest identyfikatorem okna. Pole `pixmap` zawiera identyfikator **kanwy** (*pixmap*), na której odbywa się rysowanie wihajstrów; można by je rysować bezpośrednio w oknie, ale choć to zabiera znikomy czas, byłoby widoczne migotanie (spowodowane wyświetlaniem w oknie obrazów niedokończonych). Dlatego wihajstry mają być rysowane na tej kanwie, a jej zawartość będzie kopiowana do okna, gdy obrazy wszystkich wihajstrów będą gotowe. Pole `r` opisuje wymiary okna (i kanwy). W polach `prevx`, `prevy` i `prevmask` będą pamiętane położenia kursora w oknie i stan przycisków myszy *po* zakończeniu obsługi komunikatu, aby można było ich użyć podczas obsługi następnego komunikatu. Pole `changed` ma przypisywaną wartość niezerową, gdy któryś wihajster lub aplikacja sygnalizuje potrzebę odświeżenia obrazu w oknie. Pole `expose_sent` służy do tego, aby zapobiegać wysyłaniu do okna niepotrzebnych komunikatów `Expose` podczas przetwarzania komunikatów o przesunięciu kursora, co będzie wyjaśnione dalej. Wskaźnik `data` jest przeznaczony do użytku aplikacji. Pole `wlist` jest nagłówkiem listy dwukierunkowej wihajstrów. Pole `redraw` jest wskaźnikiem procedury rysującej zawartość okna, czyli tło i na nim wszystkie wihajstry. Procedura ta ma używać do rysowania *albo* procedur systemu X Window (z biblioteki X11), *albo* OpenGL-a, przy czym ten sam sposób rysowania w oknie obowiązuje procedury rysujące *wszystkie* wihajstry w tym oknie. W tym rozdziale opisałem tylko najprostsze przykłady wihajstrów rysowanych przez procedury X11, ale trzecia aplikacja otworzy okno z wihajstrami, którego cała zawartość jest rysowana przez OpenGL-a.

[2] W języku C++ wihajster powinien być obiektem z dwiema metodami wirtualnymi.

Listing 30.1. Typy danych dla systemu wihajstrów

C

```
#define WDGSTATE_DEFAULT  0
#define WDGSTATE_INACTIVE 1

typedef struct {
    struct xwidget *prev, *next;
  } xwlink;

typedef struct xwidget {
    int              id;
    XRectangle       r;
    int              state;
    char             (*input)(struct xwidget *wdg,
                              int msg, int key, int x, int y);
    void             (*redraw)(struct xwidget *wdg);
    struct xwinmenu *wm;
    xwlink           link;
    void             *data0, *data1;
  } xwidget;

typedef struct xwinmenu {
    Window       window;
    Pixmap       pixmap;
    XRectangle   r;
    int          prevx, prevy;
    unsigned int prevmask;
    char         changed, expose_sent;
    void         *data;
    xwidget      *empty, *focus;
    XEvent       *ev;
    xwlink       wlist;
    void         (*redraw)(struct xwinmenu *wm);
    void         (*callback)(struct xwidget *wdg,
                             int msg, int key, int x, int y);
  } xwinmenu;
```

Wihajstry w oknie są połączone w listę dwukierunkową, której uporządkowanie odpowiada kolejności rysowania: pierwszy element jest „na samym spodzie", a ostatni „na samym wierzchu" stosu wihajstrów, a zatem jeśli poszczególne wihajstry nakładają się, to element „wyżej" (czyli położony dalej w liście) zasłania wihajster pod spodem. Podczas odświeżania obrazu w oknie wihajstry są rysowane „od dołu do góry", czyli od pierwszego do ostatniego. Natomiast kolejność wyszukiwania wihajstra, do którego ma trafić komunikat o zdarzeniu, które miało miejsce, gdy kursor był w pewnym punkcie okna, jest „od góry do dołu", bo komunikat ma trafić do wihajstra, który we wskazanym punkcie jest widoczny.

Procedura przetwarzania wejścia wihajstra ma poinformować, czy komunikat został przetworzony, czy nie, podając odpowiednio niezerową wartość powrotną albo zero. W tym

ostatnim przypadku lista wihajstrów będzie przeszukiwana dalej, w celu znalezienia innego wihajstra zainteresowanego tym komunikatem.

Pierwszym elementem listy wihajstrów w menu jest pusty wihajster o zerowych wymiarach. Jest on dodatkowo wskazywany przez pole `empty` i przydaje się jako parametr opisanej dalej procedury wskazywanej przez pole `callback`. Pole `focus` ma zwykle wartość `NULL`, ale wihajstry mogą na pewien czas przypisywać mu swój adres. Wtedy kolejne komunikaty (do odwołania, czyli do ponownego przypisania wartości `NULL`) będą wysyłane do tego wihajstra. Jeśli na przykład użytkownik manipuluje suwakiem i przesunie kursor poza jego obszar, to suwak nadal ma otrzymywać komunikaty do chwili, gdy użytkownik zwolni przycisk myszy, w odpowiedzi na co suwak wyłączy tryb manipulowania sobą.

Parametry procedur przetwarzania wejścia wihajstrów opisują uproszczony komunikat, przy czym opis ten w większości przypadków jest wystarczający do wykonania właściwej reakcji wihajstra na zdarzenie. Pole `ev` wskazuje strukturę typu `XEvent` z pełną informacją o komunikacie dostarczoną przez system X Window, na wypadek gdyby taka informacja była potrzebna.

Pole `callback` wskazuje procedurę aplikacji, która ma być wywoływana przez wihajstry w celu powiadomienia na przykład o naciśnięciu guzika lub przesunięciu suwaka. Procedura ta jest wywoływana także w razie nieprzetworzenia komunikatu wejściowego przez żaden wihajster w oknie albo w razie otrzymania komunikatu takiego jak `ClientMessage`. W takich przypadkach pierwszy parametr tej procedury ma wartość pola `empty`.

Listing 30.2 przedstawia procedury tworzenia, rysowania zawartości i likwidacji menu, tj. listy wihajstrów dla okna. Procedura `WinMenuRedraw`, posługując się procedurami z biblioteki X11, rysuje tło, a następnie wywołuje procedurę rysowania po kolei dla wszystkich wihajstrów z wyjątkiem nieaktywnych w danym momencie. Tło i wihajstry są rysowane na kanwie, której identyfikator jest wartością pola `pixmap`, przy użyciu kontekstu graficznego utworzonego przez aplikację, która jego identyfikator zapamiętała w zmiennej `xgc`.

Procedura `NewWinMenu` rezerwuje strukturę danych menu i zapisuje w jej polach odpowiednie informacje, w szczególności tworzy pusty wihajster, który staje się pierwszym elementem listy. Wywołując tę procedurę, aplikacja może podać parametr `redraw` pusty (`NULL`) i wtedy procedurą rysującą w tym oknie stanie się procedura `WinMenuRedraw`. Aplikacja może też podać adres innej procedury rysującej, która jeśli korzysta z OpenGL-a, to wszystkie wihajstry w tym menu muszą być rysowane za jego pomocą. Parametr `callback` musi być adresem procedury w aplikacji, która będzie wywoływana za każdym razem, gdy wihajster ma dla aplikacji komunikat, albo gdy menu przekazuje aplikacji komunikat od systemu X Window, taki jak `ConfigureNotify` lub `ClientMessage`.

Listing 30.2. Procedury tworzenia, rysowania i likwidacji menu okna

C

```
1: typedef void (*xcallback)(struct xwidget *wdg,
2:                           int msg, int key, int x, int y);
3: typedef void (*xmredraw)(struct xwinmenu *wm);
4:
5: GC xgc;
6:
```

```
7: void RedrawMenuWidgets ( xwinmenu *wm )
8: {
9:   xwidget *wdg;
10:
11:   for ( wdg = wm->wlist.next;  wdg;  wdg = wdg->link.next )
12:     if ( wdg->state != WDGSTATE_INACTIVE )
13:       wdg->redraw ( wdg );
14:   wm->expose_sent = wm->changed = false;
15: } /*RedrawMenuWidgets*/
16:
17: void WinMenuRedraw ( xwinmenu *wm )
18: {
19:   XSetForeground ( xdisplay, xgc, XWP_MENU_BACKGROUND_COLOUR );
20:   XFillRectangle ( xdisplay, wm->pixmap, xgc, 0, 0,
21:                    wm->r.width, wm->r.height );
22:   RedrawMenuWidgets ( wm );
23: } /*WinMenuRedraw*/
24:
25: xwinmenu *NewWinMenu ( Window window, int w, int h, int x, int y,
26:                        void *data, xmredraw redraw, xcallback callback )
27: {
28:   xwinmenu *wm;
29:
30:   if ( (wm = malloc( sizeof(xwinmenu) )) ) {
31:     memset ( wm, 0, sizeof(xwinmenu) );
32:     wm->window = window;
33:     wm->r.width = w;  wm->r.height = h;  wm->r.x = x;  wm->r.y = y;
34:     wm->data = data;
35:     wm->redraw = redraw ? redraw : WinMenuRedraw;
36:     if ( !redraw )
37:       wm->pixmap = XCreatePixmap ( xdisplay, window, w, h, 24 );
38:     wm->callback = callback;
39:     wm->empty = NewEmptyWidget ( wm, 0 );
40:     wm->changed = true;
41:   }
42:   return wm;
43: } /*NewWinMenu*/
44:
45: void DeleteWinMenu ( xwinmenu *wm )
46: {
47:   xwidget *wdg, *w;
48:
49:   for ( wdg = wm->wlist.next; wdg; ) {
50:     w = wdg;  wdg = w->link.next;
51:     w->input ( w, XWMSG_DELETE, 0, 0, 0 );
52:     free ( w );
53:   }
```

```
54:   if ( wm->pixmap )
55:     XFreePixmap ( xdisplay, wm->pixmap );
56:   free ( wm );
57: } /*DeleteWinMenu*/
```

Procedura `DeleteWinMenu` likwiduje kolejno wszystkie wihajstry (tj. zwalnia zajmowaną przez nie pamięć), a potem likwiduje kanwę (nieobecną w oknach z zawartością rysowaną przy użyciu OpenGL-a) i menu. Wihajster przed likwidacją jest zawiadamiany, że ona nastąpi; umożliwia to posprzątanie, jeśli na przykład utworzenie go wymagało zarezerwowania pamięci.

Procedura przedstawiona na listingu 30.3 rezerwuje pamięć na strukturę opisującą wihajster i inicjalizuje jej pola wspólne dla wszystkich wihajstrów. Nowy wihajster jest dołączany *na koniec* listy wihajstrów menu okna (zatem kolejność tworzenia wihajstrów będzie kolejnością ich rysowania). Ponadto zapamiętywane są wymiary i położenie prostokąta zajmowanego przez wihajster w oknie i nadany przez aplikację identyfikator wihajstra. Początkowa wartość pola `state` określa stan, w którym wihajster niczego szczególnego nie robi. Inne wartości będzie temu polu przypisywać procedura wskazywana przez parametr `input` lub dowolna inna procedura aplikacji. W szczególności od stanu wihajstra może zależeć jego wygląd na ekranie.

Listing 30.3. Procedura `NewWidget`

C

```
1: typedef char (*xwinput)(struct xwidget *wdg,
2:                         int msg, int key, int x, int y);
3: typedef void (*xwredraw)(struct xwidget *wdg);
4:
5: xwidget *NewWidget ( struct xwinmenu *wm, int size, int id,
6:                  int w, int h, int x, int y,
7:                  xwinput input, xwredraw redraw, void *data0, void *data1 )
8: {
9:   xwidget *wdg;
10:
11:   if ( size < sizeof(xwidget) )
12:     size = sizeof(xwidget);
13:   if ( (wdg = malloc ( size )) ) {
14:     memset ( wdg, 0, size );
15:     if ( !wm->wlist.prev )
16:       wm->wlist.prev = wm->wlist.next = wdg;
17:     else {
18:       wdg->link.prev = wm->wlist.prev;
19:       wdg->link.prev->link.next = wm->wlist.prev = wdg;
20:     }
21:     wdg->id = id;
22:     wdg->r.width = w;  wdg->r.height = h;  wdg->r.x = x;  wdg->r.y = y;
23:     wdg->input = input;  wdg->redraw = redraw;
24:     wdg->data0 = data0;  wdg->data1 = data1;
```

```
25:     wdg->wm = wm;
26:     wdg->state = WDGSTATE_DEFAULT;
27:   }
28:   return wdg;
29: } /*NewWidget*/
```

30.2. Procedury przekazujące komunikaty

Zadaniem procedury przedstawionej na listingu 30.4 jest tworzenie uproszczonych informacji na temat otrzymanych od systemu X Window komunikatów pochodzących od urządzeń wejściowych (myszy i klawiatury), co umożliwia pisanie prostszych procedur obsługi tych komunikatów. Takie informacje są przekazywane w parametrach procedury wejścia wihajstra i procedury przyjmującej polecenia wydane przez wihajstry (wskazywanej przez pole `callback` struktury typu `xwinmenu`). Informacja zawiera rodzaj komunikatu (`msg`), informację dodatkową (`key`) i współrzędne położenia kursora w oknie (`x`, `y`). Jeśli komunikat opisuje naciśnięcie lub zwolnienie przycisku myszy, to informacja dodatkowa określa, który to jest przycisk, przy czym „przyciski" 3 i 4 w X Window odpowiadają rolce myszy, w związku z czym aplikacja otrzyma komunikat `XWMSG_SCROLL`. Jeśli komunikat opisuje przesunięcie myszy, to informacja dodatkowa opisuje stan wszystkich przycisków. Jeśli został naciśnięty klawisz, to informacja dodatkowa podaje kod ASCII napisanego znaku, lub w przypadku klawisza specjalnego **symbol klawisza** (*KeySym*) przekazany w oryginalnym komunikacie zostanie zamieniony na jedną ze stałych symbolicznych zdefiniowanych w liniach 16–28. Wszystkie makrodefinicje opisujące symbole klawiszy można znaleźć w pliku `/usr/include/X11/keysymdef.h`[3]. Komunikaty inne niż pochodzące od urządzeń wejściowych otrzymują rodzaj `XWMSG_UNKNOWN`, ale aplikacja ma dostęp do oryginalnego komunikatu od systemu X Window, a dokładniej do zmiennej (wskazywanej przez pole `ev` struktury typu `xwinmenu`) opisującej ostatni otrzymany od systemu komunikat, który aplikacja właśnie przetwarza.

Listing 30.4. Procedura upraszczania komunikatów

C

```
1: #define XWMSG_NONE              0
2: #define XWMSG_UNKNOWN           1
3: #define XWMSG_ENTERING          2
4: #define XWMSG_LEAVING           3
5: #define XWMSG_BUTTON_PRESS      4
6: #define XWMSG_BUTTON_RELEASE    5
7: #define XWMSG_SCROLL            6
8: #define XWMSG_MOUSE_MOTION      7
9: #define XWMSG_KEY_PRESS         8
```

[3] Klawisze specjalne mają w systemie Windows zupełnie inne kody, ale obie implementacje GUI tłumaczą je na te same stałe symboliczne.

```
#define XWMSG_KEY_RELEASE          9
#define XWMSG_SPECIAL_KEY_PRESS   10
#define XWMSG_SPECIAL_KEY_RELEASE 11
#define XWMSG_CLIENT_MESSAGE      12
#define XWMSG_DELETE              13

#define WDGSYS_KEY_INSERT 100
#define WDGSYS_KEY_DELETE 101
#define WDGSYS_KEY_HOME   102
#define WDGSYS_KEY_END    103
#define WDGSYS_KEY_PGUP   104
#define WDGSYS_KEY_PGDN   105
#define WDGSYS_KEY_LEFT   106
#define WDGSYS_KEY_RIGHT  107
#define WDGSYS_KEY_UP     108
#define WDGSYS_KEY_DOWN   109
#define WDGSYS_KEY_F1     111
.... /* kolejne kody dla kolejnych klawiszy Fn */
#define WDGSYS_KEY_F12    122

static char btn[3] = { false, false, false };
static int  mouse_x, mouse_y;

void TranslateEventMsg ( XEvent *ev, int *msg, int *key, int *x, int *y )
{
  char   chr;
  KeySym ks;

  switch ( ev->xany.type ) {
case ButtonPress:
    switch ( *key = ev->xbutton.button ) {
  case 3: *msg = XWMSG_SCROLL;  *key = +1;  break;
  case 4: *msg = XWMSG_SCROLL;  *key = -1;  break;
 default:
      *key = ev->xbutton.button;
      btn[*key - Button1] = true;
      *msg = XWMSG_BUTTON_PRESS;
      break;
    }
    *x = mouse_x = ev->xbutton.x;  *y = mouse_y = ev->xbutton.y;
    break;
case ButtonRelease:
    if ( (*key = ev->xbutton.button) >= 3 )
      *msg = XWMSG_NONE;
    else {
      *msg = XWMSG_BUTTON_RELEASE;
      *x = mouse_x = ev->xbutton.x;  *y = mouse_y = ev->xbutton.y;
```

```
57:     }
58:     break;
59: case MotionNotify:
60:     *msg = XWMSG_MOUSE_MOTION;
61:     *key = ev->xmotion.state;
62:     *x = mouse_x = ev->xmotion.x;  *y = mouse_y = ev->xmotion.y;
63:     break;
64: case KeyPress:
65:     *msg = XWMSG_KEY_PRESS;
66:     goto decode_key;
67: case KeyRelease:
68:     *msg = XWMSG_KEY_RELEASE;
69: decode_key:
70:     XLookupString ( &ev->xkey, &chr, 1, &ks, NULL );
71:     if ( !chr ) {  /* not ASCII */
72:       *msg = ev->xany.type == KeyPress ?
73:                XWMSG_SPECIAL_KEY_PRESS : XWMSG_SPECIAL_KEY_RELEASE;
74:       switch ( ks ) {
75:     case XK_Insert:    case XK_KP_Insert: *key = WDGSYS_KEY_INSERT;  break;
76:     case XK_Delete:    case XK_KP_Delete: *key = WDGSYS_KEY_DELETE;  break;
77:     case XK_Home:      case XK_KP_Home:   *key = WDGSYS_KEY_HOME;    break;
78:     case XK_End:       case XK_KP_End:    *key = WDGSYS_KEY_HOME;    break;
79:     case XK_Page_Up:   case XK_KP_Page_Up:   *key = WDGSYS_KEY_PGUP; break;
80:     case XK_Page_Down: case XK_KP_Page_Down: *key = WDGSYS_KEY_PGDN; break;
81:     case XK_Left:      case XK_KP_Left:   *key = WDGSYS_KEY_LEFT;    break;
82:     case XK_Right:     case XK_KP_Right:  *key = WDGSYS_KEY_LEFT;    break;
83:     case XK_Up:        case XK_KP_Up:     *key = WDGSYS_KEY_LEFT;    break;
84:     case XK_Down:      case XK_KP_Down:   *key = WDGSYS_KEY_LEFT;    break;
85:     case XK_F1: case XK_KP_F1: *key = WDGSYS_KEY_F1;  break;
86:     .... /* translacja kodów kolejnych klawiszy Fn */
87:     case XK_F5:  *key = WDGSYS_KEY_F5;   break;
88:     .... /* kolejne klawisze Fn mają tylko jeden symbol */
89:     case XK_F12: *key = WDGSYS_KEY_F12;  break;
90:     default:     *key = ks;  break;
91:       }
92:     }
93:     else
94:       *key = chr;
95:     *x = ev->xkey.x;  *y = ev->xkey.y;
96:     break;
97: default:
98:     *msg = XWMSG_UNKNOWN;
99:     *x = *y = -1;
100:    break;
101:  }
102: } /*TranslateEventMsg*/
```

W odpowiedzi na komunikat Expose pokazana na listingu 30.5 procedura WinMenuInput rysuje wihajstry. Jeśli jest w użyciu kanwa X Window (pole pixmap ma wartość niezerową), to wihajstry są rysowane na niej, a następnie cały obraz z kanwy jest kopiowany do okna, przy czym rysowanie jest zbędne, jeśli pole wm->changed ma wartość false, co oznacza, że ostatnio wykonany obraz na kanwie jest aktualny. Procedura rysowania zawartości okna jest wywoływana zawsze, gdy kanwa nie jest używana (w oknie, którego zawartość ma rysować OpenGL). Komunikat ConfigureNotify powoduje zapamiętanie nowych wymiarów okna i utworzenie nowej kanwy, której wymiary są równe nowej szerokości i wysokości okna, po czym następuje wywołanie procedury callback, która może spowodować zmianę wielkości i rozmieszczenia wihajstrów w oknie. Potem do okna jest wysyłany (za pośrednictwem opisanej dalej procedury PostMenuExposeEvent) komunikat Expose, aby spowodować odświeżenie jego zawartości. Komunikat ClientMessage jest przesyłany od razu do procedury callback.

Komunikaty EnterNotify i LeaveNotify, otrzymywane, gdy kursor pojawia się w obszarze okna lub go opuszcza, są „tłumaczone" na komunikat o wejściu kursora do obszaru wihajstra lub o opuszczeniu tego obszaru. Zmienna lastinput jest wskaźnikiem wihajstra, do którego są kierowane komunikaty; jeśli kolejny komunikat wejściowy ma odbierać inny wihajster, to oba wihajstry są zawiadamiane o tej zmiane.

Listing 30.5. Procedura przesyłania komunikatów do wihajstrów

C

```
1:  static xwidget *lastinput;
2:
3:  char XYInside ( xwidget *wdg, int x, int y )
4:  {
5:    return x >= wdg->r.x && x < wdg->r.x+wdg->r.width &&
6:           y >= wdg->r.y && y < wdg->r.y+wdg->r.height;
7:  } /*XYInside*/
8:
9:  char IsButtonDown ( unsigned int button )
10: {
11:   if ( button <= Button3 )
12:     return btn[button - Button1];
13:   else
14:     return false;
15: } /*IsButtonDown*/
16:
17: void WinMenuInput ( xwinmenu *wm, XEvent *ev )
18: {
19:   int     msg, key;
20:   int     x, y;
21:   xwidget *wdg;
22:   char    inp;
23:   Window  root, child;
24:
```

```
25:   wm->ev = ev;
26:   switch ( ev->xany.type ) {
27: case Expose:
28:     if ( ev->xexpose.count == 0 ) {
29:       if ( wm->changed || !wm->pixmap ) {
30:         wm->redraw ( wm );
31:         wm->changed = wm->expose_sent = false;
32:       }
33:       if ( wm->pixmap )
34:         XCopyArea ( xdisplay, wm->pixmap, wm->window, xgc,
35:                     0, 0, wm->r.width, wm->r.height, 0, 0 );
36:     }
37:     return;
38: case ConfigureNotify:
39:     wm->r.width = ev->xconfigure.width;
40:     wm->r.height = ev->xconfigure.height;
41:     if ( wm->pixmap ) {
42:       XFreePixmap ( xdisplay, wm->pixmap );
43:       wm->pixmap = XCreatePixmap ( xdisplay, wm->window,
44:                                    wm->r.width, wm->r.height, 24 );
45:     }
46:     wm->callback ( wm->empty, WDGMSG_RECONFIGURE, 0,
47:                    wm->r.width, wm->r.height );
48:     wm->changed = true;
49:     PostMenuExposeEvent ( wm );
50:     break;
51: case ClientMessage:
52:     wm->callback ( wm->wlist.next, XWMSG_CLIENT_MESSAGE,
53:                    ev->xclient.message_type, -1, -1 );
54:     break;
55: case EnterNotify:
56:     for ( wdg = wm->wlist.prev;  wdg;  wdg = wdg->link.prev )
57:       if ( XYInside ( wdg, ev->xcrossing.x, ev->xcrossing.y ) ) {
58:         wdg->input ( wdg, XWMSG_ENTERING, 0,
59:                      ev->xcrossing.x, ev->xcrossing.y );
60:         lastinput = wdg;
61:         break;
62:       }
63:     break;
64: case LeaveNotify:
65:     if ( lastinput ) {
66:       lastinput->input ( lastinput, XWMSG_LEAVING, 0,
67:                          ev->xcrossing.x, ev->xcrossing.y );
68:       lastinput = NULL;
69:     }
70:     break;
71: case GraphicsExpose:
```

```
72: case NoExpose:
73:     wm->callback ( wm->wlist.next, XWMSG_UNKNOWN, 0, 0, 0 );
74:     break;
75: default:
76:     inp = found = false;
77:     TranslateEventMsg ( ev, &msg, &key, &x, &y );
78:     if ( (wdg = wm->focus) ) {
79:       inp = wdg->input ( wdg, msg, key, x, y );
80:       if ( !wm->focus && !XYInside ( wdg, x, y ) ) {
81:         wdg->input ( wdg, XWMSG_LEAVING, 0, x, y );
82:         lastinput = NULL;
83:       }
84:     }
85:     else {
86:       for ( wdg = wm->wlist.prev;  wdg;  wdg = wdg->link.prev ) {
87:         if ( XYInside ( wdg, x, y ) ) {
88:           found = true;
89:           if ( wdg != lastinput ) {
90:             if ( lastinput )
91:               lastinput->input ( lastinput, XWMSG_LEAVING, 0, x, y );
92:             wdg->input ( wdg, XWMSG_ENTERING, 0, x, y );
93:             lastinput = wdg;
94:           }
95:           if ( (inp = wdg->input ( wdg, msg, key, x, y )) )
96:             break;
97:         }
98:       }
99:       if ( !found && lastinput ) {
100:        lastinput->input ( lastinput, XWMSG_LEAVING, 0, x, y );
101:        lastinput = NULL;
102:      }
103:    }
104:    if ( !inp )
105:      wm->callback ( wm->wlist.next, msg, key, x, y );
106:    if ( wm->changed )
107:      PostMenuExposeEvent ( wm );
108:  }
109:  XQueryPointer ( xdisplay, wm->window, &root, &child,
110:                  &x, &y, &wm->prevx, &wm->prevy, &wm->prevmask );
111:  return;
112: } /*WinMenuInput*/
113:
114: void PostMenuExposeEvent ( xwinmenu *wm )
115: {
116:  if ( !wm->expose_sent ) {
117:    PostExposeEvent ( wm->window, wm->r.width, wm->r.height );
118:    wm->expose_sent = true;
```

```
119:   }
120: } /*PostMenuExposeEvent*/
121:
122: void GrabInput ( xwidget *wdg )
123: {
124:   wdg->wm->focus = wdg;
125: } /*GrabInput*/
126:
127: void UngrabInput ( xwidget *wdg )
128: {
129:   wdg->wm->focus = NULL;
130: } /*UngrabInput*/
```

Komunikaty `GraphicsExpose` i `NoExpose`, dla porządku, są przekazywane aplikacji, ale może ona je ignorować.

Inne komunikaty są wstępnie dekodowane przez procedurę `TranslateEventMsg`. Jeśli pole `focus` nie ma wartości `NULL`, to komunikat jest przekazywany wskazywanemu przez to pole wihajstrowi. W przeciwnym razie lista wihajstrów jest przeglądana („od góry do dołu") w poszukiwaniu wihajstra, którego prostokąt zawiera punkt wskazywany przez kursor. Jeśli wihajster nie przetworzył komunikatu, to przeglądanie listy jest kontynuowane. Jeśli żaden wihajster nie przetworzył komunikatu, to jest on przesyłany do aplikacji, tj. do procedury wskazywanej przez pole `callback`. Wihajster może zmienić swój wygląd (a także wygląd innych wihajstrów w oknie), o czym informuje, przypisując niezerową wartość polu `changed`. Powoduje to wysłanie (przez okno do siebie) komunikatu `Expose`. Po przetworzeniu komunikatu następuje wywołanie procedury `XQueryPointer`, która zapamiętuje w polach `prevx`, `prevy` i `prevmask` współrzędne położenia kursora w oknie i stan przycisków myszy.

Procedura `PostMenuExposeEvent` w celu wykonania nowego obrazu w oknie, którego dotychczasowa zawartość stała się nieaktualna, wywołuje procedurę `PostExposeEvent` (listing 3.8), ale komunikat jest wysyłany tylko wtedy, gdy pole `wm->expose_sent` ma wartość `false`. Jednocześnie z wysłaniem tego komunikatu pole to otrzymuje wartość `true`, po czym wartość `false` zostanie temu polu nadana ponownie podczas przetwarzania komunikatu `Expose`. Powodem wprowadzenia tej komplikacji jest duża częstotliwość wysyłania przez system X Window komunikatów o przemieszczeniu kursora podczas przesuwania myszy. Jeśli w odpowiedzi na przesunięcia obraz w oknie powinien się zmienić, to aplikacja może otrzymać długą serię komunikatów o przemieszczeniu kursora *przed* otrzymaniem komunikatu `Expose` wysłanego podczas obsługi pierwszego komunikatu z tej serii. Gdyby każdy komunikat z serii powodował wysłanie komunikatu `Expose`, to po serii komunikatów o przemieszczeniu kursora aplikacja dostałaby serię komunikatów `Expose`, z których pierwszy „odświeżyłby" zawartość okna, a podczas obsługi pozostałych byłby rysowany dokładnie ten sam obraz. To już miałoby zauważalny wpływ na płynność działania aplikacji, tj. opóźnienia jej reakcji na przesuwanie myszy. Dlatego komunikaty `Expose` nie są wysyłane do okna, jeśli wcześniej wysłany komunikat jeszcze jest „w drodze".

Rola i sposób używania procedur `GrabInput` i `UngrabInput` są przedstawione dalej, w opisie procedury przetwarzania komunikatów wejściowych suwaka.

30.3. Kodowanie kolorów w systemie X Window

Przed opisem procedur realizujących wihajstry, w tym rysujących je, zobaczmy sposób kodowania koloru pikseli w systemie X Window. Kolory są reprezentowane przez 32-bitowe liczby całkowite bez znaku (`unsigned int`), przy czym jeśli z oknem jest związany wizual klasy `TrueColor`, to składowe *r*, *g*, *b* piksela są reprezentowane przez spójne ciągi bitów takiej liczby. Długości i rozmieszczenie tych ciągów w liczbie mogą być różne. Dlatego na początku działania aplikacja powinna uzyskać odpowiednią informację, której będzie później używać do przetworzenia trójki liczb — składowych *r*, *g*, *b* — na odpowiednią liczbę całkowitą.

Listing 30.6 przedstawia procedurę `InitRGBXColourmap`, która powinna zostać wywołana po tym, jak zmiennej `xvii` przypisany został wskaźnik struktury dającej dostęp do wizualu (zobacz listing 3.6) lub zaraz po utworzeniu okien aplikacji. Struktura wizualu zawiera m.in. maski bitowe dla wszystkich trzech składowych. Pomocnicza procedura `parse_colourmask` na podstawie maski znajduje liczbę $2^k - 1$, gdzie k jest liczbą bitów składowej (tj. liczbą bitów o wartości 1 w masce), i położenie najmniej znaczącego bitu składowej w pikselu. Na podstawie tych informacji funkcja `RGBXColour` przetwarza trójkę liczb zmiennopozycyjnych z przedziału $[0,1]$ na liczbę 32-bitową reprezentującą zakodowany kolor.

Listing 30.6. Kodowanie koloru w X Window

C

```
static struct {
    float r_bits,  g_bits,  b_bits;
    char  r_shift, g_shift, b_shift;
  } cmap;

static void parse_colourmask ( int mask, float *bits, char *shift )
{
  char sh;

  for ( sh = 0;  !(mask & 0x01);  mask = mask >> 1, sh++ )
    ;
  *shift = sh;
  *bits = (float)mask;
} /*parse_colourmask*/

void InitRGBXColourmap ( void )
{
  parse_colourmask ( xvii->visual->red_mask, &cmap.r_bits, &cmap.r_shift );
  parse_colourmask ( xvii->visual->green_mask, &cmap.g_bits, &cmap.g_shift);
  parse_colourmask ( xvii->visual->blue_mask, &cmap.b_bits, &cmap.b_shift );
} /*InitRGBXColourmap*/

unsigned int RGBXColour ( float r, float g, float b )
{
  unsigned int ir, ig, ib;

```

```
27:   if ( r <= 0.0 )      ir = 0;
28:   else if ( r >= 1.0 ) ir = (unsigned int)cmap.r_bits;
29:   else                 ir = (unsigned int)(r*cmap.r_bits);
30:   if ( g <= 0.0 )      ig = 0;
31:   else if ( g >= 1.0 ) ig = (unsigned int)cmap.g_bits;
32:   else                 ig = (unsigned int)(g*cmap.g_bits);
33:   if ( b <= 0.0 )      ib = 0;
34:   else if ( b >= 1.0 ) ib = (unsigned int)cmap.b_bits;
35:   else                 ib = (unsigned int)(b*cmap.b_bits);
36:   return (ir << cmap.r_shift) + (ig << cmap.g_shift) + (ib << cmap.b_shift);
37: } /*RGBXColour*/
```

Procedury rysujące w bibliotece X11 posługują się **kontekstem grafiki**, tj. strukturą danych przechowującą kolor frontu i tła, grubość linii, krój pisma i wiele innych informacji potrzebnych podczas rysowania. Na początku działania aplikacja powinna utworzyć kontekst za pomocą procedury `XCreateGC` (i zapamiętać jej wartość powrotną w zmiennej `xgc` typu `GC`). Wartość tej zmiennej trzeba potem podawać jako parametr wszystkich procedur rysujących, przy czym aplikacja może utworzyć więcej niż jeden kontekst, aby oszczędzać czas na przykład przy rysowaniu wielu figur o kilku różnych kolorach.

Aby wybrać kolory frontu i tła, którymi ma być coś narysowane, trzeba wywołać procedury `XSetForeground` i `XSetBackground`. Warto, aby aplikacja na początku działania utworzyła sobie **paletę**, czyli tablicę kolorów (zakodowanych przy użyciu `RGBXColour`), które będą używane do rysowania wihajstrów. Warto też w kodzie źródłowym ponazywać te kolory zgodnie z przeznaczeniem (tzn. na przykład nie „`KOLOR_FIOLETOWY`", ale „`KOLOR_GUZIKA`"), aby w razie potrzeby łatwiej było je zmieniać i nie narobić przy tym bałaganu.

30.4. Przykłady wihajstrów

30.4.1. Wihajster pusty

Sposób realizacji wihajstrów przedstawię na czterech najprostszych przykładach. Pierwszy jest pokazany na listingu 30.7. Jest to **wihajster pusty**, który nie zabiera miejsca w oknie i nic nie robi, ale i tak jest potrzebny.

Listing 30.7. Procedury pustego wihajstra

C

```
1: static char EmptyInput ( struct xwidget *wdg,
2:                          int msg, int key, int x, int y )
3: {
4:   return false;
5: } /*EmptyInput*/
6:
7: static void EmptyRedraw ( struct xwidget *wdg )
8: { } /*EmptyRedraw*/
9:
```

```
10: xwidget *NewEmptyWidget ( xwinmenu *wm, int id )
11: {
12:   return NewWidget ( wm, sizeof(xwidget), id, 0, 0, 0, 0,
13:                      EmptyInput, EmptyRedraw, NULL, NULL );
14: } /*NewEmptyWidget*/
```

30.4.2. Guzik

Guzik (listing 30.8) jest to wihajster, który służy do wydawania aplikacji poleceń przez wskazanie kursorem i naciśnięcie przycisku myszy lub klawisza <Enter>. Obraz guzika jest obramowanym prostokątem, w którym jest napis — nazwa polecenia. Znaki tego napisu (łańcuch ASCIIZ) są w (zadeklarowanej w aplikacji) tablicy, której pierwszy element jest wskazywany przez pole data0 wihajstra. Sposób reagowania guzika na komunikaty jest chyba jasny, natomiast komentarza wymaga sposób wyświetlania tekstu: w kontekście grafiki X Window należy ustawić kolory frontu i tła i *nie należy* ustawiać kroju pisma dla znaków napisu[4]. Wtedy będzie używany domyślny krój fixed, którego znaki mają wysokość 12 pikseli i szerokość 6 pikseli; to ten krój przetworzyłem na dane w pliku font12x6.c umożliwiające wyświetlanie napisów w OpenGL-u sposobem opisanym w rozdziale 11. Guzik, po pstryknięciu, wysyła do okna aplikacji komunikat WDGMSG_BUTTON_PRESS.

Listing 30.8. Procedury wihajstra — guzika

C

```
1: #define WDGMSG_BUTTON_PRESS      15
2:
3: static char ButtonInput ( struct xwidget *wdg,
4:                           int msg, int key, int x, int y )
5: {
6:   switch ( msg ) {
7: case XWMSG_BUTTON_PRESS:
8:     if ( key == Button1 )
9:       goto issue_command;
10:    break;
11: case XWMSG_KEY_PRESS:
12:    if ( key == 0x0D ) { /* <Enter> */
13: issue_command:
14:      wdg->wm->callback ( wdg, WDGMSG_BUTTON_PRESS, 0, x, y );
15:      return true;
16:    }
17:    break;
18: default:
19:    break;
20:  }
```

[4] Chyba, że ktoś chce — służy do tego procedura XSetFont, ale wtedy trzeba wybrać taką wielkość wihajstrów, aby zmieściły się na nich potrzebne napisy i wybierać właściwy font przed każdym rysowaniem.

```
  return false;
} /*ButtonInput*/

static void ButtonRedraw ( struct xwidget *wdg )
{
  XSetForeground ( xdisplay, xgc, XWP_BUTTON_COLOUR );
  XFillRectangle ( xdisplay, wdg->wm->pixmap, xgc,
                   wdg->r.x, wdg->r.y, wdg->r.width-1, wdg->r.height-1 );
  XSetForeground ( xdisplay, xgc, XWP_TEXT_COLOUR );
  XDrawRectangle ( xdisplay, wdg->wm->pixmap, xgc,
                   wdg->r.x, wdg->r.y, wdg->r.width-1, wdg->r.height-1 );
  XSetBackground ( xdisplay, xgc, XWP_BUTTON_COLOUR );
  XDrawString ( xdisplay, wdg->wm->pixmap, xgc,
                wdg->r.x+2, wdg->r.y+13, (char*)wdg->data0,
                strlen ( (char*)wdg->data0 ) );
} /*ButtonRedraw*/

xwidget *NewButton ( xwinmenu *wm, int id,
                     int w, int h, int x, int y, char *title )
{
  return NewWidget ( wm, sizeof(xwidget), id, w, h, x, y,
                     ButtonInput, ButtonRedraw, (void*)title, NULL );
} /*NewButton*/
```

30.4.3. Przełącznik

Nieco bardziej skomplikowany jest **przełącznik**, którego procedury są na listingu 30.9. Tworząc przełącznik, aplikacja podaje wskaźniki tytułu (tj. opisu przełączanej opcji) i zmiennej typu `char`, która przyjmuje wartości 0 i 1. Pstryknięcie przełącznika powoduje zmianę dotychczasowej wartości tej zmiennej i zawiadomienie o tym fakcie aplikacji, przez wywołanie procedury `callback`. Zależnie od wartości tej zmiennej obraz przełącznika jest kwadratem, w którym nie ma nic, albo jest mniejszy biały kwadrat; tytuł przełącznika, jeśli jest obecny, jest wyświetlany obok z prawej strony.

Zmiana stanu przełącznika powoduje wysłanie do okna aplikacji komunikatu `WDGMSG_SWITCH_CHANGE`.

Listing 30.9. Procedury wihajstra — przełącznika

C

```
#define WDGMSG_SWITCH_CHANGE     16

static char SwitchInput ( struct xwidget *wdg,
                          int msg, int key, int x, int y )
{
  char *sw, s;

  switch ( msg ) {
```

```
case XWMSG_BUTTON_PRESS:
    if ( key == Button1 )
      goto issue_command;
    break;
case XWMSG_KEY_PRESS:
    if ( key == 0x0D ) { /* <Enter> */
issue_command:
      sw = ((char*)wdg->data1);  s = *sw;
      wdg->wm->callback ( wdg, WDGMSG_SWITCH_CHANGE, *sw = !s, x, y );
      wdg->wm->changed |= *sw != s;
      return true;
    }
    break;
default:
    break;
  }
  return false;
} /*SwitchInput*/

static void SwitchRedraw ( struct xwidget *wdg )
{
  char *title;

  XSetForeground ( xdisplay, xgc, XWP_SWITCH_COLOUR );
  XFillRectangle ( xdisplay, wdg->wm->pixmap, xgc,
                   wdg->r.x, wdg->r.y, wdg->r.height-1, wdg->r.height-1 );
  XSetForeground ( xdisplay, xgc, XWP_TEXT_COLOUR );
  XDrawRectangle ( xdisplay, wdg->wm->pixmap, xgc,
                   wdg->r.x, wdg->r.y, wdg->r.height-1, wdg->r.height-1 );
  if ( (title = (char*)wdg->data0) ) {
    XSetBackground ( xdisplay, xgc, XWP_MENU_BACKGROUND_COLOUR );
    XDrawString ( xdisplay, wdg->wm->pixmap, xgc,
           wdg->r.x+wdg->r.height+2, wdg->r.y+13, title, strlen ( title ) );
  }
  if ( *((char*)wdg->data1) )
    XFillRectangle ( xdisplay, wdg->wm->pixmap, xgc,
                  wdg->r.x+4, wdg->r.y+4, wdg->r.height-8, wdg->r.height-8 );
} /*SwitchRedraw*/

xwidget *NewSwitch ( xwinmenu *wm, int id,
                     int w, int h, int x, int y, char *title, char *sw )
{
  return NewWidget ( wm, sizeof(xwidget), id, w, h, x, y,
                     SwitchInput, SwitchRedraw, (void*)title, (void*)sw );
} /*NewSwitch*/
```

30.4.4. Suwak

Ostatni przykład, którego pełną implementację tu przedstawię, to **suwak**, który służy do nadawania zmiennej typu `float` wartości z przedziału $[0,1]$. Obraz suwaka jest prostokątem, wewnątrz którego jest narysowany kwadracik. Położenie tego kwadracika odpowiada wartości sterowanej przez suwak zmiennej, od 0 na końcu z lewej strony do 1 z prawej. Suwak ma dwa stany: początkowy (domyślny, `WDGSTATE_DEFAULT`) albo aktywny, `WDGSTATE_MOVING_SLIDE`. Przejście do stanu aktywnego następuje po naciśnięciu lewego przycisku myszy, gdy kursor jest w obszarze suwaka. W stanie aktywnym przesunięcie kursora powoduje obliczenie nowej wartości zmiennej sterowanej przez suwak, zawiadomienie aplikacji (przez procedurę `callback` wywołaną z drugim parametrem `WDGMSG_SLIDEBAR_CHANGE`) i spowodowanie narysowania suwaka, którego obraz odpowiada nowej wartości przywiązanej do suwaka zmiennej.

Uaktywnienie suwaka powoduje przechwycenie przez niego komunikatów wejściowych, bo użytkownik często będzie „wyjeżdżał" kursorem z obszaru suwaka, który powinien pozostawać aktywny do chwili puszczenia przycisku myszy. Dlatego, wchodząc w stan aktywny, suwak wywołuje procedurę `GrabInput` (listing 30.5), a wracając do stanu domyślnego wywołuje procedurę `UngrabInput`. W stanie aktywnym suwak ma inny kolor niż w stanie domyślnym, aby użytkownik widział, który wihajster jest aktywny. Z tego też powodu zmiana stanu powoduje przypisanie `wdg->wm->changed = true;`, którego skutkiem jest narysowanie nowego obrazu okna z wihajstrami w odpowiednich kolorach.

Listing 30.10. Procedury wihajstra — suwaka

C

```
#define WDGSTATE_MOVING_SLIDE     1
#define WDGMSG_SLIDEBAR_CHANGE   17

static char SlidebarfInput ( struct xwidget *wdg,
                             int msg, int key, int x, int y )
{
  float z, *slipos;

  slipos = (float*)wdg->data0;
  switch ( wdg->state ) {
case WDGSTATE_DEFAULT:
    switch ( msg ) {
  case XWMSG_BUTTON_PRESS:
      if ( key == Button1 ) {
        if ( x < wdg->r.x+5 ) x = (int)(wdg->r.x+5);
        else if ( x > wdg->r.x+wdg->r.width-5 )
          x = (int)(wdg->r.x+wdg->r.width-5);
        wdg->state = WDGSTATE_MOVING_SLIDE;
        wdg->wm->changed = true;
        GrabInput ( wdg );
        goto update;
```

```
22:       }
23:       break;
24:   default:
25:       break;
26:     }
27:     break;
28:
29: case WDGSTATE_MOVING_SLIDE:
30:     switch ( msg ) {
31:   case XWMSG_MOUSE_MOTION:
32:       if ( IsButtonDown ( Button1 ) ) {
33:         if ( x < wdg->r.x+5 ) x = (int)(wdg->r.x+5);
34:         else if ( x > wdg->r.x+wdg->r.width-5 )
35:           x = (int)(wdg->r.x+wdg->r.width-5);
36: update:
37:         z = (float)(x-wdg->r.x-5)/(float)(wdg->r.width-10);
38:         if ( z != *slipos ) {
39:           *slipos = z;
40:           wdg->wm->callback ( wdg, WDGMSG_SLIDEBAR_CHANGE, 0, x, y );
41:           wdg->wm->changed = true;
42:         }
43:       }
44:       else
45:         goto release;
46:       return true;
47:   case XWMSG_BUTTON_RELEASE:
48:       if ( key == Button1 ) {
49: release:
50:         wdg->state = WDGSTATE_DEFAULT;
51:         UngrabInput ( wdg );
52:         wdg->wm->changed = true;
53:         return true;
54:       }
55:       break;
56:   default:
57:       break;
58:     }
59:     break;
60:
61: default:
62:     break;
63:   }
64:   return false;
65: } /*SlidebarfInput*/
66:
67: static void SlidebarfRedraw ( struct xwidget *wdg )
68: {
```

```
69:   int   x;
70:   float *slipos;
71:
72:   slipos = (float*)wdg->data0;
73:   if ( wdg->state == WDGSTATE_MOVING_SLIDE )
74:     XSetForeground ( xdisplay, xgc, XWP_ACTIVE_SLIDEBAR_COLOUR );
75:   else
76:     XSetForeground ( xdisplay, xgc, XWP_SLIDEBAR_COLOUR );
77:   XFillRectangle ( xdisplay, wdg->wm->pixmap, xgc,
78:                    wdg->r.x, wdg->r.y, wdg->r.width-1, wdg->r.height-1 );
79:   XSetForeground ( xdisplay, xgc, XWP_TEXT_COLOUR );
80:   XDrawRectangle ( xdisplay, wdg->wm->pixmap, xgc,
81:                    wdg->r.x, wdg->r.y, wdg->r.width-1, wdg->r.height-1 );
82:   x = wdg->r.x + 2 + (int)((*slipos)*(float)(wdg->r.width - 10));
83:   XFillRectangle ( xdisplay, wdg->wm->pixmap, xgc,
84:                    x, wdg->r.y+2, 6, 6 );
85: } /*SlidebarfRedraw*/
86:
87: xwidget *NewSlidebarf ( xwinmenu *wm, int id,
88:                         int w, int h, int x, int y, float *data )
89: {
90:   return NewWidget ( wm, sizeof(xwidget), id, w, h, x, y,
91:                      SlidebarfInput, SlidebarfRedraw, (void*)data, NULL );
92: } /*NewSlidebarf*/
```

30.4.5. Edytor napisu

Ostatni wihajster w tym rozdziale opiszę skrótowo: jest to **edytor** umożliwiający wprowadzenie jednej linii tekstu, na przykład nazwy pliku do przeczytania lub zapisania. Procedura `NewLineEditor` (listing 30.11) zleca rezerwację obszaru pamięci o rozmiarze struktury `xLineEditor`, której pierwszym polem jest struktura `xwidget`. Adres bufora na tekst jest pamiętany w polu `data0` tej struktury. Pozostałe pola struktury `xLineEditor` przechowują maksymalną długość napisu, liczbę wyświetlanych znaków (krojem pisma o stałej szerokości znaku, 6 pikseli), indeks pierwszego wyświetlanego znaku i położenie kursora tekstowego.

Listing 30.11. Procedury wihajstra — edytora

C

```
1: #define WDGSTATE_EDITING           2
2: #define WDGMSG_EDITOR_ENTER       18
3: #define WDGMSG_EDITOR_ESCAPE      19
4:
5: typedef struct xLineEditor {
6:     xwidget wdg;
7:     int     maxlength,  /* maximal string length */
8:             chdisp,     /* number of characters displayed */
```

```
9:              start,      /* first character displayed */
10:             pos;        /* text cursor position */
11:   } xLineEditor;
12:
13: static char insert = true;
14:
15: void LeaveEditingState ( xwidget *wdg );
16: static char LineEditorInput ( struct xwidget *wdg,
17:                               int msg, int key, int x, int y );
18: static void LineEditorRedraw ( struct xwidget *wdg );
19:
20: xwidget *NewLineEditor ( xwinmenu *wm, int id, int w, int h, int x, int y,
21:                          int maxlength, char *txtbuf )
22: {
23:   xLineEditor *xed;
24:
25:   if ( (xed = (xLineEditor*)NewWidget ( wm, sizeof(xLineEditor), id, w, h,
26:           x, y, LineEditorInput, LineEditorRedraw, (void*)txtbuf, NULL )) ) {
27:     xed->maxlength = maxlength;
28:     xed->chdisp = (w-2)/6;
29:     xed->start = xed->pos = 0;
30:   }
31:   return (xwidget*)xed;
32: } /*NewLineEditor*/
```

Aby rozpocząć edycję, trzeba umieścić kursor na wihajstrze i nacisnąć lewy przycisk myszy. Aby edycję zakończyć, trzeba nacisnąć lewy przycisk, gdy kursor jest poza wihajstrem, lub nacisnąć klawisz `Enter`. Poza tym edytor reaguje na strzałki (w lewo i w prawo) i klawisze `Delete`, `Backspace`, `Home` i `End` i ma dwa tryby pracy, przełączane klawiszem `Insert`. O zakończeniu edycji wihajster zawiadamia aplikację, wysyłając komunikat `WDGMSG_EDITOR_ENTER`. Ponadto, jeśli użytkownik naciśnie klawisz `Esc`, wihajster wysyła komunikat `WDGMSG_EDITOR_ESCAPE`, co umożliwia aplikacji zakończenie edycji (przez wywołanie procedury `LeaveEditingState`) i na przykład odrzucenie napisu.

31

Zagęszczanie siatek

Zaimplementujemy **algorytm zagęszczania siatek** (*mesh refinement*), który generuje przybliżenia powierzchni sklejanych, będących (dalekim) uogólnieniem płatów Béziera. Siatkę, która ma stosunkowo niewielką liczbę wierzchołków (punktów kontrolnych), można dosyć łatwo kształtować, a jej zagęszczanie prowadzi do otrzymania dużej liczby trójkątów przybliżających gładką powierzchnię. Chcemy, aby zagęszczaniem zajmowała się GPU. Wyświetlaniem siatek zajmie się trzecia aplikacja, opisana w następnym rozdziale.

31.1. Definicja i warunki poprawności siatki

Siatka składa się z **wierzchołków**, **krawędzi** i **ścian**. Wierzchołek ma określone położenie w przestrzeni. Krawędź jest odcinkiem, którego końce są wierzchołkami. Ściana jest łamaną zamkniętą złożoną z co najmniej trzech krawędzi, przy czym wszystkie wierzchołki tej łamanej muszą być różnymi wierzchołkami[1].

Zakładamy, że każda krawędź należy do jednej albo dwóch ścian, przy czym w pierwszym przypadku jest to **krawędź brzegowa**, a w drugim **krawędź wewnętrzna**. Końce co najmniej jednej krawędzi brzegowej są **wierzchołkami brzegowymi**, a pozostałe wierzchołki są **wewnętrzne**. Wierzchołek brzegowy jest końcem co najmniej dwóch krawędzi[2], a wierzchołek wewnętrzny co najmniej trzech. Wszystkie ściany mające wspólny wierzchołek można odwiedzić po kolei, przechodząc przez ich wspólne krawędzie, których końcem jest ten wierzchołek.

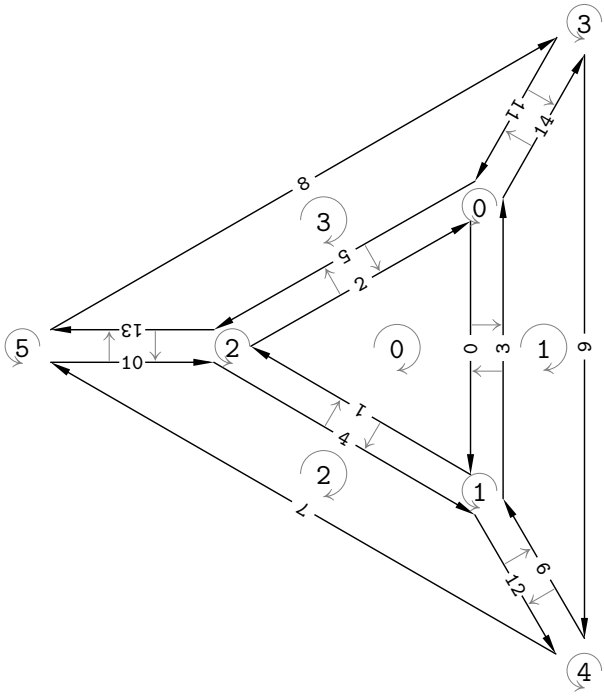
Dla wygody krawędzie będą reprezentowane przez **półkrawędzie**, przy czym krawędzi wewnętrznej odpowiadają dwie półkrawędzie powiązane w parę, a reprezentacją krawędzi brzegowej jest jedna półkrawędź bez pary. Dzięki temu każda półkrawędź należy do jednej ściany. Co więcej, półkrawędzie są zorientowane: jeden z wierzchołków krawędzi jest począt-

[1] Ale różne wierzchołki mogą mieć to samo położenie. Wierzchołki mogą mieć też dodatkowe atrybuty, na przykład wektor normalny, współrzędne tekstury i inne.

[2] Wierzchołek brzegowy jest końcem dwóch krawędzi brzegowych i pewnej (nieujemnej) liczby krawędzi wewnętrznych.

kiem półkrawędzi, a drugi jej końcem. Druga półkrawędź z pary reprezentującej krawędź wewnętrzną jest zorientowana odwrotnie. Dowolną ścianę można obejść po jej półkrawędziach zgodnie z ich orientacją[3].

Wierzchołek ma również zbiór półkrawędzi, których jest początkiem, i zbiór ten jest uporządkowany (tj. ustawiony w ciąg), co wprowadza orientację wierzchołka. Można to zrobić inaczej, ale kiedyś zaimplementowałem (na CPU) zestaw algorytmów przetwarzania siatek, dla których orientacja wierzchołka (tj. kolejność półkrawędzi wokół niego) jest odwrotna niż orientacja ściany i tak już (w moich programach) zostało. Widać to na rysunku 31.1 przedstawiającym przykład siatki; półkrawędzie tworzące pary zostały na nim porozsuwane.



Wierzchołki

```
0: 0, 14, 5
1: 1, 12, 3
2: 2, 13, 4
3: 11, 6
4: 9, 7
5: 10, 8
```

Ściany

```
0: 0, 1, 2
1: 3, 14, 6, 9
2: 4, 12, 7, 10
3: 5, 13, 8, 11
```

Półkrawędzie

```
 0: (0, 1, 0, 3)
 1: (1, 2, 0, 4)
 2: (2, 0, 0, 5)
 3: (1, 0, 1, 0)
 4: (2, 1, 2, 1)
 5: (0, 2, 3, 2)
 6: (3, 4, 1, -1)
 7: (4, 5, 2, -1)
 8: (5, 3, 3, -1)
 9: (4, 1, 1, 12)
10: (5, 2, 2, 13)
11: (3, 0, 3, 14)
12: (1, 4, 2, 9)
13: (2, 5, 3, 10)
14: (0, 3, 1, 11)
```

Rysunek 31.1. Schemat budowy siatki

W obu reprezentacjach siatki, używanych w pamięci CPU i GPU, wierzchołki, półkrawędzie i ściany są przechowywane w tablicach; identyfikatory wierzchołków, półkrawędzi i ścian są indeksami do tych tablic, tj. kolejnymi liczbami całkowitymi od 0.

Półkrawędź jest reprezentowana przez cztery identyfikatory. Pierwsze dwa są numerami wierzchołków będących początkiem i końcem półkrawędzi. Kolejna liczba to identyfikator ściany, do której należy półkrawędź, a ostatnia to identyfikator drugiej półkrawędzi z pary. Krawędzie tworzące parę przechowują nawzajem swoje identyfikatory, a ponadto

[3] Zauważmy, że to wymusza orientowalność powierzchni złożonej ze ścian wyobrażonych jako wielokąty lub błony rozpięte na krawędziach. Nie uzyskamy w ten sposób wstęgi Möbiusa.

mają te same identyfikatory wierzchołków zamienione miejscami. Taka informacja jest redundantna, ale jest ona konieczna, ponieważ półkrawędź może nie mieć pary. Wtedy reprezentuje ona krawędź brzegową, a identyfikator jej drugiej połowy jest równy `-1`; liczba `-1` pełni rolę identyfikatora pustego.

Wierzchołek oprócz położenia ma określony ciąg (identyfikatorów) półkrawędzi, których jest początkiem. Także ściana ma ciąg identyfikatorów swoich półkrawędzi, przy czym w obu przypadkach kolejność identyfikatorów musi być zgodna z orientacją: na rysunkach 31.1–31.3 kolejność półkrawędzi odpowiada obchodzeniu ściany zgodnie z ruchem wskazówek zegara, a wierzchołek jest okrążany w przeciwną stronę. Dla ściany i wierzchołka wewnętrznego nie ma znaczenia, która półkrawędź jest podana jako pierwsza, ale dla wierzchołka brzegowego wychodząca z niego półkrawędź brzegowa jest ostatnia w ciągu.

31.2. Reprezentacja siatki w pamięci RAM CPU

Reprezentacja siatki składa się z sześciu tablic. Trzy z nich (`mv`, `mhe`, `mfac`) przechowują odpowiednio struktury opisujące wierzchołki, półkrawędzie i ściany, przy czym struktury `BSMvertex` i `BSMfacet` reprezentujące wierzchołek i ścianę mają identyczną budowę (zobacz listing 31.1). Pole `degree` przechowuje liczbę półkrawędzi, których początkiem jest dany wierzchołek, lub liczbę krawędzi ściany; liczba ta jest dalej nazywana **stopniem** wierzchołka lub ściany. Pole `firsthalfedge` jest indeksem dodatkowej tablicy, w której są przechowywane identyfikatory kolejnych półkrawędzi wierzchołka lub ściany.

Listing 31.1. Struktury reprezentacji siatki

C

```
typedef struct {
    char degree;
    int  firsthalfedge;
  } BSMfacet, BSMvertex;

typedef struct {
    int v0, v1;
    int facetnum;
    int otherhalf;
  } BSMhalfedge;

typedef struct CPUmesh {
    int         nsattr, pdim, pofs, nvofs;
    int         nv, nhe, nfac;
    BSMvertex   *mv;
    BSMhalfedge *mhe;
    BSMfacet    *mfac;
    int         *mvhei, *mfhei;
    float       *vc;
  } CPUmesh;
```

Kolejne pola struktury `BSMhalfedge` przechowują identyfikatory wierzchołków — początku i końca półkrawędzi oraz ściany i drugiej półkrawędzi z pary.

Dwie wspomniane wyżej tablice dodatkowe, `mvhei` i `mfhei`, mają długość równą liczbie n_h półkrawędzi siatki. Jeśli wierzchołek albo ściana ma pola `degree` i `firsthalfedge` o wartościach d i f, to w pierwszej lub drugiej z tych tablic numery półkrawędzi są przechowywane w miejscach $f, f+1, \ldots, f+d-1$. To rozwiązanie umożliwia oszczędne reprezentowanie siatek, w których poszczególne wierzchołki lub ściany mają różne liczby półkrawędzi. W tych tablicach znajdują się pewne permutacje ciągu liczb $0, \ldots, n_h-1$ — jest to jeden z warunków poprawności reprezentacji siatki.

Współrzędne punktów położenia wierzchołków są przechowywane w osobnej tablicy `vc` liczb typu `float`[4]. Oprócz liczb n_v, n_h i n_f, odpowiednio wierzchołków, półkrawędzi i ścian siatki, trzeba podać liczbę s skalarnych atrybutów wierzchołka[5]. Długość tablicy ze współrzędnymi wierzchołków jest zatem równa sn_v.

Wprowadzona dla wygody struktura o nazwie `CPUmesh` przechowuje wskaźniki opisanych wyżej tablic i liczby s, n_v, n_h i n_f w polach `nsattr`, `nv`, `nhe` i `nfac`. Można pisać procedury przetwarzania siatek z jednym parametrem — wskaźnikiem do takiej struktury — zamiast przekazywać wszystkie jej pola jako osobne parametry. Pola `pdim`, `pofs` i `nvofs` są opisane dalej.

31.3. Reprezentacja siatki w pamięci GPU

Reprezentacja siatki w pamięci GPU podlega pewnym ograniczeniom wynikającym ze specyfikacji OpenGL-a. Istnieje limit liczby buforów magazynowych, do których szader obliczeniowy ma dostęp; specyfikacja [1] gwarantuje, że może ich być 8, zobacz p. 11.5.1. W implementacji algorytmu zagęszczania potrzebujemy mieć siatkę daną, siatkę będącą wynikiem opisanych dalej operacji podwajania lub uśredniania i tablice robocze. Dlatego siatkę w pamięci GPU umieścimy w trzech buforach magazynowych. W pierwszym z nich znajduje się tablica liczb typu `int`, której kolejne fragmenty są tablicami wierzchołków i ścian oraz tablicami zawierającymi ciągi identyfikatorów półkrawędzi poszczególnych wierzchołków i ścian. W drugim buforze umieścimy opisy półkrawędzi, do czego nadają się elementy tablicy typu `ivec4`. W trzecim buforze, zawierającym tablicę liczb typu `float`, umieścimy współrzędne wierzchołków, przy czym mogą tu też być dodatkowe atrybuty wierzchołków, takie jak wektor normalny i współrzędne tekstury. Obliczenia numeryczne wykonywane przez algorytm zagęszczania siatki poskutkują dokonaniem odpowiedniej interpolacji wszystkich tych atrybutów.

Listing 31.2 przedstawia strukturę przechowującą wymiar przestrzeni, liczby wierzchołków, półkrawędzi i ścian oraz identyfikatory czterech buforów magazynowych z opisanymi wyżej tablicami. Pokazane na listingu makrodefinicje służą do uczytelnienia kodu procedur opisanych dalej.

[4] Lub `double` — to zależy od aplikacji, ale aby użyć podwójnej precyzji, trzeba dostosować szadery.

[5] W najprostszym przypadku s jest liczbą współrzędnych położenia wierzchołka, ale będziemy używać też innych atrybutów: współrzędnych wektora normalnego i tekstury. Zobacz opis w podrozdziale 31.3.

Listing 31.2. Struktura dla CPU dająca dostęp do reprezentacji siatki w GPU

C

```
#define MVFBUF   mbuf[0]
#define MHEBUF   mbuf[1]
#define VCBUF    mbuf[2]
#define MSBUF    mbuf[3]

typedef struct GPUmesh {
    int    nsattr, pdim, pofs, nvofs;
    int    nv, nhe, nfac;
    GLuint mbuf[4];
  } GPUmesh;
```

Liczbę d półkrawędzi wierzchołka lub ściany (przechowywaną w pamięci CPU w polu `degree` struktury `BSMvertex` lub `BSMfacet`) i indeks f początku ciągu identyfikatorów (z pola `firsthalfedge`) upakujemy w 32 bitach zmiennej typu `int` za pomocą masek i przesunięć bitowych przedstawionych na listingu 31.3. Liczba f będzie przechowywana w najmniej znaczących 25 bitach, co ogranicza liczbę półkrawędzi siatki do 2^{25} (czyli ponad 32 milionów). Bit na pozycji 25 zarezerwujemy do oznaczania wierzchołka lub ściany podczas przetwarzania. Stopień wierzchołka lub ściany (czyli liczbę półkrawędzi) umieścimy w najbardziej znaczących 6 bitach (a więc liczba ta nie może przekraczać 63).

Listing 31.3. Maski bitowe i przesunięcia reprezentacji wierzchołka i ściany

C, GLSL

```
#define FHEMASK   0x01FFFFFF
#define TAGMASK   0x02000000
#define DEGMASK   0xFC000000
#define DEGSHIFT          26
```

Listing 31.4 przedstawia deklaracje buforów magazynowych w programach rysowania krawędzi i ścian siatek; programy te są zbudowane z szaderów opisanych w dalszych rozdziałach, ale blok zawierający parametry siatki jest przedstawiony w tym miejscu, bo opisane niżej procedury przesyłające reprezentację siatki między pamięcią CPU i GPU dbają o właściwą zawartość tego bloku[6].

Tablica `mvf` w bloku `meshvf` służy do przechowania czterech tablic, które w pamięci CPU mają nazwy `mv`, `mvhei`, `mfac` i `mfhei`; wszystkie ich elementy są liczbami całkowitymi. Tablica `mhe` w bloku `meshhe` zawiera opisy półkrawędzi, z których każdy składa się z czterech liczb całkowitych. Współrzędne położenia wierzchołków siatki i inne ich atrybuty (wektor normalny, współrzędne tekstury itp.) są przechowywane w tablicy `vc` w bloku `meshvc`.

[6]Opisany w podrozdziale 31.4 szader obliczeniowy, którego zadaniem jest otrzymanie siatki zagęszczonej, zamiast bloku magazynowego z liczbami elementów siatki korzysta z bloku zmiennych jednolitych zawierającego parametry obu siatek i zmienne pomocnicze. W pierwszym wydaniu książki liczby elementów siatki i inne dane trzeba było przed rysowaniem siatki przypisać zmiennym jednolitym w domyślnym bloku zmiennych jednolitych programu rysującego, czego skutkiem był bardziej skomplikowany kod aplikacji. Dlatego, przewidując, że siatki będą rysowane (a nie tylko zagęszczane), warto uprościć sobie zadanie na przyszłość.

Czwarty blok magazynowy, `meshsurf`, zawiera opis siatki potrzebny do rysowania. W jego polach `nv`, `nhe` i `nfac` są pamiętane liczby wierzchołków, półkrawędzi i ścian siatki.

Listing 31.4. Bloki magazynowe siatki w programach rysujących

GLSL

```
layout(std430,binding=0) buffer meshvf { int mvf[];   } mvf;
layout(std430,binding=1) buffer meshhe { ivec4 mhe[]; } mhe;
layout(std430,binding=2) buffer meshvc { float vc[];  } mvc;
layout(std430,binding=3) buffer meshsurf {
    int  nv, nhe, nfac, nsattr, pdim, pofs, nvofs;
    bool MeshNormals;
    vec3 Colour;
  };
```

W polu `nsattr` jest pamiętana całkowita liczba skalarnych atrybutów jednego wierzchołka, na przykład 3 lub 4, jeśli są w niej podane tylko współrzędne kartezjańskie albo jednorodne położenia wierzchołka w przestrzeni, ale jeśli jest też wektor normalny, to liczba ta wzrośnie o 3, a jeśli będą obecne dodatkowe atrybuty (np. współrzędne tekstury), to je też trzeba będzie policzyć.

Pole `pdim` przechowuje liczbę współrzędnych położenia (np. 3 lub 4), a pole `pofs` zawiera informację, którym atrybutem skalarnym jest pierwsza współrzędna położenia wierzchołka. Jeśli wektor normalny jest obecny, to jego pierwsza współrzędna jest atrybutem skalarnym o numerze podanym w polu `nvofs`. Dodanie kolejnych atrybutów (np. współrzędnych tekstury) wymaga dodania odpowiednich pól do bloku.

Procedurę `GetAccessToMeshSurfBlock` pokazaną na listingu 31.5 aplikacja musi wywołać przed przesyłaniem siatek do pamięci GPU, z parametrem, który jest identyfikatorem dowolnego programu szaderów zawierającego blok `meshsurf`[7].

Pomocnicza procedura `UploadMeshParams` przypisuje polom `nv`, `nhe`, `nfac`, `nsattr`, `pdim`, `pofs` i `nvofs` wartości pól o tych samych nazwach w reprezentującej siatkę strukturze typu `GPUmesh`. Osobne procedury przypisują wartości polom `MeshNormals` i `Colour`, których opis jest częścią opisu programów rysujących.

Makrodefinicje `SSB` i `UVB` wprowadzają synonimy długich nazw symbolicznych OpenGL-a, które dalej są potrzebne w tak wielu miejscach, że postanowiłem je skrócić.

Listing 31.5. Dostęp do bloku magazynowego `meshsurf`

C

```
#define NMBOFFS 9

#define SSB GL_SHADER_STORAGE_BUFFER
#define UVB GL_UNIFORM_BUFFER

```

[7] Ponieważ blok `meshsurf` nie ma nazwy wewnętrznej, podane w liniach 12–13 nazwy jego pól, których przesunięcia względem początku bufora ma znaleźć procedura `GetAccessToStorageBlock` nie są prefiksowane nazwą bloku. Zobacz przypis 14 na s. 192.

```
6: static GLuint bpoint = GL_INVALID_INDEX;
7: static GLint  mbsize, mbofs[NMBOFFS];
8:
9: GLuint GetAccessToMeshSurfBlock ( GLuint program_id )
10: {
11:   static const GLchar *names[] =
12:     { "meshsurf", "nv", "nhe", "nfac", "nsattr", "pdim",
13:       "pofs", "nvofs", "MeshNormals", "Colour" };
14:
15:   if ( bpoint == GL_INVALID_INDEX )
16:     GetAccessToStorageBlock ( program_id, NMBOFFS, names,
17:                               &mbsize, mbofs, &bpoint );
18:   ExitIfGLError ( "GetAccessToMeshSurfBlock" );
19:   return bpoint;
20: } /*GetAccessToMeshSurfBlock*/
21:
22: void UploadMeshParams ( GPUmesh *gmesh )
23: {
24:   glBindBuffer ( SSB, gmesh->MSBUF );
25:   glBufferSubData ( SSB, mbofs[0], sizeof(GLint), &gmesh->nv );
26:   glBufferSubData ( SSB, mbofs[1], sizeof(GLint), &gmesh->nhe );
27:   glBufferSubData ( SSB, mbofs[2], sizeof(GLint), &gmesh->nfac );
28:   glBufferSubData ( SSB, mbofs[3], sizeof(GLint), &gmesh->nsattr );
29:   glBufferSubData ( SSB, mbofs[4], sizeof(GLint), &gmesh->pdim );
30:   glBufferSubData ( SSB, mbofs[5], sizeof(GLint), &gmesh->pofs );
31:   glBufferSubData ( SSB, mbofs[6], sizeof(GLint), &gmesh->nvofs );
32:   ExitIfGLError ( "UploadMeshParams" );
33: } /*UploadMeshParams*/
```

Listing 31.6 przedstawia procedurę `ReallocGPUmesh` dokonującą rezerwacji pamięci GPU na potrzeby reprezentowania siatek o podanych liczbach wierzchołków, półkrawędzi i ścian. Jeśli identyfikatory buforów OpenGL-a w przekazanej strukturze `GPUmesh` są niezerowe, to odpowiednie bufory są zwalniane, po czym procedura tworzy nowe bufory i nadaje każdemu z nich odpowiednią wielkość, wywołując procedurę `glBufferData`. Zwracam uwagę na sposób obliczania długości buforów w liniach 10, 13 i 15. Informacje podane w parametrach są zapamiętywane w polach struktury opisującej siatkę, skąd przesłaniem do bufora z blokiem magazynowym `meshsurf` zajmuje się procedura `UploadMeshParams`.

Listing 31.6. Procedura rezerwowania pamięci GPU na reprezentacje siatek

C

```
1: char ReallocGPUmesh ( GPUmesh *gmesh, int nv, int nhe, int nfac, int nsattr,
2:                       int pdim, int pofs, int nvofs )
3: {
4:   int i;
5:
6:   for ( i = 0; i < 4; i++ )
```

```
    if ( gmesh->mbuf[i] > 0 ) glDeleteBuffers ( 1, &gmesh->mbuf[i] );
  glGenBuffers ( 4, gmesh->mbuf );
  glBindBuffer ( SSB, gmesh->MVFBUF );
  glBufferData ( SSB, (nv+nfac+2*nhe)*sizeof(GLuint),
                 NULL, GL_DYNAMIC_DRAW );
  glBindBuffer ( SSB, gmesh->MHEBUF );
  glBufferData ( SSB, nhe*4*sizeof(GLint), NULL, GL_DYNAMIC_DRAW );
  glBindBuffer ( SSB, gmesh->VCBUF );
  glBufferData ( SSB, nv*nsattr*sizeof(GLfloat), NULL, GL_DYNAMIC_DRAW );
  glBindBuffer ( SSB, gmesh->MSBUF );
  glBufferData ( SSB, mbsize, NULL, GL_DYNAMIC_DRAW );
  gmesh->nsattr = nsattr;  gmesh->pdim = pdim;  gmesh->pofs = pofs;
  gmesh->nvofs = nvofs;
  gmesh->nv = nv;
  gmesh->nhe = nhe;
  gmesh->nfac = nfac;
  UploadMeshParams ( gmesh );
  ExitIfGLError ( "ReallocGPUmesh" );
  return true;
} /*ReallocGPUmesh*/
```

Podobnie działa procedura `ReallocCPUmesh` pokazana na listingu 31.7. Zwalnia ona tablice wskazywane przez wskaźniki w strukturze typu `CPUmesh`, po czym rezerwuje (za pomocą `malloc`) nowe tablice. Liczby wierzchołków, półkrawędzi i ścian i pozostałe parametry są zapamiętywane w strukturze, ale zawartość tablic pozostaje nieokreślona. Niepowodzenie rezerwacji (wskutek braku wolnego miejsca w pamięci CPU) powoduje odwołanie rezerwacji pamięci, której udało się dokonać i jest sygnalizowane wartością powrotną `false` procedury.

Każdą zmienną typu `GPUmesh` albo `CPUmesh` przed przekazaniem jej adresu po raz pierwszy jako parametru jednej lub drugiej opisanej tu procedury trzeba wypełnić zerami.

Listing 31.7. Procedura rezerwowania pamięci CPU na reprezentacje siatek

C

```
void FreeCPUmeshTab ( CPUmesh *cmesh )
{
  if ( cmesh->mv ) free ( cmesh->mv );
  if ( cmesh->vc ) free ( cmesh->vc );
  if ( cmesh->mhe ) free ( cmesh->mhe );
  if ( cmesh->mfac ) free ( cmesh->mfac );
  if ( cmesh->mvhei ) free ( cmesh->mvhei );
  if ( cmesh->mfhei ) free ( cmesh->mfhei );
  memset ( cmesh, 0, sizeof(CPUmesh) );
} /*FreeCPUmeshTab*/

char ReallocCPUmesh ( CPUmesh *cmesh, int nv, int nhe, int nfac, int nsattr,
                      int pdim, int pofs, int nvofs )
{
  FreeCPUmeshTab ( cmesh );
```

```
16:   cmesh->mv = malloc ( nv*sizeof(BSMvertex) );
17:   cmesh->vc = malloc ( nv*nsattr*sizeof(double) );
18:   cmesh->mhe = malloc ( nhe*sizeof(BSMhalfedge) );
19:   cmesh->mfac = malloc ( nfac*sizeof(BSMfacet) );
20:   cmesh->mvhei = malloc ( nhe*sizeof(int) );
21:   cmesh->mfhei = malloc ( nhe*sizeof(int) );
22:   if ( !cmesh->mv | !cmesh->vc | !cmesh->mhe | !cmesh->mfac |
23:        !cmesh->mvhei | !cmesh->mfhei ) {
24:     FreeCPUmeshTab ( cmesh );
25:     return false;
26:   }
27:   else {
28:     cmesh->nsattr = nsattr;  cmesh->pdim = pdim;  cmesh->pofs = pofs;
29:     cmesh->nv = nv;
30:     cmesh->nhe = nhe;
31:     cmesh->nfac = nfac;
32:     return true;
33:   }
34: } /*ReallocCPUmesh*/
```

Na listingu 31.8 są pokazane procedury przesyłające reprezentację siatki między CPU a GPU, działające przy założeniu, że współrzędne wierzchołków w pamięci CPU są reprezentowane w pojedynczej precyzji (jako liczby typu `float`). Działanie obu procedur jest dość oczywiste, zwracam więc tylko uwagę na „pakowanie" i „rozpakowywanie" liczb d i f reprezentujących wierzchołek lub ścianę w liniach 20, 23, 60–61 i 65–66 oraz na obliczanie odległości (w bajtach) od początku bufora początków poszczególnych tablic (wierzchołków, ścian i tablic z ciągami indeksów półkrawędzi) w liniach 21, 24, 25, 27 (zobacz wyrażenia opisujące drugi parametr procedur `glBufferSubData`) i w liniach 58, 63, 68 i 70 (drugi parametr procedury `glGetBufferSubData`).

Listing 31.8. Procedury przesyłania siatek między CPU a GPU

C

```
1: char CPUmeshToGPU ( CPUmesh *cmesh, GPUmesh *gmesh )
2: {
3:   int       dim, nv, nhe, nfac;
4:   BSMvertex *mv;
5:   BSMfacet  *mfac;
6:   GLuint    *vf;
7:   int       i, size;
8:
9:   if ( ReallocGPUmesh ( gmesh, nv = cmesh->nv, nhe = cmesh->nhe,
10:                        nfac = cmesh->nfac, dim = cmesh->nsattr,
11:                        cmsh->pdim, cmesh->pofs, cmesh->nvofs ) ) {
12:     size = (nv > nfac ? nv : nfac)*sizeof(GLuint);
13:     i = nv*dim*sizeof(GLfloat);
14:     if ( size < i ) size = i;
```

```
15:     if ( !(vf = (GLuint*)malloc ( size )) )
16:       return false;
17:     glBindBuffer ( SSB, gmesh->MVFBUF );
18:     mv = cmesh->mv;  mfac = cmesh->mfac;
19:     for ( i = 0; i < nv; i++ )
20:       vf[i] = mv[i].firsthalfedge + (mv[i].degree << DEGSHIFT);
21:     glBufferSubData ( SSB, 0, nv*sizeof(GLuint), vf );
22:     for ( i = 0; i < nfac; i++ )
23:       vf[i] = mfac[i].firsthalfedge + (mfac[i].degree << DEGSHIFT);
24:     glBufferSubData ( SSB, nv*sizeof(GLuint), nfac*sizeof(GLuint), vf );
25:     glBufferSubData ( SSB, (nv+nfac)*sizeof(GLuint), nhe*sizeof(GLint),
26:                       cmesh->mvhei );
27:     glBufferSubData ( SSB, (nv+nfac+nhe)*sizeof(GLuint), nhe*sizeof(GLint),
28:                       cmesh->mfhei );
29:     glBindBuffer ( SSB, gmesh->MHEBUF );
30:     glBufferSubData ( SSB, 0, nhe*4*sizeof(GLint), cmesh->mhe );
31:     vc = (GLfloat*)vf;
32:     for ( i = 0; i < nv*dim; i++ )
33:       vc[i] = (GLfloat)cmesh->vc[i];
34:     glBindBuffer ( SSB, gmesh->VCBUF );
35:     glBufferSubData ( SSB, 0, nv*dim*sizeof(GLfloat), cmesh->vc );
36:     free ( vf );
37:     ExitIfGLError ( "CPUmeshToGPU" );
38:     return true;
39:   }
40:   else return false;
41: } /*CPUmeshToGPU*/
42:
43: char GPUmeshToCPU ( GPUmesh *gmesh, CPUmesh *cmesh )
44: {
45:   int       dim, nv, nhe, nfac;
46:   BSMvertex *mv;
47:   BSMfacet  *mfac;
48:   GLuint    *vf;
49:   int       i, size;
50:
51:   if ( ReallocCPUmesh ( cmesh, nv = gmesh->nv, nhe = gmesh->nhe,
52:                         nfac = gmesh->nfac, dim = gmesh->nsattr,
53:                         gmesh->pdim, gmesh->pofs, gmesh->nvofs ) ) {
54:     size = (nv > nfac ? nv : nfac)*sizeof(GLuint);
55:     if ( !(vf = (GLuint*)malloc ( size )) )
56:       return false;
57:     glBindBuffer ( SSB, gmesh->MVFBUF );
58:     glGetBufferSubData ( SSB, 0, nv*sizeof(GLuint), vf );
59:     for ( mv = cmesh->mv, i = 0;  i < nv;  i++ ) {
60:       mv[i].firsthalfedge = vf[i] & FHEMASK;
61:       mv[i].degree = vf[i] >> DEGSHIFT;
```

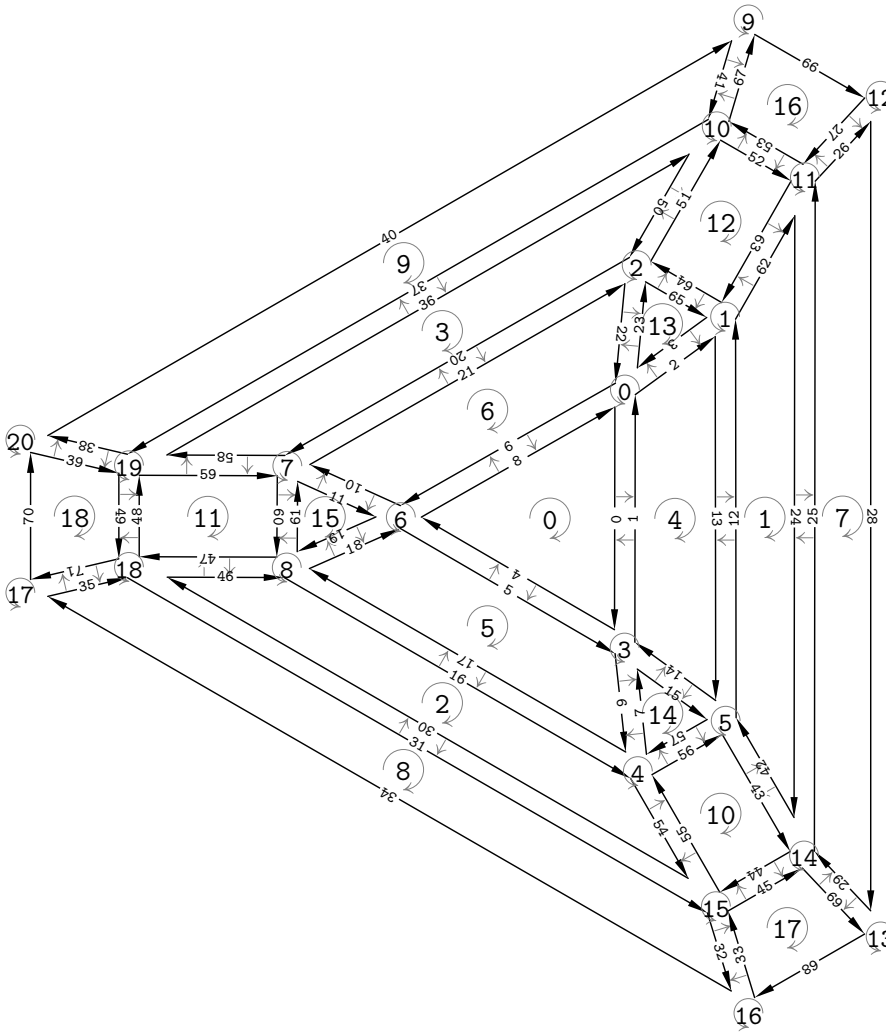
```
62:     }
63:     glGetBufferSubData ( SSB, nv*sizeof(GLint), nfac*sizeof(GLuint), vf );
64:     for ( mfac = cmesh->mfac, i = 0;  i < nfac;  i++ ) {
65:       mfac[i].firsthalfedge = vf[i] & FHEMASK;
66:       mfac[i].degree = vf[i] >> DEGSHIFT;
67:     }
68:     glGetBufferSubData ( SSB, (nv+nfac)*sizeof(GLint),
69:                           nhe*sizeof(GLint), cmesh->mvhei );
70:     glGetBufferSubData ( SSB, (nv+nfac+nhe)*sizeof(GLint),
71:                           nhe*sizeof(GLint), cmesh->mfhei );
72:     glBindBuffer ( SSB, gmesh->MHEBUF );
73:     glGetBufferSubData ( SSB, 0, nhe*4*sizeof(GLint), cmesh->mhe );
74:     glBindBuffer ( SSB, gmesh->VCBUF );
75:     glGetBufferSubData ( SSB, 0, nv*dim*sizeof(GLfloat), cmesh->vc );
76:     free ( vf );
77:     ExitIfGLError ( "GPUmeshToCPU" );
78:     return true;
79:   }
80:   else return false;
81: } /*GPUmeshToCPU*/
```

31.4. Podwajanie i uśrednianie siatki

Operacja zagęszczania siatki (*mesh refinement*) jest złożeniem dwóch bardziej elementarnych operacji, zwanych **podwajaniem** (*doubling*) i **uśrednianiem** (*averaging*): wynik podwajania jest poddawany n-krotnemu uśrednianiu, przy czym najczęściej wykonuje się 2 lub 3 uśredniania. Matematyczne podstawy zagęszczania są opisane w książkach [40] i [41]. Przed zagłębieniem się w opis implementacji zobaczmy, na czym te operacje polegają.

Podwajanie wytwarza siatkę, w której występują ściany odpowiadające wszystkim wierzchołkom, krawędziom i ścianom siatki danej. Ściany odpowiadające ścianom siatki danej są (w zasadzie) ich kopiami. Każdej krawędzi siatki danej odpowiada ściana złożona z czterech krawędzi, przy czym dwie z nich są odcinkami pokrywającymi się z krawędzią daną, a pozostałe dwie są ściągnięte do punktów — końce tych krawędzi są *różnymi* wierzchołkami o *tym samym* położeniu. Wreszcie, każdemu wierzchołkowi siatki danej odpowiada ściana ściągnięta do punktu (mająca wszystkie krawędzie o zerowej długości). Jeśli wierzchołek siatki danej jest wewnętrzny, to odpowiadająca mu ściana ma tyle samo krawędzi co on. Dla wierzchołka brzegowego też mogłoby tak być, ale lepiej jest wygenerować ścianę, która ma o jedną krawędź więcej, bo wierzchołek brzegowy może być końcem tylko dwóch krawędzi, a (zgodnie z przyjętym założeniem) ściana musi być co najmniej trójkątem.

Jeśli w siatce danej dwie ściany mają wspólną krawędź (i w tym sensie sąsiadują), to między kopie tych ścian w siatce wynikowej jest wstawiona ściana odpowiadająca tej krawędzi. Implementacja podwajania, oparta na opisanej wyżej reprezentacji, musi wygenerować odpowiednie półkrawędzie, połączone w pary.

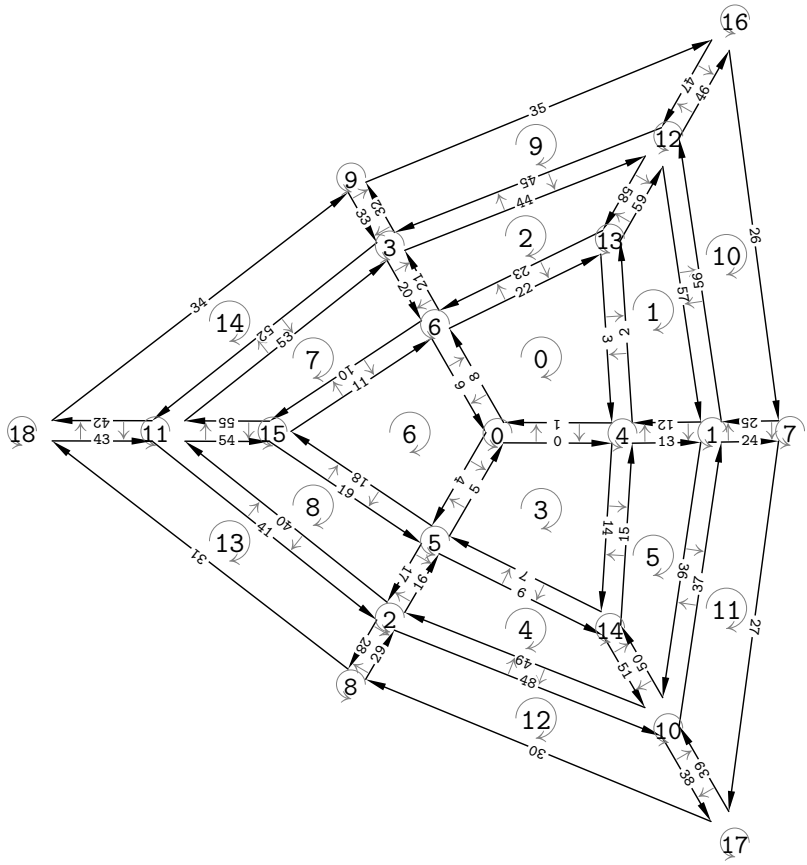


Rysunek 31.2. Wynik podwajania siatki z rysunku 31.1

Na rysunku 31.2 jest pokazany schemat siatki otrzymanej w wyniku podwajania siatki z rysunku 31.1, przy czym pokrywające się wierzchołki zostały odpowiednio porozsuwane, aby uwidocznić ściany 4–12 odpowiadające krawędziom i ściany 13–18 odpowiadające wierzchołkom siatki danej. Ponadto półkrawędzie tworzące pary zostały rozsunięte podobnie jak na rysunku 31.1.

Wynikiem **uśredniania** jest siatka, której ściany odpowiadają wierzchołkom wewnętrznym siatki danej. Każdy wierzchołek siatki wynikowej jest położony w środku ciężkości zbioru wierzchołków pewnej ściany siatki danej. Dwa takie wierzchołki są połączone krawędzią, jeśli odpowiednie ściany siatki danej mają wspólną krawędź, której przynajmniej jeden koniec jest wierzchołkiem wewnętrznym. Jeśli zatem w siatce danej wszystkie krawędzie

są wewnętrzne (czyli powierzchnia zbudowana ze ścian nie ma brzegu), to każdemu wierzchołkowi siatki danej odpowiada ściana, każdej krawędzi siatki danej odpowiada krawędź i każdej ścianie siatki danej odpowiada wierzchołek siatki wynikowej[8].



Rysunek 31.3. Wynik uśredniania siatki z rysunku 31.2

Sytuacja jest bardziej skomplikowana, gdy pewne krawędzie (i wierzchołki) siatki danej są brzegowe. Wtedy nie każdej ścianie takiej siatki musi odpowiadać wierzchołek, co więcej, przyjęta reprezentacja siatki wymusza wytworzenie więcej niż jednego wierzchołka odpowiadającego ścianie siatki danej, jeśli ciąg wierzchołków tej ściany zawiera więcej niż jeden spójny fragment złożony z wierzchołków wewnętrznych — osobny wierzchołek siatki wynikowej odpowiada każdemu takiemu fragmentowi, ponieważ te wierzchołki będą brzegowe i każdy z nich będzie początkiem jednej półkrawędzi brzegowej. Taka sytuacja nie występuje na rysunku 31.3, który przedstawia wynik uśredniania siatki z rysunku 31.2, ale w przypadku „pełnowymiarowych" siatek może się to zdarzyć i implementacja uśredniania musi poprawnie działać także wtedy.

[8]Jeśli graf danej siatki bez brzegu jest planarny, to graf siatki będącej wynikiem uśredniania jest do niego dualny.

Zwróćmy uwagę, że wszystkie wierzchołki wewnętrzne siatki otrzymanej w wyniku podwajania są końcami czterech krawędzi. Wierzchołek wewnętrzny siatki otrzymanej w wyniku uśredniania ma tyle samo krawędzi, co odpowiadająca mu ściana, a ściana tej siatki ma tyle samo krawędzi, co odpowiedni wierzchołek siatki danej. Otrzymane w wyniku podwajania ściany odpowiadające krawędziom są czworokątne, a stąd wynika, że w siatce otrzymanej przez zagęszczanie: podwajanie, po którym nastąpiło n kroków uśredniania, mamy wszystkie ściany czworokątne, gdy n jest nieparzyste, oraz wszystkie wierzchołki wewnętrzne z czterema krawędziami dla parzystego n. Zauważmy też, że podczas iterowania operacji zagęszczania siatki liczby wierzchołków, półkrawędzi i ścian rosną wykładniczo, a zatem liczbę iteracji zagęszczania trzeba wybierać z umiarem. Ciąg siatek otrzymanych przez wielokrotne zagęszczanie szybko zbiega do granicy, która w ogólności jest gładką powierzchnią, a zatem do otrzymania dobrego obrazu zazwyczaj wystarczy niewiele iteracji. Mając siatkę o ścianach czworokątnych, wyświetlimy dwa trójkąty dla każdej ściany, odpowiednio je cieniując. Ale najpierw zagęszczanie trzeba zaimplementować.

31.5. Zmienne szadera zagęszczania siatek

Listing 31.9 przedstawia makrodefinicje i deklaracje zmiennych globalnych szadera obliczeniowego realizującego zagęszczanie siatek, wykorzystywane przez obie operacje: podwajanie i uśrednianie. Szader ten zawiera także zmienne i procedurę z listingu G.8. Każda lokalna grupa robocza składa się z jednego wątku. Makrodefinicje `V0`, `V1`, `FACN` i `OTHE` wprowadzają nazwy pól wektora `ivec4` wykorzystywanego do reprezentowania półkrawędzi — dla zwiększenia czytelności kodu.

Bufory magazynowe przywiązane do punktów dowiązania `1`, `2` i `3` (w celu `GL_SHADER_STORAGE_BUFFER`) zawierają tablice reprezentujące siatkę daną, a bufory przywiązane do punktów `4`, `5` i `6` zawierają tablice, do których ma być wpisany wynik. Wszystkie zmienne jednolite zostały umieszczone w bloku `RefineBlock`. W zmiennej `nsattr` jest podana całkowita liczba skalarnych atrybutów wierzchołka siatki[9]. Wartości zmiennych jednolitych `inv`, `inhe` oraz `infac` są odpowiednio liczbami wierzchołków, półkrawędzi i ścian siatki danej. Podane w liniach `25`–`28` makrodefinicje `imv`, `imfac`, `imvhei` oraz `imfhei` ułatwiają dostęp do czterech tablic umieszczonych w buforze magazynowym `Invmf`, zawierających opisy wierzchołków i ścian oraz tablice z indeksami półkrawędzi dla wierzchołków i ścian. Do wyrażenia opisującego indeks odpowiedniej tablicy dodawane jest wyrażenie opisujące położenie jej początku w buforze. W podobny sposób zorganizowany jest dostęp do czterech tablic w buforze magazynowym `Outmvf`, w którym ma być umieszczona reprezentacja siatki wynikowej. Liczby wierzchołków, półkrawędzi i ścian tej siatki, po ich znalezieniu, zostaną przypisane zmiennym jednolitym `outnv`, `outnhe` i `outnfac`, przy czym robiąc to, aplikacja

[9] Są nimi współrzędne kartezjańskie lub jednorodne położenia wierzchołka, ale mogą być też współrzędne wektora normalnego, koloru lub tekstury. Liczba i interpretacja poszczególnych atrybutów są określone przez aplikację, a ściślej przez szadery rysujące siatkę. Szader zagęszczania siatek nie interpretuje tych atrybutów, wykonując na każdym z nich takie same obliczenia numeryczne.

utworzy także bufory magazynowe o odpowiednich długościach i przywiąże je do podanych na listingu punktów dowiązania.

Listing 31.9. Zmienne jednolite i bloki magazynowe szadera zagęszczania siatek

GLSL

```
#version 450 core

layout(local_size_x=1) in;

#define V0   x
#define V1   y
#define FACN z
#define OTHE w

layout(std430,binding=1) buffer Inmvf  { int    mvf[]; } inmvf;
layout(std430,binding=2) buffer Inmhe  { ivec4  mhe[]; } inmhe;
layout(std430,binding=3) buffer Invc   { float  vc[];  } inmvc;

layout(std430,binding=4) buffer Outmvf { int    mvf[]; } outmvf;
layout(std430,binding=5) buffer Outmhe { ivec4  mhe[]; } outmhe;
layout(std430,binding=6) buffer Outvc  { float  vc[];  } outmvc;

uniform RefineBlock {
    int  stage;
    int  nsattr, inv, inhe, infac, outnv, outnhe, outnfac;
    int  invb, inei, fvf, maxonv, fvhe;
    uint prN0, prN, prStep;
  };

#define imv(I)    inmvf.mvf[I]
#define imfac(I)  inmvf.mvf[inv+(I)]
#define imvhei(I) inmvf.mvf[inv+infac+(I)]
#define imfhei(I) inmvf.mvf[inv+infac+inhe+(I)]
#define imhe(I)   inmhe.mhe[I]
#define imvc(I)   inmvc.vc[I]
#define omv(I)    outmvf.mvf[I]
#define omfac(I)  outmvf.mvf[outnv+(I)]
#define omvhei(I) outmvf.mvf[outnv+outnfac+(I)]
#define omfhei(I) outmvf.mvf[outnv+outnfac+outnhe+(I)]
#define omhe(I)   outmhe.mhe[I]
#define omvc(I)   outmvc.vc[I]
```

W zmiennych jednolitych `invb` oraz `inei` znajdą się liczby wierzchołków brzegowych i *krawędzi* (nie półkrawędzi) wewnętrznych; liczba krawędzi (i półkrawędzi) brzegowych jest równa liczbie wierzchołków brzegowych. Role pozostałych zmiennych jednolitych są wyjaśnione dalej.

31.6. Kompilacja programu zagęszczania i procedury pomocnicze

Listing 31.10 przedstawia procedurę przygotowującą do pracy program szaderów służący do zagęszczania siatek, procedurę likwidującą ten program i dwie procedury pomocnicze używane podczas zagęszczania. Procedura kompilacji odczytuje przesunięcia pól w bloku zmiennych jednolitych `RefineBlock` i przygotowuje bufor, w którym ten blok będzie przechowywany.

Listing 31.10. Procedura kompilacji programu zagęszczania siatek

C

```
static GLuint progid = 0;
static GLuint rbuf, rbbp;
static GLint  uvofs[16];

void LoadMeshRefinementProgram ( void )
{
  const GLchar *filename[] = { "md.comp.glsl" };
  const GLchar *uvnames[] =
    { "RefineBlock", "stage", "nsattr", "inv", "inhe",  "infac", "outnv",
      "outnhe", "outnfac", "invb", "inei", "fvf", "maxonv", "fvhe", "prN0",
      "prN", "prStep" };
  GLuint shader_id;
  GLint  size;

  shader_id = CompileShaderFiles ( GL_COMPUTE_SHADER, 1, &filename[0] );
  progid = LinkShaderProgram ( 1, &shader_id, "meshes refinement 0" );
  glDeleteShader ( shader_id );
  GetAccessToUniformBlock ( progid, 16, uvnames, &size, uvofs, &rbbp );
  glGenBuffers ( 1, &rbuf );
  glBindBufferBase ( GL_UNIFORM_BUFFER, rbbp, rbuf );
  glBufferData ( GL_UNIFORM_BUFFER, size, NULL, GL_DYNAMIC_DRAW );
  ExitIfGLError ( "LoadMeshRefinementProgram" );
} /*LoadMeshRefinementProgram*/

void DeleteMeshRefinementProgram ( void )
{
  glUseProgram ( 0 );
  if ( progid ) { glDeleteProgram ( progid );  progid = 0; }
  glDeleteBuffers ( 1, &rbuf );
  ExitIfGLError ( "DeleteMeshRefinementProgram" );
} /*DeleteMeshRefinementProgram*/
```

Pomocnicza procedura `ExecStage` (listing 31.11) uruchamia program szaderów w celu wykonania kolejnego etapu obliczeń podczas podwajania lub uśredniania. Wartość parametru `stage` tej procedury jest przypisywana zmiennej jednolitej `stage` programu szaderów, z kolei parametr `gsize` określa liczbę wątków potrzebnych w danym etapie. Po wywołaniu

Listing 31.11. Procedury pomocnicze zagęszczania siatek

C

```
#define SSB GL_SHADER_STORAGE_BUFFER

#define SETUVAR(n,type,x) \
  glBufferSubData ( GL_UNIFORM_BUFFER, uvofs[n], sizeof(type), &x );

static void ExecStage ( GLint *uvofs, int stage, int gsize )
{
  SETUVAR ( 0, GLint, stage )
  COMPUTE ( gsize, 1, 1 )
} /*ExecStage*/

static void PrefixSum ( GLint *uvofs, GLuint N0, GLuint N )
{
  GLuint k, m, d;
  GLint  z = 0;

  SETUVAR ( 0, GLint, z )
  SETUVAR ( 13, GLuint, N0 )
  SETUVAR ( 14, GLuint, N )
  d = N/2;
  for ( k = 0, m = N-1;  m > 0;  k++, m >>= 1 ) {
    SETUVAR ( 15, GLuint, k )
    COMPUTE ( d, 1, 1 )
  }
  ExitIfGLError ( "PrefixSum" );
} /*PrefixSum*/

static void SumUp ( GLint *uvofs, GLuint n0, GLuint n )
{
  GLint one = 1;

  SETUVAR ( 0, GLint, one )
  SETUVAR ( 13, GLuint, n0 )
  while ( n > 1 ) {
    SETUVAR ( 14, GLuint, n )
    COMPUTE ( n/2, 1, 1 )
    n = (n+1)/2;
  }
  ExitIfGLError ( "SumUp" );
} /*SumUp*/
```

(za pomocą makrodefinicji `COMPUTE`, zobacz listing 9.1) programu szaderów CPU czeka (w procedurze `glMemoryBarrier`) na dokończenie obliczeń przez wszystkie wątki w globalnej grupie roboczej.

Ponieważ poszczególne wierzchołki i ściany mogą mieć różne liczby półkrawędzi, a reprezentacja siatki jest „spakowana” w tablicach, procedury podwajania i uśredniania, prze-

twarzając równolegle elementy siatki danej, muszą w wielu etapach obliczeń dysponować informacją, w które miejsca tablic należy wpisać odpowiednie wyniki. Aby obliczyć indeksy tych miejsc, w wielu etapach procedury zagęszczania siatek trzeba będzie pododawać na przykład liczby półkrawędzi dla kolejnych wierzchołków lub ścian, czyli obliczyć ciąg sum prefiksowych. Sumy prefiksowe są potrzebne w wielu zastosowaniach, więc opis algorytmu ich obliczania i jego implementacji na GPU umieściłem w dodatku G. Procedura `PrefixSum` pokazana na listingu 31.11 różni się od procedury z listingu G.9 tylko innym sposobem nadawania wartości zmiennym jednolitym (które tu są przechowywane w bloku `RefineBlock`). Ponadto zmienna `stage` otrzymuje wartość `0`, aby opisana w następnym podrozdziale procedura `main` szadera wywołała procedurę `iPrefixSum` pokazaną na listingu G.8. Podobnie, procedura `SumUp` jest procedurą sumowania parami z listingu G.2, dostosowaną do współpracy z procedurami i szaderem podwajania i uśredniania.

31.7. Procedura `main`

Listing 31.12 przedstawia procedurę `main` szadera, który jest działającą na GPU częścią implementacji podwajania i uśredniania. Przed każdym wywołaniem tej procedury program działający na CPU (opisana wcześniej procedura `ExecStage`, `PrefixSum` lub `SumUp`) przypisuje zmiennej `stage` odpowiednią wartość, wskutek czego instrukcja przełącznika wykona instrukcje realizujące bieżący etap obliczeń. Może to być obliczanie sum prefiksowych we wskazanym fragmencie roboczej tablicy `seq.a`, dodanie wszystkich liczb w takim fragmencie lub wywołanie jednej z opisanych dalej procedur szadera. Liczby wątków potrzebnych w kolejnych etapach obliczeń są różne, ponieważ obliczenia te dotyczą działań na wierzchołkach, półkrawędziach lub ścianach siatki danej lub (rzadziej) docelowej. Liczby te oczywiście ustala procedura działająca na CPU.

Niektóre procedury są bardzo krótkie (np. zawierają tylko jedną instrukcję). Takie procedury przerobiłem na makrodefinicje, które zastępują wywołanie procedury jej treścią, dzięki czemu szader działa szybciej, a jego kod źródłowy nie traci czytelności. Na przykład makrodefinicja `AddTwoTerms`, wywoływana w etapie `1` realizowanym przez procedurę `SumUp` z listingu 31.11, dodaje dwa elementy tablicy pomocniczej i zapamiętuje ich sumę na miejscu pierwszego składnika.

Listing 31.12. Procedura `main` szadera zagęszczania siatek

GLSL

```
1: #define AddTwoTerms(I) seq.a[prN0+(I)] += seq.a[prN0+(I)+(prN+1)/2];
2:
3: void main ( void )
4: {
5:   uint i;
6:
7:   i = gl_GlobalInvocationID.x;
```

```
  switch ( stage ) {
case  0: iPrefixSum ( i );        break;
case  1: AddTwoTerms ( i );       break;
case  2: TagVertex ( i );         break;
        /* etapy podwajania */
case  3: DSetECN ( i );           break;
case  4: DSetVCN ( i );           break;
case  5: DCopyVC ( i );           break;
case  6: DSetOVdeg ( i );         break;
case  7: DSetOVfhe ( i );         break;
case  8: DSetWLF ( i );           break;
case  9: DSetEFN1 ( i );          break;
case 10: DSetEFN2 ( i );          break;
case 11: DSetOMfac1 ( i );        break;
case 12: DSetOMfac2 ( i );        break;
case 13: DSetOMfac3 ( i );        break;
case 14: DBindNewhe1 ( i );       break;
case 15: DBindNewhe2 ( i );       break;
case 16: DBindNewhe3 ( i );       break;
case 17: DSetIFDeg ( i );         break;
case 18: DSetOMfhei1 ( i );       break;
case 19: DSetOMfhei2 ( i );       break;
case 20: DSetTgv ( i );           break;
case 21: DSetOMfhei3 ( i );       break;
        /* etapy uśredniania */
case 22: ASetNvi1 ( i );          break;
case 23: ASetNhei1 ( i );         break;
case 24: ASetNfi1 ( i );          break;
case 25: ASetNvi2 ( i, true );    break;
case 26: ASetNfi2 ( i );          break;
case 27: ASetNhei2 ( i );         break;
case 28: ASetNvi2 ( i, false );   break;
case 29: ASetNhei3 ( i );         break;
case 30: ASetNfi3 ( i );          break;
case 31: AClearFVd ( i );         break;
case 32: ASetFVd1 ( i );          break;
case 33: ASetFVd2 ( i );          break;
case 34: AClearFVd ( i );         break;
case 35: ASetOMVert ( i, true );  break;
case 36: ASetOMVert ( i, false ); break;
case 37: ABindHe ( i );           break;
case 38: ASetOMfacHe ( i );       break;
case 39: Average ( i );           break;
default: break;
  }
} /*main*/
```

31.8. Implementacja podwajania

Listing 31.13 przedstawia procedurę działającą na CPU, której zadaniem jest wywoływanie programu szaderów w celu zrealizowania kolejno wszystkich etapów obliczeń dla operacji podwajania siatki. W linii 7 procedura ta wybiera program szaderów z szaderem obliczeniowym zawierającym opisane wcześniej i dalej zmienne i procedury — identyfikator tego programu, przygotowanego zawczasu do pracy, jest pamiętany w zmiennej `progid[0]` (listing 31.10).

Listing 31.13. Procedura podwajania

C

```
1:  char GPUmeshDoubling ( GPUmesh *inmesh, GPUmesh *outmesh )
2:  {
3:    int    inv, inhe, infac, invb, inei, onv, onhe, onfac, fvf, maxonv, fvhe;
4:    GLint  bufsize;
5:    GLuint auxbuf = 0;
6:
7:    glUseProgram ( progid[0] );
8:    glBindBufferBase ( GL_UNIFORM_BUFFER, rbbp, rbuf );
9:    inv = inmesh->nv;  inhe = inmesh->nhe;  infac = inmesh->nfac;
10:   glBindBufferBase ( SSB, 1, inmesh->MVFBUF );
11:   glBindBufferBase ( SSB, 2, inmesh->MHEBUF );
12:   glBindBufferBase ( SSB, 3, inmesh->VCBUF );
13:   SETUVAR ( 1, GLint, inmesh->nsattr )
14:   SETUVAR ( 2, GLint, inv )
15:   SETUVAR ( 3, GLint, inhe )
16:   SETUVAR ( 4, GLint, infac )
17:   maxonv = inhe+2*inv;  SETUVAR ( 11, GLint, maxonv )
18:   bufsize = (3*inv + 4*inhe + infac + 3)*sizeof(GLint);
19:   glGenBuffers ( 1, &auxbuf );
20:   glBindBufferBase ( SSB, 0, auxbuf );
21:   glBufferData ( SSB, bufsize, NULL, GL_DYNAMIC_DRAW );
22:   ExecStage ( uvofs, 2, inv );  /* TagVertex */
23:   SumUp ( uvofs, 0, inv );
24:   glGetBufferSubData ( SSB, 0, sizeof(GLint), &invb );
25:   inei = (inhe-invb)/2;
26:   onv = inhe + 2*invb;  SETUVAR ( 5, GLint, onv )
27:   onhe = 8*(inei + invb);  SETUVAR ( 6, GLint, onhe )
28:   onfac = infac + inei + invb + inv;  SETUVAR ( 7, GLint, onfac )
29:   fvhe = onhe - 2*invb;  SETUVAR ( 12, GLint, fvhe )
30:   if ( !ReallocGPUmesh ( outmesh, onv, onhe, onfac, inmesh->nsattr,
31:                          inmesh->pdim, inmesh->pofs, inmesh->nvofs ) )
32:     goto failure;
33:   glBindBufferBase ( SSB, 4, outmesh->MVFBUF );
34:   glBindBufferBase ( SSB, 5, outmesh->MHEBUF );
35:   glBindBufferBase ( SSB, 6, outmesh->VCBUF );
36:   glBindBufferBase ( SSB, 0, auxbuf );
```

```
37:   ExecStage ( uvofs, 3, inhe );   /* DSetECN */
38:   PrefixSum ( uvofs, maxonv, inhe+1 );
39:   ExecStage ( uvofs, 4, inv );    /* DSetVCN */
40:   PrefixSum ( uvofs, maxonv+inhe+1, inv+1 );
41:   ExecStage ( uvofs, 5, inv );    /* DCopyVC */
42:   ExecStage ( uvofs, 6, onv-1 );  /* DSetOVdeg */
43:   PrefixSum ( uvofs, 0, onv );
44:   ExecStage ( uvofs, 7, onv );    /* DSetOVfhe */
45:   ExecStage ( uvofs, 8, infac );  /* DSetWLF */
46:   ExecStage ( uvofs, 9, inhe );   /* DSetEFN1 */
47:   PrefixSum ( uvofs, maxonv+inhe+inv+2, inhe );
48:   ExecStage ( uvofs, 10, inhe );  /* DSetEFN2 */
49:   glBindBuffer ( GL_COPY_READ_BUFFER, inmesh->MVFBUF );
50:   glBindBuffer ( GL_COPY_WRITE_BUFFER, outmesh->MVFBUF );
51:   glCopyBufferSubData ( GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER,
52:                 inv*sizeof(GLint), onv*sizeof(GLint), infac*sizeof(GLint) );
53:   SETUVAR ( 8, GLint, invb )
54:   SETUVAR ( 9, GLint, inei )
55:   fvf = infac+inei+invb;  SETUVAR ( 10, GLint, fvf )
56:   ExecStage ( uvofs, 11, inei+invb );  /* DSetOMfac1  */
57:   ExecStage ( uvofs, 12, inv );    /* DSetOMfac2 */
58:   PrefixSum ( uvofs, 0, fvf-infac+1 );
59:   ExecStage ( uvofs, 13, inv );    /* DSetOMfac3 */
60:   ExecStage ( uvofs, 14, inhe );   /* DBindNewhe1 */
61:   PrefixSum ( uvofs, 0, inhe );
62:   ExecStage ( uvofs, 15, inhe );   /* DBindNewhe2 */
63:   ExecStage ( uvofs, 16, inhe );   /* DBindNewhe3 */
64:   ExecStage ( uvofs, 17, infac );  /* DSetIFDeg */
65:   PrefixSum ( uvofs, maxonv+3*inhe+inv+3, infac );
66:   ExecStage ( uvofs, 18, infac );  /* DSetOMfhei1 */
67:   ExecStage ( uvofs, 19, inhe );   /* DSetOMfhei2 */
68:   ExecStage ( uvofs, 20, inv );    /* DSetTgv */
69:   PrefixSum ( uvofs, 0/*maxonv+4*inhe+inv+infac+3*/, inv );
70:   ExecStage ( uvofs, 21, inv );    /* DSetOMfhei3 */
71:   glDeleteBuffers ( 1, &auxbuf );
72:   glUseProgram ( 0 );
73:   ExitIfGLError ( "GPUmeshDoubling" );
74:   return true;
75:
76: failure:
77:   glDeleteBuffers ( 1, &auxbuf );
78:   glUseProgram ( 0 );
79:   return false;
80: } /*GPUmeshDoubling*/
```

W liniach 10–16 utworzone za pomocą opisanej wcześniej procedury `CPUmeshToGPU` bufory, w których znajduje się reprezentacja siatki danej, są przywiązywane do odpowied-

nich punktów dowiązania w celu GL_SHADER_STORAGE_BUFFER, a zmiennym jednolitym nsattr, inv, inhe, infac w bloku RefineBlock są nadawane wartości opisujące liczbę atrybutów wierzchołka i liczby elementów siatki. W linii 17 jest obliczana i przypisywana zmiennej jednolitej maxonv maksymalna możliwa liczba wierzchołków siatki wynikowej. W liniach 18–21 jest tworzony odpowiednio pojemny bufor zawierający tablicę roboczą dla szadera. Bufor ten jest przywiązywany do punktu dowiązania 0; w szczególności w nim będą obliczane sumy prefiksowe różnych ciągów liczb.

Opis kolejnych etapów podwajania jest podany dalej, obok listingów przedstawiających poszczególne procedury wywoływane przez procedurę main pokazaną na listingu 31.12.

Listing 31.14 przedstawia makrodefinicje używane w treści procedur szadera realizujących etapy podwajania. Używają one sześciu tablic liczb typu int, upakowanych w buforze magazynowym o nazwie seq (zobacz listing G.8). Długości tych tablic są określone przez liczby wierzchołków (n_v), półkrawędzi (n_h) i ścian (n_f) siatki danej i makrodefinicja dająca dostęp do elementów każdej tablicy dodaje do jej indeksu sumę długości tablic ją poprzedzających w buforze. Zmiennej jednolitej maxonv procedura podwajania przypisuje wartość $n_h + 2n_v$, która jest górnym oszacowaniem liczby wierzchołków siatki wynikowej.

Makrodefinicja PREVIFAC_HEDGE służy do znalezienia identyfikatora półkrawędzi poprzedzającej półkrawędź na pozycji en w zamkniętym w cykl ciągu identyfikatorów półkrawędzi ściany fn.

Listing 31.14. Makrodefinicje dla podwajania

GLSL

```
1: #define ecn(I)    seq.a[maxonv+(I)]
2: #define vcn(I)    seq.a[maxonv+inhe+1+(I)]
3: #define efn(I)    seq.a[maxonv+inhe+inv+2+(I)]
4: #define wlf(I)    seq.a[maxonv+2*inhe+inv+3+(I)]
5: #define fcn(I)    seq.a[maxonv+3*inhe+inv+3+(I)]
6:
7: #define PREVIFAC_HEDGE(fn,en) \
8:   ((en) > 0 ? \
9:     imfhei((imfac(fn) & FHEMASK) + (en) - 1) : \
10:    imfhei((imfac(fn) & FHEMASK) + (imfac(fn) >> DEGSHIFT) - 1))
```

Listing 31.15 przedstawia procedurę realizującą etap 2 podwajania (listing 31.13, linia 22): znakowanie i liczenie wierzchołków brzegowych siatki danej. Procedura bada, czy ostatnia półkrawędź wychodząca z i-tego wierzchołka jest brzegowa (co oznacza, że wierzchołek jest brzegowy) i ustawia albo kasuje bit na pozycji 25 (zobacz listing 31.3). Jednocześnie dla wierzchołka brzegowego wpisuje do tablicy seq.a liczbę 1, a dla wewnętrznego 0. Następnie (listing 31.13, linia 23) obliczana jest suma wpisanych liczb, tj. suma wpisanych jedynek, która jest liczbą wierzchołków brzegowych siatki. W linii 24 liczba ta jest odczytywana przez procedurę podwajania, po czym znajdowane są liczby krawędzi wewnętrznych siatki danej (inei) oraz wierzchołków (onv), półkrawędzi (onhe) i ścian (onfac) siatki wynikowej; liczby te są natychmiast przypisywane odpowiednim zmiennym jednolitym. W linii 29 zmiennej jednolitej fvhe zostaje przypisana wartość, od której zaczynają się identyfikatory półkrawędzi ścian siatki wynikowej odpowiadających wierzchołkom siatki danej.

Listing 31.15. Znakowanie wierzchołków brzegowych

GLSL

```
1: void TagVertex ( uint i )  /* etap 2 podwajania i uśredniania */
2: {
3:   int fhe, deg;
4:
5:   fhe = imv(i) & FHEMASK;
6:   deg = imv(i) >> DEGSHIFT;
7:   if ( imhe(imvhei(fhe+deg-1)).OTHE < 0 ) {
8:     imv(i) |= TAGMASK;
9:     seq.a[i] = 1;
10:  }
11:  else {
12:    imv(i) &= ~TAGMASK;
13:    seq.a[i] = 0;
14:  }
15: } /*TagVertex*/
```

W linii 30 następuje rezerwacja pamięci na reprezentację siatki wynikowej w pamięci GPU, po czym odpowiednie bufory są przywiązywane do punktów dowiązania 4, 5, 6 w celu GL_SHADER_STORAGE_BUFFER. Odtąd można używać makrodefinicji podanych w liniach 1–5 na listingu 31.14 i w liniach 31–36 na listingu 31.9.

Listing 31.16. Ustalanie liczb półkrawędzi i kopii wierzchołków

GLSL

```
1: void DSetECN ( uint i )  /* etap 3 */
2: {
3:   ecn(i+1) = imhe(i).OTHE < 0 ? 6 : 4;
4:   if ( i == 0 )
5:     ecn(0) = 0;
6:  /*DSetECN*/
7:
8: void DSetVCN ( uint i )  /* etap 4 */
9: {
10:  int deg;
11:
12:  deg = imv(i) >> DEGSHIFT;
13:  vcn(i+1) = (imv(i) & TAGMASK) != 0 ? deg + 2 : deg;
14:  if ( i == 0 )
15:    vcn(0) = 0;
16: } /*DSetVCN*/
```

Etap 3 podwajania (listing 31.13, linie 37–38 i listing 31.16, linie 1–6) ma na celu ustalenie dla każdej krawędzi siatki danej identyfikatorów półkrawędzi w siatce wynikowej odpowiadających tej krawędzi. Jeśli krawędź jest brzegowa, to będzie dla niej wygenerowane 6 półkrawędzi, a jeśli wewnętrzna, to 8, ale to oznacza potrzebę wygenerowania 4 półkrawędzi dla

każdej z tworzących parę półkrawędzi reprezentujących krawędź wewnętrzną. Dlatego procedura DSetECN wpisuje do tablicy ecn liczbę 6 albo 4. Pierwszy element tablicy[10] otrzymuje wartość 0, a liczba 6 lub 4 dla i-tej półkrawędzi jest wpisywana w miejsce $i+1$. Po wpisaniu tych liczb następuje obliczenie sum prefiksowych w tablicy ecn.

Podobne obliczenie jest wykonywane w etapie 4 (listing 31.13, linie 39–40 i listing 31.16, linie 8–16) dla wierzchołków siatki danej. Do tablicy vcn w miejscu $i+1$ procedura DSetVCN wpisuje stopień (liczbę półkrawędzi wychodzących z) i-tego wierzchołka, jeśli jest on wewnętrzny, lub liczbę o 2 większą, jeśli jest brzegowy, a następnie są obliczane sumy prefiksowe. W ten sposób dla każdego wierzchołka siatki danej są określane numery wierzchołków ściany siatki wynikowej odpowiadającej temu wierzchołkowi.

Listing 31.17. Tworzenie kopii wierzchołka

GLSL

```
1: void DCopyVC ( uint i )  /* etap 5 */
2: {
3:   int deg, p, j, k;
4:
5:   deg = imv(i) >> DEGSHIFT;
6:   if ( (imv(i) & TAGMASK) != 0 )
7:     deg += 2;
8:   p = vcn(i);
9:   for ( j = 0; j < deg; j++ ) {
10:    for ( k = 0; k < nsattr; k++ )
11:      omvc((p+j)*nsattr+k) = imvc(i*nsattr+k);
12:    omv(p+j) = 4 << DEGSHIFT;
13:  }
14:  if ( (imv(i) & TAGMASK) != 0 )
15:    omv(p) = omv(p+deg-1) = 2 << DEGSHIFT;
16: } /*DCopyVC*/
```

Listing 31.17 przedstawia procedurę wykonywaną w etapie 5 podwajania (listing 31.13, linia 41). Ta procedura wykonuje d (dla wierzchołka wewnętrznego) albo $d+2$ (dla wierzchołka brzegowego będącego początkiem d półkrawędzi) kopii wektora współrzędnych (położenia i innych atrybutów), a ponadto określa stopnie wierzchołków siatki wynikowej. Wierzchołki wewnętrzne tej siatki mają stopień 4, a wierzchołki brzegowe (pierwsza i ostatnia kopia wierzchołka brzegowego siatki danej) są początkami dwóch półkrawędzi.

[10] W moich wczesnych wersjach procedury GPUmeshDoubling wartość 0 była przypisywana pierwszemu elementowi tablicy przez CPU za pomocą procedury glBufferSubData. Na moim komputerze stacjonarnym to działało, a po przeniesieniu na laptopa też działało, ale dawało błędne wyniki (nie udało mi się odkryć natury tego błędu, być może chodziło o synchronizację). Dlatego w treści szadera do procedury DSetECN i opisanych dalej procedur DSetVCN, DSetOVdeg i DSetOMfac2 dopisałem instrukcje przypisujące odpowiednią wartość pierwszemu elementowi tablicy, wykonywane, gdy i == 0. Takie rozwiązanie jest bardziej eleganckie, mniej podatne na błędy podczas pielęgnacji programu i chyba korzystne dla szybkości obliczeń, choć to jest trudne do zmierzenia.

Listing 31.18. Tworzenie reprezentacji wierzchołków wyjściowych

GLSL

```
void DSetOVdeg ( uint i )  /* etap 6 */
{
  seq.a[i+1] = omv(i) >> DEGSHIFT;
  if ( i == 0 )
    seq.a[i] = 0;
} /*DSetOVdeg*/

#define DSetOVfhe(i) omv(i) |= seq.a[i];  /* etap 7 */
```

Procedura i makro na listingu 31.18 realizują etapy 6 i 7 podwajania. W etapie 6 do tablicy `seq.a` na pozycji $i+1$ jest wpisywany stopień i-tego wierzchołka wyjściowego, a element `seq.a[0]` otrzymuje wartość 0. Następnie obliczane są sumy prefiksowe ciągu w tablicy, co daje dla każdego wierzchołka wyjściowego numer f pozycji w tablicy `mvhei`, od której zaczyna się ciąg półkrawędzi tego wierzchołka. Wykonywana w etapie 7 instrukcja (zawarta w makrodefinicji `DSetOVfhe`) zapisuje w 25 najmniej znaczących bitach opisu i-tego wierzchołka wyjściowego ten numer — użyty operator przypisania to „|=", ponieważ 6 najbardziej znaczących bitów w tym momencie już przechowuje stopień wierzchołka wyjściowego (a pozostałe bity, póki co, mają wartość 0).

Listing 31.19. Wypełnianie tablicy `wlf`

GLSL

```
void DSetWLF ( uint i )  /* etap 8 */
{
  int deg, fhe, k;

  deg = imfac(i) >> DEGSHIFT;
  fhe = imfac(i) & FHEMASK;
  for ( k = 0; k < deg; k++ )
    wlf(imfhei(fhe+k)) = k;
} /*DSetWLF*/
```

Procedura `DSetWLF` na listingu 31.19, wykonywana dla każdej ściany siatki danej, wypełnia tablicę pomocniczą o długości n_h, nazwaną `wlf`. Parametr `i` jest numerem ściany. Do tablicy dla każdej półkrawędzi należącej do tej ściany jest wpisywany numer pozycji tej półkrawędzi w ciągu półkrawędzi ściany, od 0 do $d-1$, gdzie d jest stopniem ściany.

Listing 31.20 przedstawia procedury realizujące etapy 9 i 10 podwajania; procedury te są wykonywane dla każdej półkrawędzi siatki danej. W etapie 9 do pomocniczej tablicy `efn` (przechowywanej w buforze `seq`) jest wpisywany ciąg liczb całkowitych; pierwsza z nich jest równa n_f (jest to liczba ścian siatki danej), a każda następna jest jedynką albo zerem. Jedynka jest wpisywana na pozycji i-tej, jeśli półkrawędź o numerze $i-1$ nie ma pary (czyli reprezentuje krawędź brzegową) albo jeśli półkrawędź tworząca z nią parę ma większy numer. W ten sposób liczba jedynek wpisanych do tablicy `efn` jest równa liczbie *krawędzi* (brzegowych i wewnętrznych). Po wpisaniu tych zer i jedynek w tablicy `efn` są obliczane sumy

prefiksowe, a następnie wykonywana jest procedura DSetEFN2, która (dla tych półkrawędzi, którym odpowiada 0 wpisane przez procedurę DSetEFN1) przypisuje liczbę w tablicy efn odpowiadającą drugiej półkrawędzi z pary. Ściany siatki wynikowej odpowiadające krawędziom siatki danej będą miały numery od n_f (liczby ścian siatki danej) do $n_f + n_e - 1$ (gdzie n_e jest liczbą krawędzi siatki danej). W ten sposób dla każdej półkrawędzi siatki danej na odpowiednim miejscu tablicy efn jest podany numer ściany siatki wynikowej odpowiadający krawędzi reprezentowanej przez tę półkrawędź (zarówno wtedy, gdy półkrawędź ma parę, jak i wtedy, gdy jej nie ma).

Listing 31.20. Liczenie krawędzi siatki danej

GLSL

```
1: void DSetEFN1 ( uint i )  /* etap 9 */
2: {
3:   int j;
4:
5:   if ( i == 0 )
6:     efn(i) = infac;
7:   else {
8:     j = imhe(i-1).OTHE;
9:     efn(i) = j < 0 || j >= i ? 1 : 0;
10:  }
11: } /*DSetEFN1*/
12:
13: void DSetEFN2 ( uint i )  /* etap 10 */
14: {
15:   int j;
16:
17:   j = imhe(i).OTHE;
18:   if ( j >= 0 && j < i )
19:     efn(i) = efn(j);
20: } /*DSetEFN2*/
```

Instrukcje w liniach 49–52 na listingu 31.13 kopiują n_f liczb z tablicy imfac do tablicy omfac. Ma to na celu utworzenie reprezentacji ścian siatki wynikowej odpowiadających ścianom siatki danej — wszystkie te ściany mają identyczne stopnie i będą miały ciągi numerów półkrawędzi zaczynające się w tablicy omfhei od tych samych miejsc co ciągi numerów półkrawędzi w tablicy imfhei[11].

Procedury na listingu 31.21 tworzą opisy ścian odpowiadających krawędziom i wierzchołkom siatki danej; ściany odpowiadające krawędziom mają po 4 półkrawędzie (a zatem, w linii 3 w opisie ściany przechowywanym w tablicy omfac jest zapisywany stopień 4). Procedura DSetOMfac2 zapisuje tylko stopień ściany odpowiadającej wierzchołkowi — równy d dla wierzchołka wewnętrznego stopnia d oraz $d + 2$ dla wierzchołka brzegowego. Stopnie te są też zapisywane w tablicy seq.a, po czym następuje (listing 31.13, linia 58) obliczanie sum

[11] Oczywiście, numery półkrawędzi tych ścian w siatce wynikowej będą inne.

prefiksowych. Otrzymane w ten sposób indeksy początków list półkrawędzi dla ścian odpowiadających wierzchołkom są w etapie 13 (przez makro DSetOMfac3, listing 31.21, linia 20) zapisywane w tablicy omfac.

Listing 31.21. Tworzenie ścian odpowiadających krawędziom

GLSL

```
1: void DSetOMfac1 ( uint i )  /* etap 11 */
2: {
3:   omfac(infac+i) = (4 << DEGSHIFT) +
4:       (imfac(infac-1) & FHEMASK) + (imfac(infac-1) >> DEGSHIFT) + 4*int(i);
5: } /*DSetOMfac1*/
6:
7: void DSetOMfac2 ( uint i )  /* etap 12 */
8: {
9:   int deg;
10:
11:   deg = imv(i) >> DEGSHIFT;
12:   if ( (imv(i) & TAGMASK ) != 0 )
13:     deg += 2;
14:   omfac(fvf+i) = deg << DEGSHIFT;
15:   seq.a[i+1] = deg;
16:   if ( i == 0 )
17:     seq.a[i] = 4;
18: } /*DSetOMfac2*/
19:
20: #define DSetOMfac3(i) omfac(fvf+i) += (omfac(fvf-1) & FHEMASK) + seq.a[i];
```

Listing 31.22 przedstawia procedury wykonywane w etapach 14–16 podwajania; mają one na celu połączenie w pary półkrawędzi siatki wynikowej, tzn. przypisanie każdej półkrawędzi numeru jej drugiej połowy (albo numeru -1 dla półkrawędzi brzegowych) oraz numeru jej ściany.

Procedura DBindNewhe1 (wywoływana przez instrukcję w linii 60, listing 31.13) do tablicy seq.a wpisuje ciąg zer i jedynek o długości n_h; i-ty element tego ciągu odpowiada i-tej półkrawędzi siatki wejściowej i jest jedynką, jeśli druga półkrawędź z pary nie istnieje lub ma numer większy niż i, a zerem w przeciwnym razie. Następnie dla tego ciągu obliczane są sumy prefiksowe.

Listing 31.22. Łączenie półkrawędzi w pary

GLSL

```
1: void DBindNewhe1 ( uint i )  /* etap 14 */
2: {
3:   int j;
4:
5:   j = imhe(i).OTHE;
6:   seq.a[i] = j < 0 || j > i ? 1 : 0;
7: } /*DBindNewhe1*/
```

```
 8:
 9: void DBindNewhe2 ( uint i )  /* etap 15 */
10: {
11:   int j, ecni;
12:
13:   j = imhe(i).OTHE;
14:   ecni = ecn(i);
15:   omhe(ecni).OTHE = ecni+1;
16:   omhe(ecni+1).OTHE = ecni;
17:   omhe(ecni+2).OTHE = ecni+3;
18:   omhe(ecni+3).OTHE = ecni+2;
19:   omhe(ecni).FACN = imhe(i).FACN;
20:   omhe(ecni+3).FACN = fvf + imhe(i).V0;
21:   if ( j < 0 ) {
22:     omhe(ecni+4).OTHE = omhe(ecni+5).OTHE = -1;
23:     omhe(ecni+1).FACN = omhe(ecni+2).FACN =
24:         omhe(ecni+4).FACN = omhe(ecni+5).FACN = infac+seq.a[i]-1;
25:   }
26:   else if ( i < j )
27:     omhe(ecni+1).FACN = omhe(ecni+2).FACN = infac+seq.a[i]-1;
28: } /*DBindNewhe2*/
29:
30: void DBindNewhe3 ( uint i )  /* etap 16 */
31: {
32:   int j, ecni;
33:
34:   j = imhe(i).OTHE;
35:   if ( j >= 0 && j < i ) {
36:     ecni = ecn(i);
37:     omhe(ecni+1).FACN = omhe(ecni+2).FACN = omhe(omhe(ecn(j)).OTHE).FACN;
38:   }
39: } /*DBindNewhe3*/
```

Procedura DBindNewhe2 (linia 62 na listingu 31.13) w linii 14 odczytuje z tablicy ecn numer pierwszej półkrawędzi siatki wynikowej odpowiadającej i-tej półkrawędzi siatki danej, oznaczmy go literą k. Jeśli i-ta półkrawędź siatki danej jest brzegowa, to zostanie dla niej wygenerowanych 6 półkrawędzi siatki wynikowej, o numerach $k, \ldots, k+5$. W przeciwnym razie powstaną 4 półkrawędzie o numerach $k, \ldots, k+3$ i w ten sposób *krawędzi* wewnętrznej siatki danej odpowiada obiecane 8 półkrawędzi siatki wynikowej.

Sposób określania atrybutów półkrawędzi siatki wynikowej możemy prześledzić na przykładzie z rysunków 31.1 i 31.2. Półkrawędzi 0 siatki danej, która razem z półkrawędzią 3 reprezentuje krawędź wewnętrzną, odpowiadają półkrawędzie $k = 0, 1, 2$ i 3 siatki wynikowej. Półkrawędzie siatki wynikowej o numerach k i $k+1$ oraz $k+2$ i $k+3$ tworzą pary, przy czym półkrawędź o numerze k należy do kopii ściany siatki danej (przypomnijmy, że pierwsze e_f ścian siatki wynikowej to kopie ścian siatki danej), a zatem w linii 19 w reprezentacji tej półkrawędzi zostaje zapamiętany numer ściany i-tej półkrawędzi z siatki danej. Półkrawędź

$k+3$ należy do ściany odpowiadającej wierzchołkowi siatki danej; numer tego wierzchołka jest obliczany w linii 20 i jest to suma numeru wierzchołka siatki danej oraz liczby ścian i krawędzi tej siatki (bo ściany odpowiadające wierzchołkom dostają kolejne numery po ścianach odpowiadających ścianom i krawędziom).

Jeśli i-ta półkrawędź ma parę, której numer jest większy, to półkrawędzie $k+1$ i $k+2$ siatki wynikowej należą do ściany odpowiadającej krawędzi wewnętrznej reprezentowanej przez i-tą półkrawędź. Numer tej ściany jest obliczany w linii 27, na podstawie ciągu sum prefiksowych w tablicy `seq.a`. Jeśli i-ta półkrawędź ma parę, której numer jest mniejszy, to jej numer ściany zostanie przypisany później.

W przykładzie z rysunków 31.1 i 31.2 półkrawędzi brzegowej 6 odpowiadają w siatce wynikowej półkrawędzie $24 = k$, 25, ..., 29. Półkrawędzie $k+1, \ldots, k+4$ otaczają ścianę (numer 7 w rozważanym przykładzie) odpowiadającą krawędzi brzegowej siatki danej. Półkrawędź $k+4$ jest brzegowa, zatem otrzymuje numer swojej pary -1. Półkrawędź $k+5$ nie jest brzegowa, ale tymczasem dostaje numer pary -1, a właściwy numer półkrawędzi do pary będzie ustalony później.

Etapy realizowane przez procedury na listingu 31.23 mają na celu znalezienie, dla ścian siatki wynikowej odpowiadających ścianom i krawędziom siatki danej, ciągów numerów półkrawędzi i wpisanie ich do tablicy `omfhei`. Procedura (makrodefinicja) `DSetIFDeg`, wykonywana w etapie 17, wpisuje na i-tym miejscu tablicy `fcn` stopień ściany o numerze $i-1$ siatki danej (przy czym `fcn[0]` otrzymuje wartość 0), po czym następuje obliczenie sum prefiksowych ciągu liczb wpisanych do tej tablicy. W ten sposób są obliczane, dla ścian będących kopiami ścian siatki, numery miejsc w tablicy `omfhei`, od których zaczynają się listy numerów półkrawędzi tych ścian. Wywoływana w etapie 18 procedura `DSetOMfhei1` wpisuje numery tych półkrawędzi, korzystając z odwzorowania numerów półkrawędzi ścian siatki danej na numery półkrawędzi ich kopii w siatce wynikowej w tablicy `ecn` (odwzorowanie to jest reprezentowane przez sumy prefiksowe ciągu otrzymanego w etapie 2).

Listing 31.23. Ustalanie list półkrawędzi ścian

GLSL

```
#define DSetIFDeg(i) fcn(i) = i == 0 ? 0 : imfac(i-1) >> DEGSHIFT;

void DSetOMfhei1 ( uint i )  /* etap 18 */
{
  int deg, fhe, imfh, j;

  deg = imfac(i) >> DEGSHIFT;
  fhe = fcn(i);
  imfh = imfac(i) & FHEMASK;
  for ( j = 0; j < deg; j++ )
    omfhei(imfh+j) = ecn(imfhei(imfh+j));
} /*DSetOMfhei1*/

void DSetOMfhei2 ( uint i )  /* etap 19 */
{
```

```
16:   int k, ecni;
17:
18:   k = omfac(efn(i)) & FHEMASK;
19:   ecni = ecn(i);
20:   if ( imhe(i).OTHE < 0 ) {
21:     omfhei(k)   = ecni + 1;
22:     omfhei(k+1) = ecni + 2;
23:     omfhei(k+2) = ecni + 4;
24:     omfhei(k+3) = ecni + 5;
25:   }
26:   else if ( imhe(i).OTHE > i ) {
27:     omfhei(k)   = ecni + 1;
28:     omfhei(k+1) = ecni + 2;
29:   }
30:   else {
31:     omfhei(k+2) = ecni + 1;
32:     omfhei(k+3) = ecni + 2;
33:   }
34: } /*DSetOMfhei2*/
```

Procedura DSetOMfhei2, wywoływana w etapie 19, dla i-tej półkrawędzi siatki danej tworzy listę półkrawędzi ścian odpowiadających krawędziom siatki danej. Każda taka ściana ma cztery półkrawędzie. Jeśli i-ta półkrawędź nie ma pary, to w liniach 21–24 do tablicy omfhei następują przypisania wszystkich czterech numerów półkrawędzi. Jeśli zaś półkrawędź ma parę (razem z którą reprezentuje krawędź wewnętrzną), to gdy numer pary jest większy niż i, do tablicy zostają wpisane odpowiednie numery na pierwsze dwa miejsca (linie 27, 28), a jeśli mniejszy, to na ostatnie dwa miejsca na liście (linie 31, 32).

Ostatnie dwa etapy podwajania realizuje makro i procedura na listingu 31.24; ich zadaniem jest ustalenie list półkrawędzi dla ścian siatki wynikowej odpowiadających wierzchołkom siatki danej i uzupełnienie w tablicach wszystkich brakujących informacji.

Makro DSetTgv, zawierające instrukcję wykonywaną w etapie 20, wpisuje do tablicy seq.a na pozycji i liczbę 1, jeśli wierzchołek $i-1$ jest brzegowy, albo 0, jeśli wewnętrzny. Następnie obliczone zostają sumy prefiksowe w tej tablicy. Są one potrzebne do ustalenia numerów półkrawędzi w otoczeniu ścian odpowiadających wierzchołkom brzegowym, które należy połączyć w pary (co nie zostało zrobione w etapie 16).

Listing 31.24. Ustalanie list półkrawędzi ścian odpowiadających wierzchołkom

GLSL

```
1: #define DSetTgv(i) seq.a[i] = i == 0 ? \
2:           0 : ((imv(i-1) & TAGMASK) != 0 ? 1 : 0);  /* etap 20 */
3:
4: void DSetOMfhei3 ( uint i )  /* etap 21 */
5: {
6:   int d, j, v0, v1, l, f, ecnl, p, q, k;
7:
```

```
8:  d = imv(i) >> DEGSHIFT;
9:  j = imv(i) & FHEMASK;
10: if ( (imv(i) & TAGMASK) != 0 ) {
11:   v0 = vcn(i);
12:   l = imvhei(j);
13:   f = imhe(l).FACN;
14:   l = PREVIFAC_HEDGE ( f, wlf(l) );
15:   ecnl = ecn(l);
16:   omhe(ecnl+4).V1 = v0;
17:   q = fvhe + 2*seq.a[i];
18:   omhe(ecnl+5).OTHE = p = q+1;
19:   omhe(p).OTHE = ecnl+5;
20:   omhe(ecnl+5).V0 = omhe(p).V1 = v0;
21:   omhe(ecnl+5).V1 = omhe(p).V0 = v0+1;
22:   omvhei(omv(v0) & FHEMASK) = ecnl+5;
23:   omhe(q).OTHE = -1;
24:   omhe(q).V0 = v0;
25:   omhe(q).V1 = v0+d+1;
26:   omvhei((omv(v0) & FHEMASK)+1) = q;
27:   omhe(q).FACN = omhe(p).FACN = fvf+int(i);
28:   omfhei(omfac(fvf+i) & FHEMASK) = q;
29:   for ( k = 0; k < d; k++ ) {
30:     v0 = vcn(i)+k+1;
31:     l = imvhei((imv(i) & FHEMASK) + k);
32:     f = imhe(l).FACN;
33:     ecnl = ecn(l);
34:     omvhei(omv(v0) & FHEMASK) = p;
35:     omfhei((omfac(fvf+i) & FHEMASK)+d+1-k) = p;
36:     omhe(ecnl).V0 = omhe(ecnl+1).V1 = v0;
37:     omhe(omhe(ecnl).OTHE).V1 = v0;
38:     omhe(ecnl+2).V0 = omhe(ecnl+3).V1 = v0;
39:     omhe(ecnl+2).V1 = omhe(ecnl+3).V0 = v0+1;
40:     omvhei((omv(v0) & FHEMASK)+2) = ecnl;
41:     omvhei((omv(v0) & FHEMASK)+3) = ecnl+2;
42:     p = ecnl+3;
43:     l = PREVIFAC_HEDGE ( f, wlf(l) );
44:     ecnl = ecn(l);
45:     omhe(ecnl).V1 = v0;
46:     omhe(omhe(ecnl).OTHE).V0 = v0;
47:     omvhei((omv(v0) & FHEMASK)+1) = ecnl+1;
48:   }
49:   omfhei((omfac(fvf+i) & FHEMASK)+1) = p;
50:   l = imvhei((imv(i) & FHEMASK)+d-1);
51:   ecnl = ecn(l);
52:   omhe(ecnl+4).V0 = vcn(i)+d+1;
53:   omvhei(omv(vcn(i)+d+1) & FHEMASK) = ecnl+3;
54:   omvhei((omv(vcn(i)+d+1) & FHEMASK)+1) = ecnl+4;
```

```
55:   }
56:   else {
57:     for ( k = 0; k < d; k++ ) {
58:       v0 = vcn(i)+k;
59:       v1 = ( k < d-1) ? v0+1 : vcn(i);
60:       l = imvhei((imv(i) & FHEMASK)+k);
61:       f = imhe(l).FACN;
62:       ecnl = ecn(l);
63:       omhe(ecnl).V0 = v0;
64:       omhe(ecnl+2).V0 = omhe(ecnl+3).V1 = v0;
65:       omhe(ecnl+2).V1 = omhe(ecnl+3).V0 = v1;
66:       omhe(omhe(ecnl).OTHE).V1 = v0;
67:       omvhei(omv(v0) & FHEMASK) = ecnl;
68:       omvhei((omv(v0) & FHEMASK)+1) = ecnl+2;
69:       omvhei((omv(v1) & FHEMASK)+2) = ecnl+3;
70:       omfhei((omfac(fvf+i) & FHEMASK)+d-1-k) = ecnl+3;
71:       l = PREVIFAC_HEDGE ( f, wlf(l) );
72:       ecnl = ecn(l);
73:       omhe(ecnl).V1 = v0;
74:       omhe(omhe(ecnl).OTHE).V0 = v0;
75:       omvhei((omv(v0) & FHEMASK)+3) = omhe(ecnl).OTHE;
76:     }
77:   }
78: } /*DSetOMfhei3*/
```

Jeśli i-ty wierzchołek siatki danej jest brzegowy, to wywoływana w etapie 21 procedura `DSetOMfhei3` wykona dla tego wierzchołka instrukcje w liniach 11–54, a jeśli wewnętrzny, to instrukcje w liniach 57–76. W liniach 8 i 9 zmiennym `d` i `j` zostają przypisane odpowiednio stopień i-tego wierzchołka siatki danej i numer miejsca w tablicy `imvhei`, od którego zaczyna się lista numerów wychodzących z niego półkrawędzi.

Wierzchołkowi o numerze i w siatce danej odpowiada pewna liczba wierzchołków siatki wynikowej, o kolejnych numerach zaczynających się od `vcn[i]`. Na przykład wierzchołkowi brzegowemu 3 siatki z rysunku 31.1 odpowiadają wierzchołki 9, 10, 11, 12 siatki z rysunku 31.2, przy czym pierwszy i ostatni z tych wierzchołków są brzegowe. Półkrawędzie w ich otoczeniu są konstruowane odpowiednio w liniach 11–28 i 49–54, czyli przed i po pętli `for`, w której procedura tworzy dane opisujące półkrawędzie w otoczeniu wewnętrznych kopii i-tego wierzchołka (brzegowego) siatki danej.

W linii 11 zmienna `v0` ma przypisywany numer pierwszego wierzchołka siatki wynikowej odpowiadającego i-temu wierzchołkowi siatki danej (w przykładzie dla $i = 3$ jest `v0=9`). W linii 13 zmienna `f` otrzymuje wartość będącą numerem ściany, do której należy pierwsza półkrawędź wychodząca z i-tego wierzchołka, w rozpatrywanym przykładzie jest to ściana 3.

W linii 14 zmiennej `l` jest przypisywany numer wchodzącej do i-tego wierzchołka półkrawędzi tej ściany, czyli poprzedniej w zamkniętym w cykl ciągu półkrawędzi tej ściany (to jest półkrawędź brzegowa, 8 w rozpatrywanym przykładzie); służy do tego makrodefinicja `PREVIFAC_HEDGE` zamieszczona na listingu 31.14. W linii 15 zmiennej `ecnl` jest przypisy-

wany numer pierwszej z sześciu półkrawędzi wygenerowanych dla tej półkrawędzi w etapach 3 i 16 (w rozpatrywanym przykładzie półkrawędź ta ma numer 36).

Zmiennej q jest w linii 17 przypisywany identyfikator półkrawędzi brzegowej ściany siatki wynikowej odpowiadającej *i*-temu wierzchołkowi siatki danej (w przykładzie to jest półkrawędź 66 ściany 16), a zmienna p otrzymuje w linii 18 wartość identyfikującą poprzednią w cyklu półkrawędź tej ściany (numer 67 w przykładzie).

Informacje zapisywane do tablic zawierających reprezentację tworzonej siatki wynikowej w liniach 16 i 18–28 są numerami wierzchołków początkowego i końcowego półkrawędzi i numerami półkrawędzi do pary oraz numerem ściany; ponadto do list półkrawędzi wierzchołka (w linii 22 i 26) oraz ściany (w linii 28) są wpisywane numery odpowiednich półkrawędzi. W przykładzie półkrawędź q=66 otrzymuje numer pary -1 (bo reprezentuje krawędź brzegową), a półkrawędzie p=67 i ecnl+5=41 zostają połączone w parę. W linii 27 półkrawędzie 66 i 67 mają przypisywany numer ściany, do której należą, tj. 16 (numer ten jest sumą liczby ścian i krawędzi siatki danej oraz numeru *i* wierzchołka tej siatki).

Liczba półkrawędzi wychodzących z *i*-tego wierzchołka jest równa *d*; dla takiego wierzchołka jest generowanych $d + 2$ wierzchołków, z których *d* to wierzchołki wewnętrzne. Pętla w liniach 29–48 jest wykonywana *d* razy, aby wygenerować odpowiednie informacje dla tych wierzchołków. Po wykonaniu instrukcji w liniach 30–33 zmienne v0, l, f i ecnl przechowują odpowiednio numer wierzchołka początkowego półkrawędzi, numer kolejnej półkrawędzi wychodzącej z *i*-tego wierzchołka siatki danej, numer ściany tej półkrawędzi i numer pierwszej z czterech półkrawędzi siatki wynikowej wygenerowanych dla tej półkrawędzi. W przykładzie dla wierzchołka i=3 pętla zostanie wykonana dwa razy, za pierwszym razem jest v0=10, l=11, f=3 i ecnl=50, a za drugim v0=11, l=6, f=1 i ecnl=24. Dla półkrawędzi 11 zostały wygenerowane półkrawędzie 50, 51, 52 i 53, które w etapie 15 zostały połączone w dwie pary (listing 31.22, linie 15–18). W tym etapie trzeba jeszcze przypisać im odpowiednie numery wierzchołków początkowych i końcowych oraz wpisać ich numery do list półkrawędzi odpowiednich wierzchołków i ścian.

Wartości przypisane zmiennym v0, l, f i ecnl w liniach 30, 32, 43 i 44 to numer kolejnej kopii wierzchołka siatki danej (ta kopia w siatce wynikowej jest wierzchołkiem wewnętrznym), numer kolejnej półkrawędzi wychodzącej z *i*-tego wierzchołka siatki danej, numer jej ściany i numer pierwszej z półkrawędzi siatki wynikowej odpowiadających odpowiedniej półkrawędzi siatki danej. W przykładzie, w pierwszym przebiegu pętli jest v0=10, l=11, f=3 i ecnl=50, a w drugim v0=11, l=6, f=1 i ecnl=24.

W liniach 34, 35 do list półkrawędzi wierzchołka v0 (na początku) i ściany odpowiadającej *i*-temu wierzchołkowi siatki danej (na końcu) jest wpisywany numer półkrawędzi p, ustalony przed wejściem w pętlę (w linii 18) lub w poprzednim przebiegu pętli (w linii 42). W przykładzie za pierwszym razem jest p=67, a za drugim razem p=53. W linii 43 jest znajdowany numer półkrawędzi siatki danej poprzedzający półkrawędź l, tj. kończącej się w *i*-tym wierzchołku półkrawędzi ściany f, co umożliwia przypisanie odpowiedniej półkrawędzi siatki wynikowej numeru jej wierzchołka końcowego, czyli v0.

Po zakończeniu pętli pozostaje uzupełnienie informacji dla półkrawędzi w otoczeniu ostatniej kopii *i*-tego wierzchołka (w przykładzie kopia ta ma numer 12). Numer p pierwszej

wychodzącej z niego półkrawędzi (27 w przykładzie) został ustalony w ostatnim przebiegu pętli i trzeba go dopisać (w linii 49) do listy półkrawędzi ściany odpowiadającej i-temu wierzchołkowi. Ostatnie dwa przypisania (w liniach 53, 54) wpisują do listy półkrawędzi tej kopii wierzchołka indeksy dwóch półkrawędzi z niego wychodzących.

Postępowanie dla ściany odpowiadającej wierzchołkowi wewnętrznemu siatki danej jest prostsze, bo wszystkie wierzchołki tej ściany są wewnętrzne, czyli jednakowego rodzaju. Wydaje mi się, że zamieszczanie szczegółowego opisu instrukcji w liniach 57–76 jest zbędne, bo po lekturze opisu postępowania ze ścianą dla i-tego wierzchołka brzegowego Czytelnik jest w stanie poradzić sobie z rozszyfrowaniem działania tego fragmentu procedury. Może w tym pomóc przykład: wierzchołkowi 0 siatki z rysunku 31.1 odpowiada ściana 13 siatki wynikowej z rysunku 31.2. Podczas przetwarzania tej ściany zmienna `v0` przyjmuje (w kolejnych przebiegach pętli) wartości 0, 1, 2. Zmienna `f` przyjmuje wartści 0, 1, 3, a zmiennej `ecnl` w linii 62 są nadawane kolejno wartości 0, 62, 20, a w linii 72 wartości 8, 12, 50, identyfikujące odpowiednie krawędzie siatki wynikowej.

31.9. Implementacja uśredniania

Listing 31.25 przedstawia procedurę w C, która wywołuje szader w celu zrealizowania kolejno wszystkich etapów uśredniania. Numery tych etapów następują po numerach etapów podwajania, a procedury i makrodefinicje realizujące poszczególne etapy są przedstawione na kolejnych listingach.

Listing 31.25. Procedura uśredniania

C

```
1:  char GPUmeshAveraging ( GPUmesh *inmesh, GPUmesh *outmesh )
2:  {
3:    int    inv, inhe, infac, invb, onv, onhe, onfac;
4:    GLint  bufsize, bs;
5:    GLuint auxbuf = 0;
6:
7:    glUseProgram ( progid[0] );
8:    glBindBufferBase ( GL_UNIFORM_BUFFER, rbbp, rbuf );
9:    inv = inmesh->nv;  inhe = inmesh->nhe;  infac = inmesh->nfac;
10:   glBindBufferBase ( SSB, 1, inmesh->MVFBUF );
11:   glBindBufferBase ( SSB, 2, inmesh->MHEBUF );
12:   glBindBufferBase ( SSB, 3, inmesh->VCBUF );
13:   SETUVAR ( 1, GLint, inmesh->nsattr )
14:   SETUVAR ( 2, GLint, inv )
15:   SETUVAR ( 3, GLint, inhe )
16:   SETUVAR ( 4, GLint, infac )
17:   bs = inv > infac ? inv : infac;
18:   bufsize = (2*inv+2*inhe+infac+bs)*sizeof(GLint);
19:   glGenBuffers ( 1, &auxbuf );
20:   glBindBufferBase ( SSB, 0, auxbuf );
```

```
21:  glBufferData ( SSB, bufsize, NULL, GL_DYNAMIC_DRAW );
22:  ExecStage ( uvofs, 2, inv );     /* TagVertex */
23:  SumUp ( uvofs, 0, inv );
24:  glGetBufferSubData ( SSB, 0, sizeof(GLint), &invb );
25:  onfac = inv - invb;  SETUVAR ( 7, GLint, onfac )
26:  if ( invb == 0 ) {
27:    onv = infac;  onhe = inhe;
28:    ExecStage ( uvofs, 22, infac );  /* ASetNvi1 */
29:    ExecStage ( uvofs, 23, inhe );   /* ASetNhei1 */
30:    ExecStage ( uvofs, 24, inv );    /* ASetNfi1 */
31:  }
32:  else {
33:    ExecStage ( uvofs, 25, infac );  /* ASetNvi2 */
34:    PrefixSum ( uvofs, 0, infac );
35:    SumUp ( uvofs, 2*infac+inhe+inv, infac );
36:    glGetBufferSubData ( SSB, (2*infac+inhe+inv)*sizeof(GLint),
37:                         sizeof(GLint), &onv );
38:    ExecStage ( uvofs, 26, inv );    /* ASetNfi2, true */
39:    PrefixSum ( uvofs, 2*infac+inhe, inv );
40:    ExecStage ( uvofs, 27, inhe );   /* ASetNhei2 */
41:    PrefixSum ( uvofs, 2*infac, inhe );
42:    glGetBufferSubData ( SSB, (2*infac+inhe-1)*sizeof(GLint),
43:                         sizeof(GLint), &onhe );
44:    ExecStage ( uvofs, 28, infac );  /* ASetNvi2, false */
45:    ExecStage ( uvofs, 29, inhe );   /* ASetNhei3 */
46:    ExecStage ( uvofs, 30, inv );    /* ASetNfi3 */
47:  }
48:  SETUVAR ( 5, GLint, onv )
49:  SETUVAR ( 6, GLint, onhe )
50:  if ( !ReallocGPUmesh ( outmesh, onv, onhe, onfac, inmesh->nsattr,
51:                         inmesh->pdim, inmesh->pofs, inmesh->nvofs ) )
52:    goto failure;
53:  glBindBufferBase ( SSB, 4, outmesh->MVFBUF );
54:  glBindBufferBase ( SSB, 5, outmesh->MHEBUF );
55:  glBindBufferBase ( SSB, 6, outmesh->VCBUF );
56:  ExecStage ( uvofs, 31, inv );    /* AClearFVd */
57:  ExecStage ( uvofs, 32, inv );    /* ASetFVd1 */
58:  PrefixSum ( uvofs, 2*infac+inhe+inv, inv );
59:  ExecStage ( uvofs, 33, inv );    /* ASetFVd2 */
60:  ExecStage ( uvofs, 34, infac );  /* AClearFvd */
61:  ExecStage ( uvofs, 35, infac );  /* ASetOMVert, true */
62:  PrefixSum ( uvofs, 2*infac+inhe+inv, infac );
63:  ExecStage ( uvofs, 36, infac );  /* ASetOMVert, false */
64:  ExecStage ( uvofs, 37, inhe );   /* ABindHe */
65:  ExecStage ( uvofs, 38, inv );    /* ASetOMfacHe */
66:  ExecStage ( uvofs, 39, infac );  /* Average */
67:  glUseProgram ( 0 );
```

```
68:   glDeleteBuffers ( 1, &auxbuf );
69:   ExitIfGLError ( "GPUmeshAveraging" );
70:   return true;
71:
72: failure:
73:   glUseProgram ( 0 );
74:   glDeleteBuffers ( 1, &auxbuf );
75:   return false;
76: } /*GPUMeshAveraging*/
```

W chwili wywołania procedury uśredniania siatka dana jest reprezentowana przez zawartość tablic umieszczonych w odpowiednich buforach w pamięci GPU; instrukcje w liniach 10–16 przywiązują te bufory do odpowiednich punktów dowiązania w celu GL_SHADER_STORAGE_BUFFER i przypisują zmiennym jednolitym nsattr, inv, inhe i infac liczbę atrybutów wierzchołka i liczby wierzchołków, półkrawędzi i ścian siatki danej. W liniach 17–18 jest obliczana potrzebna wielkość bufora pomocniczego, a następnie bufor ten jest rezerwowany i przywiązywany do punktu 0 w celu GL_SHADER_STORAGE_BUFFER. W liniach 22–23 szader zlicza wierzchołki (czyli także krawędzie) brzegowe siatki danej. Dalsze obliczenia zależą od tego, czy liczba tych wierzchołków jest zerem (wtedy wykonywane są instrukcje w liniach 27–30), czy też nie (i wtedy trzeba wykonać instrukcje w liniach 33–46).

Zmienne jednolite wykorzystywane przez procedurę uśredniania są przedstawione na listingu 31.9, a ponadto podczas uśredniania jest potrzebnych pięć tablic liczb całkowitych przechowywanych we wspomnianym wcześniej pomocniczym buforze magazynowym seq dowiązanym do punktu dowiązania 0; odpowiednie makrodefinicje ułatwiające dostęp do tych tablic są pokazane na listingu 31.26.

Listing 31.26. Tablice pomocnicze procedury uśredniania

GLSL

```
1: #define nvi(I)    seq.a[I]
2: #define fvnum(I)  seq.a[infac+(I)]
3: #define nhei(I)   seq.a[2*infac+(I)]
4: #define nfi(I)    seq.a[2*infac+inhe+(I)]
5: #define fvd(I)    seq.a[2*infac+inhe+inv+(I)]
```

Listing 31.27 przedstawia instrukcje realizujące etapy 22–24, wykonywane, gdy siatka dana nie ma brzegu, tj. gdy wszystkie jej wierzchołki i krawędzie są wewnętrzne. Wtedy każdej ścianie odpowiada jeden wierzchołek siatki wynikowej (makro ASetNvi wpisuje do tablicy nvi, na i-tym miejscu numer ściany, której odpowiada wierzchołek, równy i, a w tablicy fvnum liczbę wierzchołków odpowiadających tej ścianie, czyli 1). Również każdej półkrawędzi siatki danej odpowiada jedna półkrawędź siatki wynikowej, może ona mieć ten sam numer, wpisywany do tablicy nhei przez makro ASetNhei1. Ponadto i-temu wierzchołkowi siatki danej odpowiada jedna, i-ta ściana siatki wynikowej, numer i jest wpisywany przez makro ASetNfi1 do tablicy nfi.

Listing 31.27. Początkowe etapy uśredniania dla siatek bez krawędzi brzegowych

GLSL

```
#define ASetNvi1(i) { nvi(i) = int(i); fvnum(i) = 1; }  /* etap 22 */
#define ASetNhei1(i) nhei(i) = int(i);  /* etap 23 */
#define ASetNfi1(i) nfi(i) = int(i);  /* etap 24 */
```

Bardziej skomplikowane obliczenia są potrzebne podczas uśredniania siatek zawierających wierzchołki i krawędzie brzegowe. Zadaniem procedury `ASetNvi2` (listing 31.28) jest znalezienie, dla i-tej ściany siatki danej, liczby odpowiadających jej wierzchołków siatki wynikowej. Procedura ta jest wywoływana dwukrotnie, w etapach 25 i 28, przy czym za pierwszym razem parametr `first` ma wartość `true`, a za drugim `false` (listing 31.12, linie 36 i 39). Pętla w liniach 8–14 służy do zbadania, czy wszystkie wierzchołki ściany są wewnętrzne; jeśli tak, to ścianie tej odpowiada jeden wierzchołek (co zostaje odnotowane w linii 16). W przeciwnym razie instrukcje w liniach 19–27 znajdują liczbę spójnych podciągów wierzchołków wewnętrznych półkrawędzi tej ściany.

Listing 31.28. Liczenie wierzchołków odpowiadających ścianie siatki z brzegiem

GLSL

```
1:  void ASetNvi2 ( uint i, bool first )  /* etap 25, 28 */
2:  {
3:    int  d, fhe, v0, v1, j, k, l;
4:    bool s0, s1;
5:
6:    d = imfac(i) >> DEGSHIFT;
7:    fhe = imfac(i) & FHEMASK;
8:    for ( j = 0, s0 = true;  j < d;  j++ ) {
9:      v0 = imhe(imfhei(fhe+j)).V0;
10:     if ( (imv(v0) & TAGMASK) != 0 ) {
11:       s0 = false;
12:       break;
13:     }
14:   }
15:   if ( s0 )
16:     k = 1;
17:   else {
18:     s0 = (imv(v0) & TAGMASK) != 0;
19:     for ( l = k = 0;  l < d;  l++ ) {
20:       v1 = imhe(imfhei(fhe+j)).V1;
21:       s1 = (imv(v1) & TAGMASK) != 0;
22:       if ( s0 && !s1 )
23:         k ++;
24:       v0 = v1;  s0 = s1;
25:       j = j >= d-1 ? 0 : j+1;
26:     }
27:   }
28:   if ( first ) {
```

```
29:     fvnum(i) = fvd(i) = k;
30:     if ( i == 0 )
31:       nvi(0) = 0;
32:     if ( i < infac-1 )
33:       nvi(i+1) = k;
34:   }
35:   else if ( k == 0 )
36:     nvi(i) = -1;
37: } /*ASetNvi2*/
```

Podczas pierwszego wywołania procedura zapisuje liczbę k wierzchołków odpowiadających i-tej ścianie w tablicach `fvnum` i `fvd`, a w tablicy `nvi`, na pozycji $i+1$; ponadto następuje przypisanie `nvi[0] = 0;`, po czym (listing 31.25, linie 34 i 35) w tablicy `nvi` jest obliczany ciąg sum prefiksowych, a w tablicy `fvd` jest obliczana suma wszystkich liczb. Suma ta, odczytywana w liniach 36–37, jest całkowitą liczbą wierzchołków siatki wynikowej.

Jeśli i-tej ścianie odpowiada 0 wierzchołków, to podczas drugiego wywołania procedury `ASetNvi2` dla wyróżnienia tego faktu zmiennej `nvi[i]` jest przypisywana wartość `-1`.

Procedura `ASetNfi2` (listing 31.29), wykonywana w etapie 26, wpisuje do tablicy `nfi` ciąg zer i jedynek; zero odpowiada wierzchołkowi brzegowemu siatki danej, a jedynka wewnętrznemu, przy czym `nfi[0]` otrzymuje wartość 0, a dla i-tego wierzchołka odpowiadająca mu liczba trafia do tablicy na miejsce $i+1$. Następnie jest obliczany ciąg sum prefik-

Listing 31.29. Kolejne etapy wstępne uśredniania siatki z brzegiem

GLSL

```
1: void ASetNfi2 ( uint i )  /* etap 26 */
2: {
3:   if ( i == 0 )
4:     nfi(i) = 0;
5:   else
6:     nfi(i) = int((imv(i-1) & TAGMASK) == 0);
7: } /*ASetNfi2*/
8:
9: void ASetNhei2 ( uint i )  /* etap 27 */
10: {
11:   if ( i == 0 )
12:     nhei(i) = 0;
13:   else
14:     nhei(i) = int((imv(imhe(i-1).V1) & TAGMASK) == 0);
15: } /*ASetNhei2*/
16:
17: #define ASetNhei3(i)  /* etap 29 */ \
18:   { if ( (imv(imhe(i).V1) & TAGMASK) != 0 ) nhei(i) = -1; }
19: #define ASetNfi3(i)  /* etap 30 */ \
20:   { if ( (imv(i) & TAGMASK) != 0 ) nfi(i) = -1; }
```

sowych w tej tablicy. Elementy tego ciągu przyporządkowują wierzchołkom wewnętrznym siatki danej odpowiadające im numery ścian siatki wynikowej.

Podobnie procedura ASetNhei2 (etap 27) dla półkrawędzi $i-1$ siatki danej przypisuje zmiennej nhei[i] zero albo jedynkę; jedynkę wtedy, gdy końcowy wierzchołek półkrawędzi nie jest brzegowy. Dla wpisanego do tablicy ciągu (z zerem na początku) jest obliczany ciąg sum prefiksowych, który określa numery półkrawędzi siatki wynikowej odpowiadające półkrawędziom siatki danej. W etapie 29 jest wykonywana instrukcja w makrodefinicji ASetNhei3, która dla półkrawędzi zakończonych wierzchołkiem brzegowym wpisuje liczbę -1 na miejsce obliczonej (i nieistotnej dla takiej półkrawędzi) sumy prefiksowej. Podobne zadanie wykonuje makro ASetNfi3 (etap 30), które do tablicy nfi wpisuje -1 dla wierzchołków brzegowych — w siatce wynikowej nie ma ścian odpowiadających takim wierzchołkom.

Procedura GPUmeshAveraging po zakończeniu opisanych wyżej etapów ma obliczone liczby wierzchołków, półkrawędzi i ścian siatki wynikowej i w linii 50 dokonuje rezerwacji buforów w pamięci GPU, w których ma się znaleźć ta siatka. W liniach 53–55 procedura przywiązuje te bufory do odpowiednich punktów dowiązania.

Listing 31.30. Tworzenie list półkrawędzi dla ścian siatki wynikowej

GLSL

```
1: #define AClearFVd(i) fvd(i) = 0;  /* etapy 31 i 34 */
2:
3: void ASetFVd1 ( uint i )  /* etap 32 */
4: {
5:   int k;
6:
7:   if ( (k = nfi(i)) >= 0 )
8:     fvd(k) = imv(i) >> DEGSHIFT;
9: } /*ASetFVd1*/
10:
11: void ASetFVd2 ( uint i )  /* etap 33 */
12: {
13:   int k;
14:
15:   if ( (k = nfi(i)) >= 0 )
16:     omfac(k) = (imv(i) & DEGMASK) | (k > 0 ? fvd(k-1) : 0);
17: } /*ASetFVd2*/
```

Pokazane na listingu 31.30 procedury, wykonywane w etapach 31–33, znajdują, dla ścian siatki wynikowej, pozycje początków ich list półkrawędzi w tablicy omfhei. W tym celu makro AClearFVd kasuje zawartość tablicy fvd, po czym procedura ASetFVd1, jeśli i-ty wierzchołek siatki danej jest wewnętrzny (czyli odpowiada mu pewna ściana siatki wynikowej, ma ona numer k znaleziony wcześniej i zapisany w tablicy nfi), zmiennej fvd[k] przypisuje stopień tej ściany (który będzie stopniem tego wierzchołka). Po zakończeniu etapu 32 w tablicy fvd zostaje obliczony ciąg sum prefiksowych, określający pozycje początków list półkrawędzi. Procedura ASetFVd2, wywołana dla i-tego wierzchołka, jeśli odpowiada mu

ściana (o numerze k), zapisuje w tablicy `omfac` stopień tej ściany (który jest stopniem i-tego wierzchołka) i pozycję początku jej listy półkrawędzi.

W etapie 34 tablica `fvd` jest ponownie kasowana za pomocą procedury `AClearFVd` (przy czym teraz jej długość jest równa n_f), po czym w etapie 35 następuje pierwsze wywołanie (z parametrem `first` równym `true`) procedury `ASetOMVert` pokazanej na listingu 31.31. Parametr `i` jest numerem ściany siatki danej.

Listing 31.31. Tworzenie wierzchołków siatki wynikowej

GLSL

```
void ASetOMVert ( uint i, bool first )  /* etap 35, 36 */
{
  int n, d, fhe, r, s, t, j, k, l, v0, v1, m, e;

  if ( (r = fvnum(i)) > 0 ) {
    n = nvi(i);
    d = imfac(i) >> DEGSHIFT;
    fhe = imfac(i) & FHEMASK;
    for ( k = 0; k < d; k++ ) {
      v1 = imhe(imfhei(fhe+k)).V1;
      if ( (imv(v1) & TAGMASK) != 0 )
        break;
    }
    if ( !first )
      j = i > 0 ? fvd(i-1) : 0;
    for ( s = 0;  s < r;  s++, n++ ) {
      do {
        k = k >= d-1 ? 0 : k+1;
        v1 = imhe(imfhei(fhe+k)).V1;
      } while ( (imv(v1) & TAGMASK) != 0 );
      for ( m = 0, t = (k+1) % d;  m < d;  m++, t = (t+1) % d ) {
        v0 = imhe(imfhei(fhe+t)).V0;
        if ( (imv(v0) & TAGMASK) != 0 )
          break;
      }
      if ( first )
        fvd(i) += m;
      else {
        omv(n) = (m << DEGSHIFT) | j;
        for ( l = m-1, t = k;  l >= 0;  l--, t = (t+1) % d ) {
          v1 = imhe(imfhei(fhe+t)).V1;
          omvhei(j+l) = e = nhei(imfhei(fhe+t));
          omhe(e).V0 = n;
          omhe(e).FACN = nfi(v1);
        }
        j += m;
      }
    }
```

```
39:   }
40: } /*ASetOMVert*/
```

Pierwszą czynnością w procedurze jest sprawdzenie, czy liczba r wierzchołków siatki wynikowej odpowiadających tej ścianie jest dodatnia (ta informacja jest obecna w tablicy `fvnum`). Jeśli $r = 0$, to procedura nie ma nic do roboty. W przeciwnym razie w pętli w liniach 9–13 jest wyszukiwana (i zapamiętywana w zmiennej `k`) pozycja pierwszego identyfikatora półkrawędzi, której końcowy wierzchołek jest brzegowy.

W pętli w liniach 16–38 lista półkrawędzi ściany jest przeglądana (cyklicznie) od tego miejsca i dla każdego spójnego podciągu półkrawędzi, których końce są wierzchołkami wewnętrznymi, jest tworzony kolejny wierzchołek siatki wynikowej odpowiadający i-tej ścianie. Pętla w liniach 17–20 ma na celu znalezienie pierwszej półkrawędzi w liście, której końcowy wierzchołek jest wewnętrzny, a pętla w liniach 21–25 znajduje ostatnią taką półkrawędź. Wartość zmiennej `m` po zakończeniu tej pętli jest liczbą półkrawędzi wychodzących z tworzonego wierzchołka; liczby półkrawędzi wychodzących ze wszystkich wierzchołków utworzonych dla i-tej ściany są (w pierwszym wywołaniu procedury) sumowane na i-tej pozycji w tablicy `fvd`.

Po zakończeniu etapu 35 obliczany jest ciąg sum prefiksowych w tablicy `fvd`. W etapie 36 procedura `ASetOMVert` (listing 31.31) jest wywoływana ponownie, z parametrem `first` równym `false`. Podczas tego wywołania lista półkrawędzi i-tej ściany jest ponownie przeszukiwana. Liczba na i-tym miejscu w tablicy `fvd` określa miejsce w tablicy `omvhei`, od którego zaczyna się lista półkrawędzi pierwszego wierzchołka siatki wynikowej odpowiadającego i-tej ścianie; w linii 15 jest ona przypisywana zmiennej `j`. Instrukcje w liniach 29–36 tworzą opisy wierzchołków. W linii 29 jest zapisywany stopień i pozycja początku listy kolejnego wierzchołka (który ma numer n), a w pętli w liniach 30–35 numery półkrawędzi wychodzących z tego wierzchołka są zapisywane w liście (tj. wpisywane do tablicy `omvhei`). W reprezentacji każdej z tych półkrawędzi jest zapisywany numer jej wierzchołka początkowego (czyli n) i numer jej ściany.

Listing 31.32. Łączenie półkrawędzi w pary

GLSL

```
1: void ABindHe ( uint i )  /* etap 37 */
2: {
3:   int k;
4:
5:   if ( (k = nhei(i)) >= 0 )
6:     omhe(k).OTHE = nhei(imhe(i).OTHE);
7: } /*ABindHe*/
```

Zadaniem procedury `ABindHe` wykonywanej w etapie 37 (listing 31.32) jest połączenie w pary półkrawędzi reprezentujących krawędzie wewnętrzne siatki wynikowej. Parametr `i` jest numerem półkrawędzi w siatce danej, a zmiennej `k` jest przypisywany numer odpowiadającej jej półkrawędzi siatki wynikowej, znaleziony i zapamiętany w tablicy `nhei` w etapie 23 (jeśli siatka dana nie ma brzegu) albo 27 (jeśli ma).

Listing 31.33. Tworzenie list półkrawędzi ścian

GLSL

```
1: void ASetOMfacHe ( uint i )  /* etap 38 */
2: {
3:   int k, d, j, l, m, v1;
4:
5:   if ( (k = nfi(i)) >= 0 ) {
6:     d = imv(i) >> DEGSHIFT;
7:     j = imv(i) & FHEMASK;
8:     l = omfac(k) & FHEMASK;
9:     for ( m = 0; m < d; m++ )
10:      omfhei(l+m) = nhei(imhe(imvhei(j+d-1-m)).OTHE);
11:    for ( m = d-1, v1 = omhe(omfhei(l)).V0;
12:          m >= 0;
13:          v1 = omhe(omfhei(l+m)).V0, m-- )
14:      omhe(omfhei(l+m)).V1 = v1;
15:  }
16: } /*ASetOMfacHe*/
```

W etapie 38 procedura `ASetOMfacHe` tworzy listy półkrawędzi ścian siatki wynikowej i uzupełnia informacje w reprezentacji półkrawędzi, przypisując im numery wierzchołków końcowych.

Wisienką na torcie jest ostatni etap uśredniania, czyli numeryczne obliczenie środków ciężkości zbiorów wierzchołków ścian siatki danej. Wykonuje go procedura `Average` przedstawiona na listingu 31.34. Dla i-tej ściany, której odpowiada co najmniej jeden wierzchołek siatki wynikowej (to sprawdzane jest w linii 6) procedura odczytuje z tablicy `nvi` numer n pierwszego odpowiadającego jej wierzchołka oraz stopień ściany, czyli liczbę jej wierzchołków. Pętle w liniach 12–13 i 14–18 obliczają sumy poszczególnych atrybutów (np. współrzędnych położeń) tych wierzchołków, a pętla w liniach 20–21 dzieli obliczone sumy przez stopień. Zadaniem pętli w liniach 22–23 jest wykonanie odpowiedniej liczby kopii obliczonych atrybutów, dla wszystkich wierzchołków siatki wynikowej odpowiadających i-tej ścianie siatki danej (zobacz opis etapów 24–27).

Listing 31.34. Obliczanie atrybutów wierzchołków

GLSL

```
1: void Average ( uint i )  /* etap 39 */
2: {
3:   int   r, n, d, j, iv, ov, k, l;
4:   float id;
5:
6:   if ( (r = fvnum(i)) > 0 ) {
7:     n = nvi(i);
8:     ov = n*nsattr;
9:     d = imfac(i) >> DEGSHIFT;
10:    j = imfac(i) & FHEMASK;
11:    iv = imhe(imfhei(j)).V0*nsattr;
```

```
12:     for ( l = 0; l < nsattr; l++ )
13:       omvc(ov+l) = imvc(iv+l);
14:     for ( k = 1; k < d; k++ ) {
15:       iv = imhe(imfhei(j+k)).V0*nsattr;
16:       for ( l = 0; l < nsattr; l++ )
17:         omvc(ov+l) += imvc(iv+l);
18:     }
19:     id = 1.0/float(d);
20:     for ( l = 0; l < nsattr; l++ )
21:       omvc(ov+l) *= id;
22:     for ( l = 0; l < (r-1)*nsattr; l++ )
23:       omvc(ov+l+nsattr) = omvc(ov+l);
24:   }
25: } /*Average*/
```

31.10. Procedura zagęszczania siatki

Listing 31.35 przedstawia procedurę, która posługując się opisanymi wcześniej procedurami podwajania i uśredniania dokonuje zagęszczania siatki. Przed jej użyciem należy przygotować do pracy program szaderów, którego części są przedstawione na listingach 31.9, 31.12, 31.14–31.24 oraz 31.26–31.34 i G.8 i umieścić siatkę daną w pamięci GPU. Procedury, które to robią, są przedstawione na listingach 31.10 i 31.8.

Uwaga: W przedstawionej tu implementacji zagęszczania brakuje sprawdzeń koniecznych w oprogramowaniu ogólnoużytkowym. W wersji „komercyjnej" powinno być sprawdzenie, czy operacja podwajania nie wytworzy siatki, która ma więcej niż 2^{25} półkrawędzi, a także sprawdzenia, czy siatka dana nie jest lub siatka otrzymana przez uśrednianie nie będzie pusta.

Pierwszy parametr procedury `GPUmeshRefinement` jest jest liczbą kroków uśredniania po zagęszczaniu siatki, musi mieć wartość co najmniej 1 (i co najwyżej kilka). Drugi parametr jest wskaźnikiem struktury, w której są przechowywane informacje o siatce danej: liczby wierzchołków, półkrawędzi i ścian, wymiar przestrzeni wierzchołków (tj. liczba współrzędnych położenia każdego wierzchołka i innych atrybutów) oraz tablica z identyfikatorami buforów OpenGL-a z tablicami zawierającymi reprezentację siatki. Parametr trzeci wskazuje analogiczną strukturę, w której zostanie zapisana informacja o wyniku zagęszczania, przy czym może w niej być albo poprawna reprezentacja innej siatki (która zostanie usunięta) albo należy tę strukturę (np. przed pierwszym wywołaniem procedury zagęszczania) wypełnić zerami.

Procedura `GPUmeshRefinement` posługuje się dodatkową strukturą typu `GPUmesh` do reprezentowania pośrednich wyników zagęszczania — wyniku podwajania i kolejnych uśrednień oprócz ostatniego. Wywołanie procedury `glDeleteBuffers` w linii 16 likwiduje reprezentację już niepotrzebnej siatki, która została poddana uśrednianiu. Kasowanie (w linii 17) zmiennych, w których były pamiętane identyfikatory buforów, jest w zasadzie zbędne; procedura `ReallocGPUmesh`, wywołana przez procedury podwajania i uśredniania (listing 31.13,

linie 30–31 i listing 31.25, linie 50–51), ponownie wywołałaby procedurę `glDeleteBuffers`, aby zwolnić bufory przed ich ponownym utworzeniem (i nadaniem nowej potrzebnej wielkości), co *nie jest* szkodliwe[12], ale zapobieganie temu przez odpowiednio napisany kod uważam za przejaw większej elegancji stylu programowania.

Listing 31.35. Procedura zagęszczania siatki

C

```
char GPUmeshRefinement ( int n, GPUmesh *inmesh, GPUmesh *outmesh )
{
  GPUmesh mmesh, *am, *bm, *cm;
  int     i;

  if ( n < 1 )
    return false;
  memset ( &mmesh, 0, sizeof(GPUmesh) );
  if ( n & 0x01 ) { am = &mmesh;  bm = outmesh; }
             else { am = outmesh;  bm = &mmesh; }
  if ( !GPUmeshDoubling ( inmesh, am ) )
    goto failure;
  for ( i = 0; i < n; i++ ) {
    if ( !GPUmeshAveraging ( am, bm ) )
      goto failure;
    glDeleteBuffers ( 4, am->mbuf );
    memset ( am->mbuf, 0, 4*sizeof(GLuint) );
    cm = am;  am = bm;  bm = cm;
  }
  return true;

failure:
  glDeleteBuffers ( 4, am->mbuf );
  glDeleteBuffers ( 4, bm->mbuf );
  return false;
} /*GPUmeshRefinement*/
```

Siatkę o stosunkowo niewielkiej liczbie wierzchołków i ścian możemy poddać zagęszczaniu, którego wynik możemy również zagęścić i powtórzyć to kilka razy — otrzymamy w ten sposób kilka siatek, których możemy użyć do wybrania odpowiedniego poziomu szczegółowości rysowanego modelu. Jeśli powierzchnia na obrazie jest mała, to wystarczy wyświetlić obraz siatki o małej liczbie ścian, nie marnując czasu na niedostrzegalne na obrazie szczegóły. Siatki gęste przydadzą się do wykonania obrazów przedstawiających powierzchnie w powiększeniu.

[12] Podobnie zachowują się inne procedury gospodarowania zasobami w OpenGL-u; jeśli przekazany procedurze likwidacji identyfikator nie jest związany z istniejącym obiektem takim jak bufor, obraz, tekstura, programem szaderów itd., to jest przez tę procedurę ignorowany, bez sygnalizowania błędu.

Czytelnik może zadawać sobie pytanie: jak szader o takiej wielkości jak opisany w tym rozdziale szader do zagęszczania siatek napisać i doprowadzić do działania? Podstawową pomocą dla mnie były napisane wcześniej sekwencyjne implementacje algorytmów podwajania i uśredniania działające na CPU. W roboczych wersjach procedur `GPUmeshDoubling` i `GPUmeshAveraging` umieściłem po wywołaniach poszczególnych etapów obliczeń instrukcje przepisujące zawartość buforów do tablicy w pamięci CPU, co umożliwiło mi porównywanie wyników obliczeń w tych etapach z częściowymi wynikami obliczeń implementacji sekwencyjnej. Podstawowa zasada jest taka, aby zabierać się do programowania kolejnego etapu obliczeń dopiero po uruchomieniu etapów wcześniejszych. Trzeba się liczyć z tym, że zaimplementowanie i uruchomienie algorytmu na GPU może zabrać dużo więcej czasu niż napisanie algorytmu sekwencyjnego — tak było w tym przypadku.

31.11. *Uzupełnienia

Numeryczne obliczanie współrzędnych wierzchołków siatki będącej wynikiem podwajania lub uśredniania jest tylko jednym z wielu etapów obliczeń; pozostałe etapy mają na celu znalezienie list półkrawędzi wychodzących z poszczególnych wierzchołków lub otaczających ściany oraz powiązań półkrawędzi w pary, co (nie bez powodu) będziemy nazywać **topologią siatki**. Obliczanie współrzędnych zajmuje niewielką część czasu trwania całej operacji przetwarzania siatki[13]. Jeśli podczas działania aplikacji trzeba zagęszczać siatkę, której wierzchołki zmieniają położenia, ale topologia pozostaje niezmieniona, warto podzielić zagęszczanie siatki na dwie procedury. Topologię siatki zagęszczonej wystarczy znaleźć tylko raz, w preprocesingu, a później, podczas rysowania kolejnych klatek animacji, pozostaje tylko obliczanie współrzędnych.

31.11.1. Macierz zagęszczania

Każdy wierzchołek siatki otrzymanej przez podwajanie jest kopią jednego, a każdy wierzchołek wyniku uśredniania jest środkiem ciężkości (średnią arytmetyczną) kilku wierzchołków ściany siatki danej. Można stąd udowodnić, że każdy wierzchołek siatki otrzymanej przez zagęszczanie jest kombinacją afiniczną (zobacz s. 108) wierzchołków siatki danej. Dzięki temu obliczenie współrzędnych może być wykonane przez mnożenie macierzy

$$Y = RX.$$

Symbol X w powyższym wzorze oznacza macierz o wymiarach $n_v \times d$, której każdy wiersz składa się z d współrzędnych kolejnego wierzchołka siatki danej; liczba n_v jest liczbą tych wierzchołków. Podobnie, wiersze macierzy Y o wymiarach $m_v \times d$ reprezentują wierzchołki siatki zagęszczonej. Macierz R o wymiarach $m_v \times n_v$, którą nazwiemy **macierzą zagęszczania**, jest **rzadka**, tj. większość jej współczynników jest równa 0. Pozostałe współczynniki są dodatnie, a w każdym wierszu ich suma jest równa 1.[14]

[13] Jaką dokładnie, to zależy od siatki i od procesora graficznego.

[14] Macierz o tej własności jest nazywana **macierzą stochastyczną**. Iloczyn takich macierzy też jest macierzą stochastyczną.

Macierz $R = A_n \ldots A_1 D$ jest iloczynem **macierzy podwajania** D oraz n **macierzy uśredniania** A_k. Jeśli i-ty wierzchołek siatki otrzymanej przez podwajanie jest kopią j-tego wierzchołka siatki danej, to współczynnik d_{ij} macierzy D jest równy 1, a pozostałe współczynniki w i-tym wierszu są zerem. Z kolei w i-tym wierszu macierzy A_k jest s niezerowych współczynników; są one równe $1/s$, gdzie s jest stopniem, tj. liczbą wierzchołków pewnej ściany siatki poddawanej uśrednianiu. Współczynniki te znajdują się w kolumnach, których numery są numerami tych wierzchołków[15]. Zapewnia to, że i-ty wierzchołek siatki otrzymanej przez uśrednianie jest środkiem ciężkości wierzchołków tej ściany.

Macierz R jest nieregularna, tj. jej niezerowe współczynniki są rozmieszczone dowolnie. Podrozdział G.4 zawiera opis sposobu reprezentowania takich macierzy w pamięci GPU i implementacji algorytmów ich przetwarzania, odpowiednich do naszych celów.

Listing 31.36. Opakowanie macierzy zagęszczania

C

```
1: typedef struct {
2:     GPUmesh         *cm, *fm;
3:     GPUSparseMatrix mat;
4:   } MeshRefineMatrix;
```

Na listingu 31.36 jest pokazana struktura służąca jako opakowanie macierzy zagęszczania. Ponieważ macierz jest związana z siatkami o ustalonych topologiach, struktura zawiera wskaźniki `cm` i `fm` opakowań tych siatek. Pole `mat` jest opakowaniem samej macierzy; definicja typu `GPUSparseMatrix` jest zamieszczona na listingu G.13. W polu tym są przechowywane wymiary macierzy, całkowita liczba niezerowych współczynników i identyfikatory buforów w pamięci GPU, w których są przechowywane tablice z położeniami niezerowych współczynników i te współczynniki.

31.11.2. Szader i procedura znajdowania macierzy zagęszczania

Aby otrzymać szader obliczeniowy i procedury, które znajdują tę samą topologię zagęszczonej siatki, ale zamiast obliczać współrzędne wierzchołków znajdują macierz zagęszczania, wystarczy szader i procedury opisane w podrozdziale 31.5–31.10 poddać niewielkim zmianom. Wszystkie zmienne jednolite i tablice robocze używane przez szader zagęszczania są niezmienione. Zamiast bloków magazynowych `Invc` i `Outvc`, przywiązanych do punktów 3 i 6, trzeba wprowadzić pokazane na listingu 31.37 bloki magazynowe, w których szader ma pozostawić reprezentację macierzy podwajania lub uśredniania. Przeznaczyłem dla nich punkty dowiązania 3 i 6.

Makrodefinicje `rd` i `cd` (linie 4 i 5) ułatwiają dostęp do przechowywanych w jednym buforze tablic `r` i `c` tworzonej przez szader reprezentacji macierzy podwajania lub uśredniania. Liczba wierszy macierzy jest liczbą wierzchołków siatki wynikowej, długość tablicy `r` jest o 1 większa.

[15] Wiersze i kolumny macierzy numerujemy, zaczynając od 0.

Listing 31.37. Bloki magazynowe dla macierzy podwajania lub uśredniania

GLSL

```
layout(std430,binding=3) buffer Outrc { uint  rc[]; } outrc;
layout(std430,binding=6) buffer Outa  { float a[];  } outa;

#define rd(I) outrc.rc[I]
#define cd(I) outrc.rc[outnv+1+(I)]
```

Procedurę DCopyVC z listingu 31.17, która kopiuje wierzchołki, trzeba zmienić na (tak samo nazwaną) procedurę pokazaną na listingu 31.38. Jej zadaniem jest wypełnienie wierszy macierzy podwajania, tj. wpisanie współczynnika 1 do wszystkich wierszy odpowiadających kopiom i-tego wierzchołka siatki danej. Wiersze macierzy D odpowiadające kopiom wierzchołka są kolejne. W linii 10 w tablicy r reprezentacji macierzy (zobacz podrozdz. G.4) jest wpisywany indeks miejsc w tablicach c i a, w których znajdą się numer kolumny i współczynnik, zapamiętywane tam w liniach 11 i 12. W linii 18 jeden wątek szadera „zakańcza" ciąg liczb w tablicy r, wpisując tam całkowitą liczbę niezerowych współczynników macierzy, równą liczbie jej wierszy.

Listing 31.38. Procedura tworzenia wierszy macierzy podwajania

GLSL

```
1  void DCopyVC ( uint i )
2  {
3    int deg, p, j, k;
4
5    deg = imv(i) >> DEGSHIFT;
6    if ( (imv(i) & TAGMASK) != 0 )
7      deg += 2;
8    p = vcn(i);
9    for ( j = 0; j < deg; j++ ) {
10     rd(p+j) = p+j;
11     cd(p+j) = i;
12     outa.a[p+j] = 1.0;
13     omv(p+j) = 4 << DEGSHIFT;
14   }
15   if ( (imv(i) & TAGMASK) != 0 )
16     omv(p) = omv(p+deg-1) = 2 << DEGSHIFT;
17   if ( i == 0 )
18     rd(outnv) = outnv;
19 } /*DCopyVC*/
```

Zmienione procedury szadera związane z uśrednianiem są pokazane na listingu 31.39. Ostatni etap uśredniania zamienił się w dwa etapy, w których są wykonywane procedury Average0 i Average1, a między nimi jest jeszcze jedno obliczenie sum prefiksowych.

Listing 31.39. Procedury znajdowania macierzy uśredniania

GLSL

```
1: #define Average0(i) fvd(i) = fvnum(i) * (imfac(i) >> DEGSHIFT);
2:
3: void Average1 ( uint i )  /* etap 40 */
4: {
5:   int   r, n, d, j, k, l, m, t;
6:   float id;
7:
8:   if ( (r = fvnum(i)) > 0 ) {
9:     n = nvi(i);
10:    d = imfac(i) >> DEGSHIFT;
11:    j = imfac(i) & FHEMASK;
12:    id = 1.0/float(d);
13:    l = n == 0 ? 0 : fvd(i-1);
14:    for ( k = t = 0;  k < r;  k++ ) {
15:      rd(n+k) = l + t;
16:      for ( m = 0;  m < d;  m++, t++ ) {
17:        cd(l+t) = imhe(imfhei(j+m)).V0;
18:        outa.a[l+t] = id;
19:      }
20:    }
21:  }
22:  if ( i == 0 )
23:    rd(outnv) = fvd(infac-1);
24: } /*Average1*/
```

Procedura (makro) `Average0` wykonywana w etapie 39 ma za zadanie utworzyć ciąg liczb niezerowych współczynników w wierszach macierzy uśredniania odpowiadających poszczególnym ścianom siatki, przy czym jednej ścianie może odpowiadać więcej niż jeden wiersz (czyli więcej niż jeden wierzchołek siatki wynikowej). Ciąg ten jest wpisywany do tablicy `fvd` w buforze roboczym. Tablica ta była potrzebna we wcześniejszych etapach, a teraz jest do dyspozycji i ma wystarczającą długość. Po obliczeniu sum prefiksowych tego ciągu jest wykonywany etap 40, czyli procedura `Average1`. W linii 13 zmienne `l` i `n` otrzymują wartość indeksu miejsc w tablicach `a` i `c` reprezentacji macierzy, w których ma się znaleźć pierwszy niezerowy współczynnik w pierwszym wierszu odpowiadającym danej ścianie. Zewnętrzna pętla (w liniach 14–20) przebiega przez te wiersze. W linii 15 jest wypełniana tablica `r`, po czym w pętli wewnętrznej w tablicy `c` są zapisywane numery kolumn, a do tablicy `a` jest wpisywana liczba $1/s$, obliczona w linii 12. Jeden wątek szadera, w linii 23, zakańcza ciąg liczb w tablicy `r`, wpisując tam ostatni element ciągu sum prefiksowych, który jest liczbą niezerowych współczynników macierzy uśredniania.

Procedura `main` szadera wymaga tylko zamienienia linii 50 na listingu 31.12 na dwie linie:

```
case 39: Average0 ( i );          break;
case 40: Average1 ( i );          break;
```

Procedura kompilacji i łączenia programu z szaderem opisanym wyżej jest prawie identyczna z tą na listingu 31.10; ma tylko nową nazwę `LoadMeshRefinementMatrixProgram` i napis `"mdm.comp.glsl"`, który jest nazwą pliku źródłowego szadera.

Zobaczmy teraz zmiany procedur podwajania i uśredniania w C (listing 31.40) mających znaleźć odpowiednie macierze. Poza zmienionymi nazwami procedury mają dodatkowy parametr wskazujący opakowanie macierzy, którą mają znaleźć. Procedury te korzystają z tych samych procedur `ExecStage`, `PrefixSum` i `SumUp`. Instrukcje przywiązujące bufory ze współrzędnymi wierzchołków zostały usunięte. Zamiast tego procedura `GPUmeshDoublingMatrix` tworzy dwa bufory, w których znajdą się tablice `r`, `c` i `a` macierzy podwajania. W liniach 12 i 15 są ustalane wielkości tych buforów. Instrukcje uruchamiające wszystkie etapy podwajania są identyczne z instrukcjami w procedurze na listingu 31.13 (inna jest jedynie procedura `DCopyVC` wykonywana w etapie 5). Instrukcje w liniach 17–19 wypełniają wskazane przez dodatkowy parametr opakowanie macierzy podwajania.

Listing 31.40. Procedury znajdowania macierzy podwajania i uśredniania

C

```
1: char GPUmeshDoublingMatrix ( GPUmesh *inmesh, GPUmesh *outmesh,
2:                              MeshRefineMatrix *mm )
3: {
4:   int    inv, inhe, infac, invb, inei, onv, onhe, onfac, fvf, maxonv, fvhe;
5:   GLint  bufsize;
6:   GLuint auxbuf = 0, drc = 0, da = 0;
7:
8:   glUseProgram ( progid[1] );
9:   .... /* linie 8-11 i 13-34 z listingu 31.13 bez zmian */
10:  glGenBuffers ( 1, &drc );
11:  glBindBufferBase ( SSB, 3, drc );
12:  glBufferData ( SSB, (onv+onv+1)*sizeof(GLint), NULL, GL_DYNAMIC_DRAW );
13:  glGenBuffers ( 1, &da );
14:  glBindBufferBase ( SSB, 6, da );
15:  glBufferData ( SSB, onv*sizeof(GLfloat), NULL, GL_DYNAMIC_DRAW );
16:  .... /* linie 36-72 z listingu 31.13 bez zmian */
17:  mm->cm = inmesh;  mm->fm = outmesh;
18:  mm->mat.m = mm->mat.nnz = outmesh->nv;  mm->mat.n = inmesh->nv;
19:  mm->mat.buf[0] = drc;  mm->mat.buf[1] = da;
20:  ExitIfGLError ( "GPUmeshDoublingMatrix" );
21:  return true;
22:
23: failure:
24:  glDeleteBuffers ( 1, &auxbuf );
25:  glDeleteBuffers ( 1, &drc );
26:  glDeleteBuffers ( 1, &da );
27:  memset ( mm, 0, sizeof(MeshRefineMatrix) );
28:  glUseProgram ( 0 );
29:  return false;
30: } /*GPUmeshDoublingMatrix*/
```

```
31:
32: char GPUmeshAveragingMatrix ( GPUmesh *inmesh, GPUmesh *outmesh,
33:                               MeshRefineMatrix *mm )
34: {
35:   int    inv, inhe, infac, invb, onv, onhe, onfac, nnz;
36:   GLint  bufsize, bs;
37:   GLuint auxbuf = 0, arc = 0, aa = 0;
38:
39:   glUseProgram ( progid[1] );
40:   .... /* linie 8-11, 13-54 i 56-65 z listingu 31.25 bez zmian */
41:   ExecStage ( uvofs, 39, infac );  /* Average0 */
42:   PrefixSum ( uvofs, 2*infac+inhe+inv, infac );
43:   glBindBuffer ( SSB, auxbuf );
44:   glGetBufferSubData ( SSB, (3*infac+inhe+inv-1)*sizeof(GLuint),
45:                        sizeof(GLuint), &nnz );
46:   glGenBuffers ( 1, &arc );
47:   glBindBufferBase ( SSB, 3, arc );
48:   glBufferData ( SSB, (onv+nnz+1)*sizeof(GLuint), NULL, GL_DYNAMIC_DRAW );
49:   glGenBuffers ( 1, &aa );
50:   glBindBufferBase ( SSB, 6, aa );
51:   glBufferData ( SSB, nnz*sizeof(GLfloat), NULL, GL_DYNAMIC_DRAW );
52:   ExecStage ( uvofs, 40, infac );  /* Average1 */
53:   glUseProgram ( 0 );
54:   glDeleteBuffers ( 1, &auxbuf );
55:   ExitIfGLError ( "GPUmeshAveragingMatrix" );
56:   mm->cm = inmesh;  mm->fm = outmesh;
57:   mm->mat.m = outmesh->nv;  mm->mat.n = inmesh->nv;  mm->mat.nnz = nnz;
58:   mm->mat.buf[0] = arc;  mm->mat.buf[1] = aa;
59:   return true;
60:
61: failure:
62:   glDeleteBuffers ( 1, &auxbuf );
63:   glDeleteBuffers ( 1, &arc );
64:   glDeleteBuffers ( 1, &aa );
65:   memset ( mm, 0, sizeof(MeshRefineMatrix) );
66:   glUseProgram ( 0 );
67:   return false;
68: } /*GPUMeshAveragingMatrix*/
```

Procedura `GPUmeshAveragingMatrix` w etapie 39 powoduje wywołanie procedury `Average0`, a następnie oblicza sumy prefiksowe znalezionego w tym etapie ciągu. Ostatnia suma jest odczytywana z bufora w liniach 44–45; jest to liczba niezerowych współczynników konstruowanej macierzy uśredniania. Mając ją, można utworzyć bufory dla macierzy `r`, `c` i `a`, co jest robione w liniach 46–51. W linii 52 jest uruchamiany etap 40 obliczeń, w którym procedura `Average1` z listingu 31.39 wypełnia te tablice. Opakowanie macierzy wskazywane przez trzeci parametr jest wypełniane w liniach 56–58.

Listing 31.41. Procedura znajdowania macierzy zagęszczania

C

```
char GPUmeshRefinementMatrix ( int n, GPUmesh *inmesh, GPUmesh *outmesh,
                               MeshRefineMatrix *mm )
{
  GPUmesh          mmesh, *am, *bm, *cm;
  MeshRefineMatrix md, ma;
  int              i;

  if ( n < 1 )
    return false;
  memset ( &mmesh, 0, sizeof(GPUmesh) );
  if ( n & 0x01 ) { am = &mmesh;  bm = outmesh; }
             else { am = outmesh;  bm = &mmesh; }
  if ( !GPUmeshDoublingMatrix ( inmesh, am, &md ) )
    goto failure;
  for ( i = 0; i < n; i++ ) {
    if ( !GPUmeshAveragingMatrix ( am, bm, &ma ) )
      goto failure;
    if ( !GPUMultSparseMatricesf ( &mm->mat, &ma.mat, &md.mat );
      goto failure;
    glDeleteBuffers ( 4, am->mbuf );
    memset ( am->mbuf, 0, 4*sizeof(GLuint) );
    mm->m = ma.m;  mm->n = md.n;
    cm = am;  am = bm;  bm = cm;
    GPUDeleteMeshRefineMatrix ( &md );
    GPUDeleteMeshRefineMatrix ( &ma );
    md = *mm;
  }
  mm->cm = inmesh;  mm->fm = outmesh;  mm->mat.lmax = 0;
  return true;

failure:
  glDeleteBuffers ( 4, am->mbuf );
  glDeleteBuffers ( 4, bm->mbuf );
  return false;
} /*GPUmeshRefinementMatrix*/

void GPUDeleteMeshRefineMatrix ( MeshRefineMatrix *mm )
{
  glDeleteBuffers ( 2, mm->mat.buf );
  memset ( mm, 0, sizeof(MeshRefineMatrix) );
} /*GPUDeleteMeshRefineMatrix*/
```

Listing 31.41 przedstawia procedurę znajdującą topologię siatki zagęszczonej i macierz zagęszczania z n krokami uśredniania. Powstała ona przez modyfikację procedury z listingu 31.35. Po znalezieniu macierzy podwajania, w pętli wykonywanej n razy procedura

znajduje kolejną macierz uśredniania, po czym wykonuje mnożenie macierzy rzadkich za pomocą procedury opisanej w p. G.4.3. W liniach 24 i 25 macierze te są kasowane; pozostaje ich iloczyn, który w następnym przebiegu pętli będzie z lewej strony pomnożony przez kolejną macierz uśredniania, a po zakończeniu pętli będzie gotową macierzą zagęszczania. W linii 28 opakowanie macierzy jest uzupełniane o brakujące informacje.

31.11.3. Obliczanie współrzędnych wierzchołków

Mając reprezentacje (topologie) siatek oryginalnej i zagęszczonej i macierz zagęszczania oraz ustalone (bieżące) położenia wierzchołków siatki oryginalnej, wystarczy wywołać procedurę z listingu 31.42, która obliczy współrzędne wierzchołków siatki zagęszczonej, mnożąc macierz zagęszczania przez wektor położeń wierzchołków. Procedura ta zawiera wywołanie procedury pokazanej na listingu G.15 z parametrami wziętymi z opakowania macierzy zagęszczania przygotowanego przez procedurę `GPUmeshRefinementMatrix`.

Listing 31.42. Procedura obliczania współrzędnych wierzchołków zagęszczonej siatki

C

```
1: void GPUMatrixRefineMesh ( MeshRefineMatrix *mm )
2: {
3:   GPUMultSparseMatrixVectorf ( mm->fm->VCBUF, &mm->mat,
4:                                 mm->cm->nsattr, mm->cm->VCBUF );
5: } /*GPUMatrixRefineMesh*/
```

Co to wszystko daje? Opisany tu zestaw procedur wbudowałem w aplikację 3D (zobacz podrozdz. 36.7), co umożliwiło ich przetestowanie i pomiary czasu. Przetwarzana przez tę aplikację siatka (opisana w podrozdz. 32.1), która ma 144 wierzchołki, 564 półkrawędzie i 140 ścian, była poddawana czterem zagęszczeniom z trzema krokami uśredniania.

Tabela 31.1. Wyniki zagęszczania przykładowej siatki

n_v	n_h	n_f	$m \times n$	N	$l_{\max}$	B
144	564	140	566×144	3600	14	33336
566	2256	564	2258×566	14102	15	121888
2258	9024	2256	9026×2258	56402	15	487324
9026	36096	9024	36098×9026	225602	15	1949212
36098	144384	36096				

W wierszach tabeli 31.1 są podane liczby n_v, n_h i n_f wierzchołków, półkrawędzi i ścian siatki danej i siatek otrzymanych przez kolejne zagęszczenia. Obok są podane wymiary macierzy zagęszczania, liczby N ich niezerowych współczynników, maksymalne liczby $l_{\max}$ współczynników w wierszach i liczby B bajtów zajmowanych przez reprezentacje tych macierzy, tj. tablice `r`, `c` i `a`. Jak widać, w sumie potrzeba na nie nieco mniej niż 2.5 MB.[16] Łatwo jest

[16] Podane liczby bajtów nie uwzględniają narzutów potrzebnych do działania OpenGL-a. Wymiary bloków rezerwowanych w pamięci GPU są zaokrąglane w górę do wielokrotności liczby zapewniającej szybki dostęp do danych w buforze.

też sprawdzić, że średnia liczba niezerowych współczynników w wierszach wszystkich tych macierzy nie przekracza 6.5. Najmniejsza macierz ma mniej niż 4.4% niezerowych współczynników, a największa ma ich mniej niż 0.07%.

Tabela 31.2. Czasy zagęszczania przykładowej siatki

RTX 3060			GTX 940M		
T_0	T_1	T_2	t_0	t_1	t_2
0.001694	0.000939	0.000052	0.005362	0.020722	0.000082
0.002550	0.006018	0.000093	0.012763	0.038580	0.000288
0.005424	0.013674	0.000272	0.040882	0.166048	0.001127
0.017820	0.046080	0.001014	0.163901	0.697506	0.008372

W tabeli 31.2 są pokazane czasy obliczeń zmierzone[17] na komputerze stacjonarnym i na laptopie, wyposażonych w procesory graficzne o różnych mocach obliczeniowych. Podane w kolejnych wierszach czasy T_0 i t_0 zajęły kolejne zagęszczania siatek przy użyciu szadera wykonującego pełne obliczenie. Znalezienie macierzy zagęszczania trwało na tych komputerach odpowiednio T_1 i t_1 sekund, a T_2 i t_2 to czasy obliczania współrzędnych wierzchołków zagęszczonych siatek za pomocą tych macierzy.

Znajdowanie topologii siatki zagęszczonej i macierzy zagęszczania trwa dłużej niż zagęszczanie, podczas którego znajduje się topologię i oblicza współrzędne wierzchołków. Jest tak dlatego, bo mnożenie macierzy podwajania i uśredniania zabiera więcej czasu niż obliczanie współrzędnych. Ale to nie jest istotne, jeśli macierze są znajdowane tylko raz, na początku działania aplikacji albo podczas jej instalacji przez program, który topologie potrzebnych siatek i macierze zagęszczania znajdzie i zapisze w pliku. Obliczanie współrzędnych za pomocą „gotowych" macierzy zagęszczania jest znacznie szybsze niż pełne zagęszczanie, dzięki czemu płynna animacja siatki odkształcanej i zagęszczanej w czasie rzeczywistym staje się wykonalna nawet na niezbyt potężnym laptopie — może z wyjątkiem siatki o największym stopniu zagęszczenia, której rysowanie (a nie obliczanie położeń wierzchołków) trwa trochę za długo.

31.12. Ćwiczenia

1. Napisz procedurę, która odwraca orientację siatki. Dla każdej półkrawędzi należy zamienić indeksy końca i początku. Dla każdej ściany trzeba odwrócić kolejność indeksów jej półkrawędzi, a dla każdego wierzchołka trzeba utworzyć zupełnie nowy ciąg indeksów półkrawędzi wychodzących z niego.
2. Napisz procedurę, która łączy dwie siatki, tzn. kopiuje do wspólnych tablic jedną z nich i przepisuje wierzchołki, półkrawędzie i ściany drugiej, odpowiednio je przenumerowując.

[17] Ostatnia cyfra znacząca wyników tych pomiarów lub nawet ostatnie dwie cyfry są niepewne.

3. *Napisz procedurę, która wybiera fragment siatki składający się z zaznaczonych wierzchołków, incydentnych z nimi półkrawędzi i składających się z tych półkrawędzi ścian.

Każdą z tych procedur napisz i uruchom w wersji działającej na CPU, a potem zaimplementuj na GPU.

32

Trzecia aplikacja

— Dlaczego nie rysujesz?
— Bo nie mam ołówka.
— To rysuj odręcznie.

Usłyszane przez Annę w szkole na lekcji

Siatki reprezentowane w sposób przedstawiony w poprzednim rozdziale, opisujące rozmaite obiekty, można projektować za pomocą jednego z programów demonstracyjnych pisanego przeze mnie pakietu BSTools [42]. Prace nad nim zacząłem w roku 2003, a w latach 2005 i 2019 aktualne wersje pakietu dołączyłem do drugiego i trzeciego wydania mojej książki [41]. Obecnie pakiet składa się z kilkunastu bibliotek zawierających w sumie kilka tysięcy rozmaitych procedur (napisanych w języku C) i garści programów demonstracyjnych. Głównym jego przeznaczeniem jest przetwarzanie krzywych i powierzchni Béziera i B-sklejanych oraz siatek w aplikacjach graficznych.

Program demonstracyjny `pozwalaj` jest aplikacją starego OpenGL-a (wersji 2.1), działającą w środowisku Linux/X Window. Umożliwia on m.in. modelowanie siatek, które zapisuje w plikach tekstowych łatwych do przetworzenia na odpowiednie fragmenty programów w C. Programu tego użyłem do wymodelowania siatki reprezentującej powierzchnię wyglądającą z grubsza jak ludzka dłoń. Aplikacja opisana w tym rozdziale wyświetla tę siatkę i powierzchnie otrzymane przez jej zagęszczanie. Ten model dłoni nie jest bardzo dokładny (zaprojektowałem go odręcznie), ale nic nie stoi na przeszkodzie, by go udoskonalić w stopniu wystarczającym nawet na potrzeby pianistyki, kryminalistyki i chiromancji.

32.1. Model dłoni

Listing 32.1 przedstawia deklarację tablic zawierających reprezentację siatki będącej modelem dłoni. Pominąłem prawie całą zawartość tych tablic, bo zajmuje ona ponad 300 linii kodu w C, ale Czytelnik znajdzie ją w odpowiednim pliku źródłowym trzeciej aplikacji. Procedura `EnterPalmToGPU` odpowiada za przesłanie tych danych do buforów w pamięci GPU, za pomocą procedury `CPUmeshToGPU` z listingu 31.8.

Listing 32.1. Siatka opisująca dłoń

C

```
#define PALM_NV   144
#define PALM_NHE  564
#define PALM_NFAC 140

static BSMvertex mv[PALM_NV] = {{4,0},....,{5,559}};
static int mvhei[PALM_NHE] = {0,428,....,139,4};
static float vc[PALM_NV][3] =
   {{-0.64428,-0.01967,0.10625},....,{0.16963,0.01677,0.10533}};
static BSMhalfedge mhe[PALM_NHE] = {{0,1,0,427},....,{143,4,134,562}};
static BSMfacet mfac[PALM_NFAC] = {{7,0},....,{4,560}};
static int mfhei[PALM_NHE] = {0,1,....,561,562};

GPUmesh *EnterPalmToGPU ( void )
{
  CPUmesh cpalm;
  GPUmesh *gpalm;

  if ( !(gpalm = malloc ( sizeof(GPUmesh) )) )
    return NULL;
  memset ( gpalm, 0, sizeof(GPUmesh) );
  cpalm.nsattr = cpalm.pdim = 3;  cpalm.pofs = 0;  cpalm.nvofs = -1;
  cpalm.nv = PALM_NV;   cpalm.nhe = PALM_NHE;  cpalm.nfac = PALM_NFAC;
  cpalm.mv = mv;        cpalm.mhe = mhe;       cpalm.mfac = mfac;
  cpalm.mvhei = mvhei;  cpalm.mfhei = mfhei;   cpalm.vc = &vc[0][0];
  if ( CPUmeshToGPU ( &cpalm, gpalm ) )
    return gpalm;
  else {
    free ( gpalm );
    return NULL;
  }
} /*EnterPalmToGPU*/
```

32.2. Rysowanie siatki

W pierwszej wersji trzeciej aplikacji użyjemy dwóch prostych programów do rysowania siatek: pierwszy z nich rysuje krawędzie siatki, a drugi trójkąty otrzymane z podziału ścian. Aby skupić się na sednie rzeczy, nie wprowadzimy żadnych wyrafinowanych modeli oświetlenia (użyjemy tylko modelu Lamberta) ani efektów takich jak cienie. Zmiennej typu pokazanego na listingu 32.2 użyjemy do opakowania obu programów rysujących. Przechowamy w nim tylko identyfikatory programów.

Przed przystąpieniem do rysowania bufory z reprezentacją siatki muszą być przywiązane do odpowiednich punktów dowiązania w celu `GL_SHADER_STORAGE_BUFFER`; deklaracje odpowiednich bloków w treści szaderów rysujących są takie jak na listingu 31.4.

Listing 32.2. Opakowanie programów rysujących krawędzie i ściany siatki

C

```
typedef struct {
    GLuint progid[2];
  } MeshRenderPrograms;
```

Współrzędne otrzymane od etapu pobierania wierzchołków zostaną zignorowane; dane te są potrzebne do uruchomienia potoku przetwarzania grafiki, ale zadanie *wybrania* właściwego wierzchołka wykona szader na podstawie numeru instancji rysowanego prymitywu i danych opisujących siatkę. W pewnym sensie szadery w tych programach przejmują rolę etapu pobierania wierzchołków wprowadzającego do potoku atrybuty wierzchołków na podstawie tablicy indeksów przywiązanej do celu `GL_ELEMENT_ARRAY_BUFFER`, który to mechanizm jest niewystarczający w zastosowaniu do siatek reprezentowanych w sposób opisany w rozdziale 31.[1] Zatem procedura `main` szadera wierzchołków w obu programach zawiera instrukcje będące treścią makrodefinicji `FetchVertex` przedstawionej na listingu 32.3. Jej parametrem jest indeks pierwszej współrzędnej położenia w tablicy atrybutów wierzchołków. Instrukcje te przypisują zmiennej `gl_Position` współrzędne jednorodne wierzchołka o numerze obliczonym przez procedurę `main`, nadając współrzędnym z i w wartości domyślne 0 i 1, jeśli w tablicy `mvc.mvc` współrzędne te są nieobecne.

Listing 32.3. Makrodefinicja `FetchVertex`

GLSL

```
....          /* bloki magazynowe jak na listingu 31.4 */

#define FetchVertex(I) \
  switch ( pdim ) { \
case 2: \
    gl_Position = vec4 ( mvc.vc[I], mvc.vc[(I)+1], 0.0, 1.0 ); \
    break; \
case 3: \
    gl_Position = vec4 ( mvc.vc[I], mvc.vc[(I)+1], mvc.vc[(I)+2], 1.0 ); \
    break; \
default: \
    gl_Position = vec4 ( mvc.vc[I], mvc.vc[(I)+1], \
                         mvc.vc[(I)+2], mvc.vc[(I)+3] ); \
    break; \
  }
```

Zadaniem pierwszego programu rysowania siatek, składającego się z szaderów pokazanych na listingach 32.4–32.6, jest narysowanie krawędzi siatki w ustalonym kolorze. Program będzie wywołany przez procedurę `glDrawArraysInstanced`, która ma narysować jeden odcinek w liczbie instancji równej liczbie półkrawędzi siatki. Każda krawędź wewnętrzna

[1] Można na podstawie reprezentacji siatki utworzyć odpowiednie tablice z indeksami (za pomocą dodatkowego szadera obliczeniowego), aby następnie rysować siatki za pomocą procedury `glDrawElements`, ale chyba szkoda miejsca w pamięci GPU na dodatkowe tablice. Niemniej, napisanie takiego szadera byłoby dobrym ćwiczeniem z programowania w języku GLSL.

jest reprezentowana przez dwie półkrawędzie, a chcielibyśmy, aby była narysowana tylko raz. Dlatego rysowany będzie odcinek odpowiadający tylko jednej półkrawędzi z pary; zadba o to szader geometrii.

Listing 32.4. Szader wierzchołków pierwszego programu rysowania siatek

GLSL

```
#version 450 core

layout(location=0) out int instance;
layout(location=1) out vec3 colour;

.... /* tu treść listingu 32.3 */

#define V0 x   /* tak samo jak na listingu 31.9 */
#define V1 y

void main ( void )
{
  int  i, j, k;
  vec4 vp;

  instance = i = gl_InstanceID;
  j = gl_VertexID == 0 ? mhe.mhe[i].V0 : mhe.mhe[i].V1;
  k = nsattr*j + pofs;
  FetchVertex ( k );
  colour = Colour;  /* kolor z bloku meshsurf */
} /*main*/
```

Szader wierzchołków wyprowadza (w zmiennej `instance`) numer instancji, który jest numerem półkrawędzi. Na podstawie wartości zmiennej `gl_VertexID` szader określa, który wierzchołek — początek, czy koniec półkrawędzi — ma przekazać dalej i pobiera jego współrzędne z tablicy za pomocą makra `FetchVertex`. Przejściem do układu współrzędnych kostki standardowej zajmie się szader geometrii. Wartość przypisana w linii 16 zmiennej `i` jest numerem półkrawędzi do narysowania. W linii 17 jest znajdowany numer wierzchołka końcowego tej półkrawędzi, a w linii 18 jest obliczany indeks pierwszej współrzędnej tego wierzchołka.

Listing 32.5. Szader geometrii pierwszego programu rysowania siatek

GLSL

```
#version 450 core

layout(lines) in;
layout(line_strip,max_vertices=2) out;

layout(location=0) in int instance[];
layout(location=1) in vec3 Colour[];

```

```
9: layout(location=0) out vec3 colour;
10:
11: layout(std430,binding=1) buffer meshhe { ivec4 mhe[]; } mhe;
12:
13: #define OTHE w  /* tak samo jak na listingu 31.9 */
14:
15: uniform TransBlock {
16:     mat4 mm, mmti, vm, pm, vpm;
17:     vec4 eyepos;
18:   } trb;
19:
20: void main ( void )
21: {
22:   int i;
23:
24:   i = instance[0];
25:   if ( mhe.mhe[i].OTHE > i ) {
26:     for ( i = 0; i < 2; i++ ) {
27:       gl_Position = trb.vpm * (trb.mm * gl_in[i].gl_Position);
28:       colour = Colour[i];
29:       EmitVertex ();
30:     }
31:     EndPrimitive ();
32:   }
33: } /*main*/
```

Szader geometrii jest przedstawiony na listingu 32.5; sposób przekształcania współrzędnych wierzchołków (obu końców krawędzi) jest taki jak w wielu szaderach opisanych wcześniej. Jedyny zatem nowy szczegół to warunek sprawdzany w linii 25: jeśli numer drugiej półkrawędzi z pary jest mniejszy niż numer półkrawędzi przetwarzanej przez daną instancję szadera, to szader nie wyprowadzi odcinka do dalszego przetwarzania. W ten sposób są pomijane także nieistniejące (mające numer -1) półkrawędzie z „par" reprezentujących krawędzie brzegowe siatki.

Listing 32.6. Szader fragmentów pierwszego programu rysowania siatek

GLSL

```
1: #version 450 core
2:
3: layout(location=0) in vec3 colour;
4:
5: out vec4 out_Colour;
6:
7: void main ( void )
8: {
9:   out_Colour = vec4 ( colour, 1.0 );
10: } /*main*/
```

Listing 32.7. Procedura rysowania krawędzi siatki

C

```
void SetMeshColour ( GPUmesh *gmesh, GLfloat Colour[3] )
{
  glBindBuffer ( SSB, gmesh->MSBUF );
  glBufferSubData ( SSB, mbofs[8], 3*sizeof(GLfloat), Colour );
  ExitIfGLError ( "SetMeshColour" );
} /*SetMeshColour*/

void DrawMeshEdges ( MeshRenderPrograms *prog,
                     GPUmesh *mesh, GLfloat colour[4] )
{
  int i;

  glUseProgram ( prog->progid[0] );
  for ( i = 0; i < 4; i++ )
    glBindBufferBase ( SSB, i, mesh->mbuf[i] );
  SetMeshColour ( mesh, colour );
  glBindVertexArray ( empty_vao );
  glDrawArraysInstanced ( GL_LINES, 0, 2, mesh->nhe );
  glBindVertexArray ( 0 );
  ExitIfGLError ( "DrawMeshEdges" );
} /*DrawMeshEdges*/
```

Szader fragmentów pierwszego programu (listing 32.6) jest tak prosty, że nie ma o czym pisać. Procedura na CPU rysująca krawędzie siatki za pomocą programu składającego się z szaderów opisanych wyżej jest pokazana na listingu 32.7. Procedura ta wybiera program, przywiązuje bufory z reprezentacją siatki, przypisuje polu `Colour` bloku `meshsurf` reprezentującą kolor wartość parametru `Colour`, przywiązuje pusty obiekt tablicy wierzchołków i rysuje jeden odcinek w odpowiedniej liczbie egzemplarzy (tj. instancji).

Listing 32.8. Szader wierzchołków drugiego programu rysowania siatek

GLSL

```
#version 450 core

layout(location=0) out vec3 colour;

.... /* tu treść listingu 32.3 */

#define FHEMASK  0x01FFFFFF            /* tak samo jak na listingu 31.3 */
#define mfac(I)  mvf.mvf[nv+(I)]
#define mfhei(I) mvf.mvf[nv+nfac+nhe+(I)]
#define V0 x

void main ( void )
{
  int i;

```

```
  i = mhe.mhe[mfhei((mfac(gl_InstanceID) & FHEMASK) + gl_VertexID)].V0;
  i = nsattr*i + pofs;
  FetchVertex ( i );
  colour = Colour;
} /*main*/
```

Szadery pokazane na listingach 32.8 i 32.9 są częściami drugiego programu, którego zadaniem jest narysowanie ścian siatki. Program działa przy założeniu, że wszystkie ściany siatki są czworokątne, co ma miejsce, jeśli siatka została otrzymana jako wynik zagęszczania z *nieparzystą* liczbą kroków uśredniania. Dla każdej ściany zostaną narysowane dwa trójkąty. Dokładniej, program jest wywoływany przez procedurę `glDrawArraysInstanced`, której parametry opisują wachlarz złożony z dwóch trójkątów. Liczba instancji szadera jest równa liczbie ścian siatki. Zadaniem szadera wierzchołków jest wyprowadzenie wierzchołka o numerze ustalonym na podstawie numeru instancji (czyli numeru ściany) oraz numeru *i* wierzchołka w rysowanym wachlarzu. Numer ten, podany w zmiennej `gl_VertexID` (od 0 do 3), jest początkiem *i*-tej półkrawędzi ściany.

Na listingu 32.9 jest pokazany szader geometrii, którego wejście stanowi tablica z trzema wierzchołkami trójkąta. Współrzędne tych wierzchołków są podane w układzie modelu, zatem szader geometrii dokonuje przejścia do układu świata — w linii 23 szader oblicza współrzędne jednorodne wierzchołka w układzie świata, a w linii 24 oblicza współrzędne kartezjańskie. Wektory współrzędnych kartezjańskich są używane (w liniach 26, 27) do obliczenia wektora normalnego płaszczyzny trójkąta oraz wyprowadzane (w bloku wyjściowym `FVertex`) razem z wektorem normalnym do dalszych obliczeń oświetlenia przez szader fragmentów.

Listing 32.9. Szader geometrii drugiego programu rysowania siatek

GLSL

```
#version 450 core

layout(triangles) in;
layout(triangle_strip,max_vertices=3) out;

layout(location=0) in vec3 Colour[];

out FVertex {
    vec3 Position;
    vec3 Colour;
    flat vec3 Normal;
  } Out;

uniform TransBlock {
    mat4 mm, mmti, vm, pm, vpm;
    vec4 eyepos;
  } trb;

```

```
19: void main ( void )
20: {
21:   int  i;
22:   vec4 p[3];
23:   vec3 q[3], v1, v2, nv;
24:
25:   for ( i = 0; i < 3; i++ ) {
26:     p[i] = trb.mm * gl_in[i].gl_Position;
27:     q[i] = p[i].xyz/p[i].w;
28:   }
29:   v1 = q[1] - q[0];  v2 = q[2] - q[0];
30:   nv = normalize ( cross ( v1, v2 ) );
31:   for ( i = 0; i < 3; i++ ) {
32:     gl_Position = trb.vpm * p[i];
33:     Out.Colour = Colour[i];
34:     Out.Position = q[i];
35:     Out.Normal = i == 0 ? nv : vec3(0.0);
36:     EmitVertex ();
37:   }
38:   EndPrimitive ();
39: } /*main*/
```

Szader fragmentów drugiego programu jest prawie identyczny z szaderem przedstawionym na listingu 10.4, jedyne zmiany to dodanie kwalifikatora `flat` pola `Normal` w bloku `In` oraz dodanie pola `mmti` do bloku zmiennych jednolitych `TransBlock`. Na listingu 32.10 jest pokazana procedura rysowania ścian siatki za pomocą opisanego wyżej programu szaderów. Procedura ta deklaruje, że zmienna wejściowa `Normal` z kwalifikatorem `flat` jest podana z pierwszym wierzchołkiem trójkąta przekazanego przez szader geometrii, a potem przywiązuje bufory z reprezentacją siatki, wybiera program szaderów, przypisuje kolor ścian, przywiązuje pusty obiekt tablicy wierzchołków i uruchamia potok przetwarzania grafiki.

Listing 32.10. Procedura rysowania ścian siatki

C

```
1: void DrawMeshFacets ( MeshRenderPrograms *prog,
2:                       GPUmesh *mesh, GLfloat colour[3] )
3: {
4:   int i;
5:
6:   glProvokingVertex ( GL_FIRST_VERTEX_CONVENTION );
7:   for ( i = 0; i < 4; i++ )
8:     glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, i, mesh->mbuf[i] );
9:   SetMeshColour ( mesh, colour );
10:  glUseProgram ( prog->progid[1] );
11:  glBindVertexArray ( empty_vao );
12:  glDrawArraysInstanced ( GL_TRIANGLE_FAN, 0, 4, mesh->nfac );
13:  glBindVertexArray ( 0 );
```

```
14:   ExitIfGLError ( "DrawMeshFacets" );
15: } /*DrawMeshFacets*/
```

Na początku działania aplikacji należy opisane wyżej programy przygotować do działania, wywołując procedurę `LoadMeshRenderingPrograms` z listingu 32.11 z parametrem — wskaźnikiem zadeklarowanej w aplikacji zmiennej, która jest opakowaniem dla programów rysujących. Po lekturze wcześniejszych rozdziałów działanie większości instrukcji w tej procedurze jest oczywiste.

Listing 32.11. Przygotowanie i likwidacja programów rysowania siatek

C

```
1: void LoadMeshRenderingPrograms ( MeshRenderPrograms *prog )
2: {
3:   static const GLchar *filename[] =
4:     { "app3_0.vert.glsl", "app3_0.geom.glsl", "app3_0.frag.glsl",
5:       "app3_1.vert.glsl", "app3_1.geom.glsl", "app3_1.frag.glsl"};
6:   static const GLuint shtype[] =
7:     { GL_VERTEX_SHADER, GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER,
8:       GL_VERTEX_SHADER, GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER };
9:   GLuint shader_id[6];
10:  int    i;
11:
12:  for ( i = 0; i < 6; i++ )
13:    shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
14:  prog->progid[0] = LinkShaderProgram ( 3, &shader_id[0], "0" );
15:  prog->progid[1] = LinkShaderProgram ( 3, &shader_id[3], "1" );
16:  GetAccessToMeshSurfBlock ( prog->progid[1] );
17:  GetAccessToTransBlockUniform ( prog->progid[0] );
18:  AttachUniformTransBlockToBP ( prog->progid[1] );
19:  GetAccessToLightBlockUniform ( prog->progid[1] );
20:  for ( i = 0; i < 6; i++ )
21:    glDeleteShader ( shader_id[i] );
22:  ExitIfGLError ( "LoadMeshRenderingPrograms" );
23: } /*LoadMeshRenderingPrograms*/
24:
25: void DeleteMeshRenderingPrograms ( MeshRenderPrograms *prog )
26: {
27:  int i;
28:
29:  glUseProgram ( 0 );
30:  for ( i = 0; i < 2; i++ )
31:    glDeleteProgram ( prog->progid[i] );
32:  ExitIfGLError ( "DeleteMeshRenderingPrograms" );
33: } /*DeleteMeshRenderingPrograms*/
```

Siatki będące wynikiem zagęszczania z parzystą liczbą n kroków uśredniania mogą mieć ściany o innej niż 4 liczbie krawędzi i procedura `DrawMeshFacets` nie może wykonać po-

prawnego obrazu takiej siatki. Najprostszy sposób poradzenia sobie z tym problemem polega na wykonaniu jeszcze jednego kroku uśredniania, którego wynikiem będzie siatka z wszystkimi ścianami czworokątnymi.

Uwaga: Jeśli w siatce występują krawędzie brzegowe (w siatce dłoni takich krawędzi nie ma), to lepiej jest pominąć na obrazie niektóre ściany. Mianowicie dla nieparzystego n należy pominąć $(n-1)/2$ „warstw" ścian przyległych do brzegu. Na przykład dla $n = 3$ pominięte powinny być ściany, które mają pewien wierzchołek brzegowy, a dla $n = 5$ trzeba dodatkowo pominąć ściany mające wspólny wierzchołek ze ścianą mającą wierzchołek brzegowy. Aby narysować siatkę otrzymaną przez zagęszczanie z parzystą liczbą n, powinniśmy wykonać wspomniane wyżej dodatkowe uśrednianie, a potem odrzucić $n/2 - 1$ „warstw" ścian przy brzegu siatki. Reguła ta wynika stąd, że powierzchnia graniczna (tj. granica nieskończonego ciągu siatek otrzymanych przez zagęszczanie) jest lepiej przybliżana przez ściany pozostałe.

32.3. Część graficzna trzeciej aplikacji

Podobnie jak w aplikacji pierwszej i drugiej, część okienkowa nie wywołuje bezpośrednio żadnych procedur OpenGL-a (tylko procedury z biblioteki GLX w celu utworzenia i zlikwidowania kontekstu oraz obsługi podwójnego buforowania). Przetwarzanie komunikatów wejściowych powoduje wywoływanie (niewielu) procedur, które stanowią interfejs części graficznej; ta z kolei jest całkowicie niezależna od środowiska, dzięki czemu mogłaby stać się na przykład częścią aplikacji systemu Windows bez żadnych zmian. Utworzone i obsługiwane przez część okienkową wihajstry (w tej wersji aplikacji tylko przełączniki) służą do wydawania poleceń dla części graficznej. Dane części graficznej są niewidoczne dla części okienkowej, *z wyjątkiem* zmiennych, które muszą być widoczne dla obu części aplikacji, na przykład tych, którym wihajstry w menu mają przypisywać wartości[2]. Zmienne te są polami zmiennej typu `AppWidgets`.

Plik nagłówkowy interfejsu między częściami graficzną a okienkową jest przedstawiony na listingu 32.12; makrodefinicje w liniach `3–9` są identyfikatorami wihajstrów (wihajstra obrazu, guzika zatrzymania i przełączników), przy czym identyfikatory przełączników są jednocześnie identyfikatorami poleceń dla części graficznej wydawanych przez te wihajstry. Procedura `InitMyWorld`, wywoływana po utworzeniu okien, przekazuje wskaźnik wspomnianej zmiennej typu `AppWidgets`; wihajstry są tworzone *po powrocie* z tej procedury i są „przyczepiane" do pól tej zmiennej.

[2]Zmiana stanu wihajstra może spowodować zmianę stanu innych wihajstrów, o czym decyduje część graficzna. Dlatego budowa struktury `AppWidgets` i jej pola są widoczne dla obu części aplikacji. Oczywiście, można uczynić zmienne każdej części aplikacji niewidocznymi dla drugiej części, odpowiednio rozbudowując procedury interfejsu (w tym przypadku procedura `ProcessWorldRequest` mogłaby wykonywać polecenie ustawiania przełączników w części okienkowej), ale kod w C byłby sporo dłuższy, a to jest książka o OpenGL-u i GLSL-u.

Listing 32.12. Interfejs części okienkowej i części graficznej trzeciej aplikacji

C

```
#define NPALMMESHES   4

#define GLWIN_ID_VIEW 1
#define BTN_ID_EXIT   2
#define SW_ID_MESH0   3
#define SW_ID_MESH1   4
#define SW_ID_MESH2   5
#define SW_ID_MESH3   6
#define SW_ID_MESH4   7

#define WMSG_ANIMATION_ON  1
#define WMSG_ANIMATION_OFF 2

typedef struct {
    char sw[NPALMMESHES+1];
    char animation;
  } AppWidgets;

AppWidgets *InitMyWorld ( int argc, char *argv[], int width, int height );
void ResizeMyWorld ( int width, int height );
void RedrawMyWorld ( void );
void RotateViewer ( double delta_xi, double delta_eta );
void DeleteMyWorld ( void );
char ProcessCharCommand ( char charcode );
char ProcessSwitchCommand ( int wdg_id );
char MoveOn ( void );

char ProcessWorldRequest ( int msg, void *data, void *reply );
```

Opisana dalej procedura obsługi komunikatów wihajstrów w części okienkowej wywołuje odpowiednie procedury części graficznej; reakcje na zmianę stanu przełączników wykonuje procedura `ProcessSwitchCommand`. Obszar okna z obrazem też jest wihajstrem; wprowadzone w pierwszej i drugiej aplikacji procedury `ResizeMyWorld`, `RedrawMyWorld`, `RotateViewer`, `ProcessCharCommand` i `MoveOn` są wywoływane przez procedury obsługi komunikatów tego wihajstra.

Procedura `ProcessWorldRequest` należy do części okienkowej; jest ona wywoływana z części graficznej w odpowiedzi na polecenia uruchomienia i zatrzymania animacji; identyfikatory tych poleceń są wprowadzone w liniach 11 i 12.

Listing 32.13 przedstawia deklaracje typów strukturalnych opakowujących dane części graficznej aplikacji. Pierwsze pole struktury `AppData` zawiera przełączniki, tj. zmienne, którym wartości nadają wihajstry w części okienkowej. Struktura typu `KLMesh` zawiera tablicę wskaźników do reprezentacji siatek dłoni; pierwsza siatka jest wprowadzana do pamięci GPU przez procedurę z listingu 32.1, a każda następna jest otrzymana z poprzedniej przez zagęsz-

czanie z trzema krokami uśredniania. Oprócz tego są podane kolory, w których mają być rysowane krawędzie i ściany siatki. Deklaracja typu `Camera` jest pokazana na listingu 15.13.

Listing 32.13. Struktury danych części graficznej

C

```
typedef struct {
    GPUmesh *mesh[NPALMMESHES+1];
    GLfloat ecolour[3], fcolour[3];
  } KLMesh;

typedef struct {
    AppWidgets          wdg;
    KLMesh              palm;
    Camera              camera;
    TransBl             trans;
    LightBl             light;
    char                lod, edges;
    float               speed;
    float               model_rot_axis[3];
    double              model_rot_angle;
    MeshRenderPrograms mrprog;
  } AppData;
```

Pola `trans` i `light` są strukturami przechowującymi macierze przekształceń i opisy źródeł światła.

Wartość pola `lod` określa, która siatka ma być wyświetlana — kolejne siatki mają coraz więcej ścian, dzięki czemu coraz lepiej przybliżają gładką powierzchnę, ale coraz więcej czasu zajmuje ich rysowanie. Pole `edges` jest przełącznikiem, którego wartość `true` powoduje wyświetlanie krawędzi (zamiast wypełnionych ścian) zagęszczonej siatki. Pola `speed`, `model_rot_axis` i `model_rot_angle` przechowują prędkość kątową obracającej się siatki, wektor kierunku osi i bieżący kąt tego obrotu. W polu `mrprog` są przechowywane idntyfikatory programów szaderów używanych do rysowania.

Listing 32.14 przedstawia procedurę inicjalizacji danych. W kolejnych instrukcjach zmienna `appdata` jest wypełniana zerami, po czym następuje wywołanie procedur kompilujących i łączących programy szaderów, tworzenie pustego obiektu tablicy wierzchołków, tworzenie buforów z blokami zmiennych jednolitych `TransBlock` i `LightBlock`, inicjalizacja zegara, określanie osi i prędkości kątowej ruchu obrotowego modelu, inicjalizacja macierzy modelu, inicjalizacja rzutowania perspektywicznego i źródeł światła, wybór początkowo wyświetlanych siatek (krawędzie siatki oryginalnej i ściany siatki po dwóch zagęszczeniach), a potem do pamięci GPU siatka dłoni jest wprowadzana i poddawana kolejnym zagęszczeniom. Przekazanie adresu zmiennej `wdg` (będącej polem zmiennej `appdata`) w instrukcji `return` umożliwia części okienkowej „doczepienie" wihajstrów (przełączników) do odpowiednich pól tej zmiennej.

Listing 32.14. Procedura `InitMyWorld`

C

```
1: static AppData appdata;
2:
3: AppWidgets *InitMyWorld ( int argc, char *argv[], int width, int height )
4: {
5:   static const float model_rot_axis[3] = {0.0,1.0,0.0};
6:
7:   memset ( &appdata, 0, sizeof(AppData) );
8:   LoadMeshRefinementPrograms ( true, false );
9:   LoadMeshRenderingPrograms ( &appdata.mrprog );
10:  ConstructEmptyVAO ();
11:  appdata.trans.trbuf = NewUniformTransBlock ();
12:  appdata.light.lsbuf = NewUniformLightBlock ();
13:  TimerInit ();
14:  memcpy ( appdata.model_rot_axis, model_rot_axis, 3*sizeof(float) );
15:  appdata.speed = 0.5*3.1415926;
16:  SetupModelMatrix ( &appdata );
17:  InitCamera ( &appdata, width, height );
18:  InitLights ( &appdata );
19:  appdata.wdg.sw[0] = appdata.wdg.sw[2] = true;
20:  appdata.lod = 2;
21:  appdata.edges = appdata.wdg.animation = false;
22:  InitPalmMeshes ( &appdata );
23:  ExitIfGLError ( "InitMyObject" );
24:  return &appdata.wdg;
25: } /*InitMyWorld*/
```

Procedury `LoadMeshRefinementProgram` i `LoadMeshRenderingPrograms` z listingów 31.10 i 32.11 przygotowują programy zagęszczania i rysowania siatek. Do rysowania jest potrzebny pusty obiekt tablicy wierzchołków tworzony w linii 10. Bufory w pamięci GPU przechowujące macierze przekształceń i opisy źródeł światła są tworzone przez procedury z listingów 10.7 i 10.9. Macierze widoku i rzutowania przygotowuje procedura `InitCamera` z listingu 15.16, na którym jest też pokazana wywoływana przez nią procedura `_ResizeMyWorld` i procedura `InitLights`, wprowadzająca oświetlenie.

Listing 32.15. Procedury inicjalizacji części graficznej

C

```
1: void SetupModelMatrix ( AppData *ad )
2: {
3:   M4x4RotateVfv ( ad->trans.mm, ad->model_rot_axis, ad->model_rot_angle );
4:   LoadMMatrix ( &ad->trans, NULL );
5: } /*SetupModelMatrix*/
6:
7: void InitPalmMeshes ( AppData *ad )
8: {
```

```
 9:  static const GLfloat edges_colour[3] = {0.0,0.5,0.7};
10:  static const GLfloat facets_colour[3] = {0.91,0.65,0.5};
11:  KLMesh *palm;
12:  int    i;
13:
14:  palm = &ad->palm;
15:  if ( (palm->mesh[0] = EnterPalmToGPU ()) ) {
16:    for ( i = 1; i <= NPALMMESHES; i++ ) {
17:      if ( !(palm->mesh[i] = malloc ( sizeof(GPUmesh) )) )
18:        ExitOnError ( "InitPalmMeshes 0" );
19:      memset ( palm->mesh[i], 0, sizeof(GPUmesh) );
20:      if ( !GPUmeshRefinement ( MESHDEG,
21:                                palm->mesh[i-1], palm->mesh[i] ) )
22:        ExitOnError ( "InitPalmMeshes 1" );
23:      printf ( "%d: nv = %d, nhe = %d, nfac = %d\n", i,
24:               palm->mesh[i]->nv, palm->mesh[i]->nhe, palm->mesh[i]->nfac );
25:    }
26:    memcpy ( palm->ecolour, edges_colour, 3*sizeof(GLfloat) );
27:    memcpy ( palm->fcolour, facets_colour, 3*sizeof(GLfloat) );
28:  }
29:  else
30:    ExitOnError ( "InitPalmMeshes" );
31: } /*InitPalmMeshes*/
```

Rysowane przedmioty, tj. siatki dłoni, przygotowuje procedura `InitPalmMeshes`, która w linii 15 przesyła do pamięci GPU siatkę opisaną w podrozdziale 32.1, a następnie, w pętli, wywołuje (w linii 20) procedurę zagęszczania siatki opisaną w poprzednim rozdziale. W ten sposób powstają cztery siatki będące coraz dokładniejszymi przybliżeniami powierzchni gładkiej reprezentującej dłoń. W każdej chwili aplikacja będzie wyświetlać co najwyżej jedną z tych powierzchni. W liniach 26 i 27 są określane kolory krawędzi i ścian siatek.

Procedury `ResizeMyWorld` i `RotateViewer` są takie same jak w drugiej aplikacji (zobacz listing 15.17).

Listing 32.16 przedstawia procedury rysowania sceny i sprzątania podczas zatrzymania aplikacji. Procedura `RedrawMyWorld` wywołuje procedurę `DrawMyScene`, która na początek kasuje tło, a potem jest wywoływana procedura z listingu 32.7. Pętla w liniach 10–16 ma za zadanie wyszukanie pierwszej zagęszczonej siatki, której przełącznik jest włączony, i narysowanie jej, za pomocą procedury z listingu 32.10. Jeśli użytkownik wyłączy wszystkie te przełączniki, to na obrazie może zobaczyć tylko siatkę niezagęszczoną[3].

Procedura `DeleteMyWorld` likwiduje programy szaderów, reprezentacje siatek, bufory z macierzami i światłami i pusty obiekt tablicy wierzchołków, zostawiając porządek po części graficznej aplikacji.

Listing 32.17 przedstawia pozostałe procedury części graficznej wywoływane z części okienkowej aplikacji. Procedura `ProcessSwitchCommand` jest wywoływana po każdej

[3] lub tylko tło, jeśli jej rysowanie też wyłączył

Listing 32.16. Procedury rysowania i sprzątania

C

```
void DrawMyScene ( AppData *ad, AppWidgets *wdg )
{
  int i;

  glClearColor ( 1.0, 1.0, 1.0, 1.0 );
  glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glEnable ( GL_DEPTH_TEST );
  if ( wdg->sw[0] )
    DrawMeshEdges ( &ad->mrprog, ad->palm.mesh[0], ad->palm.ecolour );
  for ( i = 1; i <= NPALMMESHES; i++ )
    if ( wdg->sw[i] ) {
      if ( ad->edges )
        DrawMeshEdges ( &ad->mrprog, ad->palm.mesh[i], ad->palm.fcolour );
      else
        DrawMeshFacets ( &ad->mrprog, ad->palm.mesh[i], ad->palm.fcolour );
    }
} /*DrawMyScene*/

void RedrawMyWorld ( void )
{
  DrawMyScene ( &appdata, &appdata.wdg );
  glFlush ();
} /*RedrawMyWorld*/

void DeleteMyWorld ( void )
{
  int i;

  DeleteMeshRefinementPrograms ();
  DeleteMeshRenderingPrograms ( &appdata.mrprog );
  for ( i = 0; i <= NPALMMESHES; i++ )
    DeleteGPUmesh ( appdata.palm.mesh[i] );
  glDeleteBuffers ( 1, &appdata.trans.trbuf );
  glDeleteBuffers ( 1, &appdata.light.lsbuf );
  DeleteEmptyVAO ();
  ExitIfGLError ( "DeleteMyWorld" );
} /*DeleteMyWorld*/
```

zmianie stanu któregoś przełącznika w menu. Pierwszy przełącznik steruje wyświetlaniem siatki niezagęszczonej i działa niezależnie od pozostałych. Z pozostałych przełączników może być włączony tylko jeden naraz, zatem włączenie dowolnego z nich powoduje wyłączenie pozostałych.

Procedura `ProcessCharCommand` jest wywoływana po napisaniu dowolnego znaku na klawiaturze, gdy kursor jest w obszarze okna z obrazem. Napisanie litery `K` zmienia war-

Listing 32.17. Procedury `ProcessSwitchCommand`, `ProcessCharCommand` i `MoveOn`

C

```
char ProcessSwitchCommand ( int wdg_id )
{
  switch ( wdg_id ) {
case SW_ID_MESH1:  case SW_ID_MESH2:  case SW_ID_MESH3:  case SW_ID_MESH4:
    if ( appdata.wdg.sw[wdg_id-SW_ID_MESH0] ) {
      memset ( &appdata.wdg.sw[1], false, NPALMMESHES );
      appdata.wdg.sw[(int)(appdata.lod = wdg_id-SW_ID_MESH0)] = true;
    }
    return true;
case SW_ID_MESH0:
    return true;
default:
    return false;
  }
} /*ProcessSwitchCommand*/

void ToggleAnimation ( AppData *ad )
{
  if ( (ad->wdg.animation = !ad->wdg.animation) ) {
    ProcessWorldRequest ( WMSG_ANIMATION_ON, NULL, NULL );
    TimerTic ();
  }
  else
    ProcessWorldRequest ( WMSG_ANIMATION_OFF, NULL, NULL );
} /*ToggleAnimation */

char ProcessCharCommand ( char charcode )
{
  switch ( toupper ( charcode ) ) {
case ' ':
    ToggleAnimation ( &appdata );
    return true;
case 'K':
    appdata.edges = !appdata.edges;
    return true;
default:
    return false;
  }
} /*ProcessCharCommand*/

char MoveOn ( void )
{
  if ( appdata.wdg.animation ) {
    if ( (appdata.model_rot_angle += appdata.speed * TimerTocTic ()) >= PI )
      appdata.model_rot_angle -= 2.0*PI;
```

```
46:     SetupModelMatrix ( &appdata );
47:   }
48:   return appdata.wdg.animation;
49: } /*MoveOn*/
```

tość zmiennej edges, czyli przełącznie między rysowaniem trójkątów (ścian siatki) wypełnionych a rysowaniem krawędzi siatki. Naciśnięcie klawisza spacji jest poleceniem włączenia albo wyłączenia animacji, która polega na obracaniu przedmiotu wokół ustalonej osi ze stałą prędkością kątową. W obu przypadkach pomocnicza procedura ToggleAnimation wywołuje procedurę ProcessWorldRequest, która, jeśli animacja została włączona, powoduje wywoływanie co chwila procedury MoveOn; ta ostatnia na podstawie odczytu zegara oblicza fazę ruchu (tj. bieżący kąt obrotu), konstruuje macierz modelu i przesyła ją do pamięci GPU. Niezerowa wartość powrotna opisanych tu procedur jest zawiadomieniem części okienkowej, że należy wykonać nowy obraz.

32.4. Okna trzeciej aplikacji

Graficzny interfejs użytkownika jest tworzony przy użyciu procedur opisanych w rozdziale 30. Aplikacja tworzy jedno okno z dwoma podoknami, z których pierwsze zawiera menu z guzikiem i przełącznikami, a w drugim jest wyświetlany obraz dłoni, tj. powierzchni zbudowanej z trójkątów odpowiednio zagęszczonej siatki opisanej w podrozdziale 32.1. Listing 32.18 przedstawia procedurę main oraz wywoływane przez nią procedury inicjalizacji i sprzątania. Nazwy kolejno wywoływanych procedur powinny dostatecznie objaśniać wykonywane przez nie zadania[4], z jednym wyjątkiem. Procedura XInternAtom otrzymuje od systemu X Window atom, który będzie identyfikatorem komunikatów używanych do zrealizowania animacji. W zasadzie można do tego użyć dowolnej liczby całkowitej, ale otrzymany atom będzie inny niż identyfikatory wszelkich własności (*properties*) określonych przez system, menedżera okien i aplikacje — chyba że któraś z nich została nazwana "aAnimate", co też nie wyrządzi żadnych szkód.

Ostatnie dwa parametry procedury InitApp3Windows w linii 14 określają numer potrzebnej wersji OpenGL-a. Ponieważ jest to pierwsza opisana tu aplikacja biblioteki X11, kolejne listingi przedstawiają kompletny opis jej procedur (z pominięciem tych, które zostały wzięte z poprzednich aplikacji bez zmian).

Listing 32.18. Procedura main i dwie inne procedury

C

```
1: static Window     window[3];
2: static GLXContext glxcontext;
3: static char       terminate;
4: static Atom       aAnimate;
```

[4] Procedury, których nazwy zaczynają się od litery X, należą do biblioteki X11. Informacje na ich temat najprościej jest uzyskać za pomocą programu man.

```
5:
6: static int         window0_width, window0_height;
7: static xwinmenu    *wm1, *wm2;
8: static xwidget     *mywdg;
9: static AppWidgets *appwdg;
10:
11: void Initialise ( int argc, char **argv )
12: {
13:   InitXServerConnection ( argc, argv, false );
14:   InitApp3Windows ( argc, argv, APP3_GL_MAJOR, APP3_GL_MINOR );
15:   aAnimate = XInternAtom ( xdisplay, "aAnimate", False );
16:   appwdg = InitMyWorld ( argc, argv, WIN0_WIDTH-MENU_WIDTH, WIN0_HEIGHT );
17:   wm1 = SetupApp3Menu ();
18:   wm2 = SetupApp3GLWindow ();
19: } /*Initialise*/
20:
21: void Cleanup ( void )
22: {
23:   DeleteMyWorld ( &myglwdata, &appdata );
24:   DeleteWinMenu ( wm1 );
25:   DeleteWinMenu ( wm2 );
26:   XDestroySubwindows ( xdisplay, window[0] );
27:   XDestroyWindow ( xdisplay, window[0] );
28:   XFreeGC ( xdisplay, xgc );
29:   XCloseDisplay ( xdisplay );
30: } /*Cleanup*/
31:
32: int main ( int argc, char **argv )
33: {
34:   Initialise ( argc, argv );
35:   MessageLoop ();
36:   Cleanup ();
37:   exit ( 0 );
38: } /*main*/
```

Listing 32.19 przedstawia procedurę, która tworzy okna aplikacji. Dalej będzie mowa o **oknie głównym**, **oknie menu** i **oknie obrazu**. Podane przez system X Window identyfikatory tych okien zostaną zapamiętane odpowiednio w zmiennych `window[0]`, `window[1]` i `window[2]`.

Okna menu i obrazu są podoknami okna głównego i w całości wypełniają jego obszar. Zawartość okna menu ma być rysowana przez procedury z biblioteki X11, z kolei zawartość okna obrazu wyprodukuje OpenGL. Procedura pokazana na listingu 32.19 powstała przez modyfikację procedury `InitMyGLXWindow` z listingu 3.7; niezmienione instrukcje powinny być (i są) jasne.

W aplikacji OpenGL-a przeznaczonej do działania w systemie X Window możemy również użyć wielokrotnego próbkowania w celu przeprowadzania antyaliasingu. W tym celu,

wywołując procedurę `InitGLXContext`, trzeba podać tablicę potrzebnych atrybutów wizualu z dodanymi liniami 14 i 15 (porównaj tablicę `visattr` z tablicą `vattr` na listingu 3.6). Liczba próbek na piksel ma być w tablicy podana po stałej symbolicznej `GLX_SAMPLES`.

Listing 32.19. Tworzenie okien trzeciej aplikacji

C

```
1: #define WIN0_WIDTH   480
2: #define WIN0_HEIGHT  360
3: #define MENU_WIDTH   120
4:
5: void InitApp3Windows ( int argc, char **argv, int major, int minor )
6: {
7:   int vattr[] =
8:     { GLX_RGBA,           True,
9:       GLX_DOUBLEBUFFER,   True,
10:      GLX_RED_SIZE,       8,
11:      GLX_GREEN_SIZE,     8,
12:      GLX_BLUE_SIZE,      8,
13:      GLX_DEPTH_SIZE,    24,
14:      GLX_SAMPLE_BUFFERS, True,
15:      GLX_SAMPLES,        8,
16:      None };
17:  static const int wx[3] = { 0, 0, MENU_WIDTH };
18:  static const int wh[3] = { WIN0_WIDTH, MENU_WIDTH,
19:                           WIN0_WIDTH-MENU_WIDTH };
20:  XSetWindowAttributes swa;
21:  Colormap             xcolormap;
22:  XVisualInfo          *xvii;
23:  Window               upw;
24:  int                  i;
25:
26:  InitGLXContext ( major, minor, vattr, &xvii, &glxcontext );
27:  if ( !(xcolormap = XCreateColormap ( xdisplay, xrootwin,
28:                                       xvii->visual, AllocNone )) )
29:    ExitOnError ( "InitApp3Windows 1" );
30:  swa.colormap = xcolormap;
31:  swa.event_mask = ExposureMask | StructureNotifyMask| ButtonPressMask |
32:                   ButtonReleaseMask | PointerMotionMask | KeyPressMask;
33:  for ( i = 0, upw = xrootwin; i < 3; i++, upw = window[0] ) {
34:    window[i] = XCreateWindow ( xdisplay, upw, wx[i], 0, wh[i], WIN0_HEIGHT,
35:                                0, xvii->depth, InputOutput, xvii->visual,
36:                                CWColormap | CWEventMask, &swa );
37:    XMapWindow ( xdisplay, window[i] );
38:  }
39:  XFreeColormap ( xdisplay, xcolormap );
40:  XFree ( xvii );
41:  XSetWMProtocols ( xdisplay, window[0], &DeleteWindow, 1 );
```

```
42:   XStoreName ( xdisplay, window[0], "Trzecia aplikacja" );
43:   xgc = XCreateGC ( xdisplay, window[1], 0, 0 );
44:   InitRGBXColourmap ();
45:   InitWinMenuPalette ();
46:   if ( !glXMakeCurrent ( xdisplay, window[2], glxcontext ) )
47:     ExitOnError ( "InitApp3Windows 2" );
48:   GetGLProcAddresses ( major, minor );
49:   PrintGLVersion ();
50:   window0_width = WIN0_WIDTH;
51:   window0_height = WIN0_HEIGHT;
52: } /*InitApp3Windows*/
```

Przodkiem głównego okna aplikacji jest pulpit lub inne okno wskazane przez menedżera okien; jego identyfikator jest otrzymywany za pomocą procedury `RootWindow`. Okno główne jest zgłoszone jako przodek okna menu oraz okna aplikacji, a zatem są one podoknami okna głównego. Relacje między oknami są ustalane przez drugi parametr procedury `XCreateWindow`.

Listing 32.20 przedstawia procedury obsługi menu aplikacji. Po utworzeniu menu w linii 42 jest tworzony guzik, a potem w pętli przełączniki. Adres zmiennej, której przełącznik ma nadawać wartości, jest polem struktury wskazywanej przez zmienną `appwdg`; wartość tej zmiennej została nadana przez instrukcję w linii 16 na listingu 32.18.

Listing 32.20. Tworzenie i obsługa menu

C

```
1: char str_EXIT[] = "Exit";  /* napis na guziku */
2:
3: void Win1Callback ( struct xwidget *wdg, int msg, int key, int x, int y )
4: {
5:   switch ( msg ) {
6: case WDGMSG_BUTTON_PRESS:
7:     switch ( wdg->id ) {
8:   case BTN_ID_EXIT:
9:       terminate = true;  /* powoduje zakończenie działania */
10:      break;
11:  default:
12:      break;
13:    }
14:    break;
15:
16: case WDGMSG_SWITCH_CHANGE:
17:    if ( ProcessSwitchCommand ( wdg->id ) ) {
18:      wm2->changed = true;
19:      PostMenuExposeEvent ( wm2 );
20:    }
21:    break;
22:
23: case XWMSG_KEY_PRESS:
```

```
24:     mywdg->input ( mywdg, msg, key, x, y );
25:     if ( wm2->changed )
26:       PostMenuExposeEvent ( wm2 );
27:     break;
28:
29: default:
30:     break;
31:   }
32: } /*Win1CallBack*/
33:
34: xwinmenu* SetupApp3Menu ( void )
35: {
36:   xwinmenu *wm;
37:   int      i;
38:
39:   if ( !(wm = NewWinMenu ( window[1], MENU_WIDTH, WIN0_HEIGHT, 0, 0,
40:                            NULL, NULL, Win1Callback )) )
41:     ExitOnError ( "SetupApp3Menu" );
42:   NewButton ( wm, BTN_ID_EXIT, 60, 18, 2, 2, str_EXIT );
43:   for ( i = 0; i < NPALMMESHES; i++ )
44:     NewSwitch ( wm, SW_ID_MESH0+i, 16, 16, 2+20*i, 22, NULL,
45:                 &appwdg->meshsw[i] );
46:   return wm;
47: } /*SetupApp3Menu*/
```

Procedura `Win1Callback`, wywołana po pstryknięciu guzika, w linii 9 nadaje zmiennej `terminate` wartość powodującą zakończenie pętli komunikatów i zatrzymanie aplikacji. Po pstryknięciu przełącznika procedura ta wywołuje procedurę `ProcessSwitchCommand` z części graficznej. Jej parametr jest identyfikatorem przełącznika, który ma w tym momencie nadaną nową wartość. Procedura `ProcessSwitchCommand` może zmienić wartości innych przełączników; gdy tak się stanie, przekaże wartość powrotną `true`. Instrukcje w liniach 18 i 19 spowodują wykonanie w oknie menu uaktualnionego obrazu wszystkich wihajstrów.

W liniach 23–27 następuje przekazanie wszystkich komunikatów o naciśnięciu klawisza, nieobsłużonych przez wihajstry w menu pierwszego podokna, wihajstrowi wyświetlającemu obraz sceny w drugim podoknie. Dzięki temu na przykład naciśnięcie spacji uruchamia lub zatrzymuje obracanie obiektu niezależnie od położenia kursora w głównym oknie aplikacji. Jeśli obraz sceny ma zostać zmieniony, to instrukcja w linii 26 powoduje wykonanie nowego obrazu.

Listing 32.21 przedstawia procedurę `SetupApp3GLWindow`, która tworzy menu dla okna z obrazem; menu to zawiera jeden wihajster, wyświetlający obraz i przyjmujący polecenia wydawane za pomocą klawiatury i myszy, oraz procedury reagujące na zdarzenia związane z tym oknem. Metody wihajstra są podane dalej.

Struktura typu `widget3d` zawiera dane wihajstra obrazu; pole `data0` wihajstra wskazuje taką strukturę, utworzoną przez procedurę konstrukcji wihajstra. Są w niej pola do prze-

Listing 32.21. Procedury przetwarzania wejścia okna z obrazem

C

```
typedef struct {
    int  last_x, last_y;
    char opti;
  } widget3d;

void RedrawWin2 ( xwinmenu *wm )
{
  widget3d *ww;

  ww = (widget3d*)mywdg->data0;
  if ( ww->opti > 0 ) {
    ww->opti --;
    PostExposeEvent ( wm->window, wm->r.width, wm->r.height );
  }
  else {
    for ( wdg = wm->wlist.next;  wdg;  wdg = wdg->link.next )
      wdg->redraw ( wdg );
    glFlush ();
  }
  glXSwapBuffers ( xdisplay, wm->window );
} /*RedrawWin2*/

void Win2Callback ( struct xwidget *wdg, int msg, int key, int x, int y )
{
  widget3d *ww;

  ww = (widget3d*)mywdg->data0;
  switch ( msg ) {
case WDGMSG_RECONFIGURE:
    mywdg->input ( mywdg, WDGMSG_RECONFIGURE, 0, x, y );
    ww->opti = 4;
    PostMenuExposeEvent ( wdg->wm );
    break;
case XWMSG_CLIENT_MESSAGE:
    mywdg->input ( mywdg, msg, key, x, y );
    break;
default:
    break;
  }
} /*Win2Callback*/

xwinmenu *SetupApp3GLWindow ( void )
{
  xwinmenu *wm;

```

```
46:   if ( !(wm = NewWinMenu ( window[2], WIN0_WIDTH-MENU_WIDTH, WIN0_HEIGHT,
47:                            0, 0, NULL, RedrawWin2, Win2Callback )) )
48:     ExitOnError ( "SetupApp3GLWindow 0" );
49:   if ( !(mywdg = New3DWidget ( wm, GLWIN_ID_VIEW,
50:                                wm->r.width, wm->r.height )) )
51:     ExitOnError ( "SetupApp3GLWindow 1" );
52:   return wm;
53: } /*SetupApp3GLWindow*/
```

chowywania poprzedniego położenia kursora w oknie (na potrzeby obracania obserwatora wokół obiektu) i zmienną `opti`, której rola jest wyjaśniona w p. 3.5.2.

W liniach 46–47 jest tworzone menu, a w liniach 49–50 jedyny wihajster tego menu, wyświetlający obraz i umożliwiający oglądanie przedstawionych na nim obiektów z różnych stron. Wskaźnik do tego wihajstra jest przypisywany zmiennej `mywdg`. Procedura `RedrawWin2` wykonuje obraz. Procedura `Win2Callback` jest wywoływana po otrzymaniu przez okno komunikatów `ConfigureNotify` i `ClientMessage`; w pierwszym przypadku zawiadamia wihajster o zmianie wymiarów i żąda wykonania nowego obrazu, a w drugim przekazuje wihajstrowi komunikat wysłany w celu kontynuowania animacji.

Listing 32.22 przedstawia procedurę tworzenia wihajstra z obrazem i jego metody. Procedura `My3DWidgetRedraw` wywołuje procedurę z części graficznej, która wykonuje obraz.

Procedura `My3DWidgetInput` przetwarza komunikaty wejściowe. Wihajster ma dwa stany, nazwane `WDGSTATE_DEFAULT` i `WDGSTATE_VIEW_TURNING`; przełączanie między nimi następuje po naciśnięciu i zwolnieniu lewego przycisku myszy, podobnie jak w aplikacjach pierwszej i drugiej. W tym drugim stanie przesuwanie myszy powoduje wywoływanie procedury `RotateViewer`.

W obu stanach wihajster tak samo reaguje na pisanie znaków na klawiaturze (wywołując procedurę `ProcessCharCommand` z części graficznej) i na komunikaty `ClientMessage`. Po sprawdzeniu (w linii 64), że komunikat `ClientMessage` jest związany z animacją, jest wywoływana procedura `MoveOn` z części graficznej, która odczytuje zegar i oblicza nowe położenia obiektów. Instrukcje w liniach 66 i 67 zawiadamiają o konieczności wykonania nowego obrazu, a instrukcja w linii 69 wysyła kolejny komunikat `ClientMessage` dla podtrzymania ruchu.

Listing 32.22. Metody wihajstra z obrazem

C

```
1: void My3DWidgetRedraw ( struct xwidget *wdg )
2: {
3:   RedrawMyWorld ();
4: } /*My3DWidgetRedraw*/
5:
6: char My3DWidgetInput ( struct xwidget *wdg, int msg, int key, int x, int y )
7: {
8:   XClientMessageEvent *xclient;
9:   widget3d            ww;
```

```
10:
11:   ww = (widget3d*)wdg->data0;
12:   switch ( wdg->state ) {
13: case WDGSTATE_DEFAULT:
14:     switch ( msg ) {
15:   case XWMSG_BUTTON_PRESS:
16:       if ( key == Button1 ) {
17:         wdg->last_x = x;  wdg->last_y = y;
18:         wdg->state = WDGSTATE_VIEW_TURNING;
19:         GrabInput ( wdg );
20:         return true;
21:       }
22:       break;
23:   case XWMSG_KEY_PRESS:
24:       goto process_key;
25:   case WDGMSG_RECONFIGURE:
26:       ResizeMyWorld ( wdg->r.width = x, wdg->r.height = y );
27:       break;
28:   case XWMSG_CLIENT_MESSAGE:
29:       goto process_client_message;
30:   case XWMSG_DELETE:
31:       goto process_delete_message;
32:   default:
33:       break;
34:     }
35:     break;
36:
37: case WDGSTATE_VIEW_TURNING:
38:     switch ( msg ) {
39:   case XWMSG_MOUSE_MOTION:
40:       if ( ((XMotionEvent*)wdg->wm->ev)->state & Button1Mask ) {
41:         RotateViewer ( (double)(x - ww->last_x), (double)(y - ww->last_y) );
42:         ww->last_x = x;  ww->last_y = y;
43:         wdg->wm->changed = true;
44:       }
45:       else
46:         goto release;
47:       break;
48:   case XWMSG_BUTTON_RELEASE:
49:       if ( key == Button1 ) {
50: release:
51:         wdg->state = WDGSTATE_DEFAULT;
52:         UngrabInput ( wdg );
53:         return true;
54:       }
55:       break;
56:   case XWMSG_KEY_PRESS:
```

```
process_key:
      if ( ProcessCharCommand ( key ) )
        return wdg->wm->changed = true;
      break;
  case XWMSG_CLIENT_MESSAGE:
process_client_message:
      xclient = (XClientMessageEvent*)wdg->wm->ev;
      if ( xclient->message_type == aAnimate && appwdg->animation ) {
        if ( MoveOn () ) {
          wm2->changed = true;
          PostMenuExposeEvent ( wm2 );
        }
        PostClientMessageEvent ( window[2], aAnimate, 8, NULL );
      }
      break;
  case XWMSG_DELETE:
process_delete_message:
      free ( wdg->data0 );
      return true;
  default:
      break;
    }
    break;

default:
    break;
  }
  return false;
} /*My3DWidgetInput*/

xwidget *New3DWidget ( struct xwinmenu *wm, int id, int w, int h )
{
  widget3d *ww;

  if ( !(ww = malloc ( sizeof(widget3d) )) )
    ExitOnError ( "New3DWidget" );
  ww->opti = 0;
  return NewWidget ( wm, sizeof(xwidget), id, w, h, 0, 0,
                     My3DWidgetInput, My3DWidgetRedraw, ww, NULL );
} /*New3DWidget*/
```

Procedura `New3DWidget` jest konstruktorem wihajstra obrazu; rezerwuje ona pamięć na dane dodatkowe i nadaje początkową wartość polu `opti`, a następnie wywołuje procedurę `NewWidget`, która tworzy wihajster, włącza go do menu, przypisuje wskaźniki metod `input` i `redraw` tego wihajstra i przypisuje polu `data0` adres bloku pamięci z danymi dodatkowymi. Blok ten jest zwalniany po otrzymaniu przez wihajster komunikatu `XWMDG_DELETE`, wysyłanego przez menu w trakcie jego likwidacji.

Listing 32.23 przedstawia procedurę `ProcessWorldRequest`, wywoływaną przez część graficzną, gdy polecnie wydane przez naciśnięcie klawisza spacji ma uruchomić albo zatrzymać animację. Włączenie animacji polega na wysłaniu pierwszego z serii komunikatu `ClientMessage` do okna obrazu. Wyłączenie animacji nie wymaga żadnych działań; wartość `false` zmiennej `appwdg->animation` (nadana przez część graficzną) zatrzyma wysyłanie komunikatów `ClientMessage` „napędzających" animację[5].

Listing 32.23. Procedura `ProcessWorldRequest`

C

```
1: char ProcessWorldRequest ( int msg, void *data, void *reply )
2: {
3:   switch ( msg ) {
4: case WMSG_ANIMATION_ON:
5:     PostClientMessageEvent ( window[2], aAnimate, 8, NULL );
6:     return true;
7: case WMSG_ANIMATION_OFF:
8:     return true;
9: default:
10:    return false;
11:  }
12: } /*ProcessWorldRequest*/
```

Pozostałe procedury części okienkowej przedstawia listing 32.24. Procedura `MessageLoop` realizuje główną pętlę komunikatów, w której dla każdego komunikatu otrzymanego od systemu X Window ustala adresata, tj. jedno z trzech okien aplikacji i wywołuje odpowiednią procedurę, aby przetworzyła ten komunikat. Procedura przetwarzania komunikatów okna głównego reaguje na dwa zdarzenia: komunikat `ConfigureNotify`, otrzymany po zmianie wymiarów okna przez użytkownika, powoduje obliczenie nowych wymiarów podokien, tj. okna menu i okna obrazu. Oba podokna mają wysokość taką jak okno główne. Okno menu ma stałą szerokość `MENU_WIDTH` pikseli i zajmuje obszar z lewej strony okna głównego, a okno obrazu zajmuje pozostały obszar okna głównego. Wywołania procedury `XMoveResizeWindow` w liniach 7–10 spowodują wysłanie komunikatów `ConfigureNotify` do podokien, a obsługą tych komunikatów zajmą się metody obiektów menu.

Listing 32.24. Procedury `WinOMessageProc` i `MessageLoop`

C

```
1: void WinOMessageProc ( XEvent *ev )
2: {
3:   XClientMessageEvent *xclient;
4:
5:   switch ( ev->xany.type ) {
```

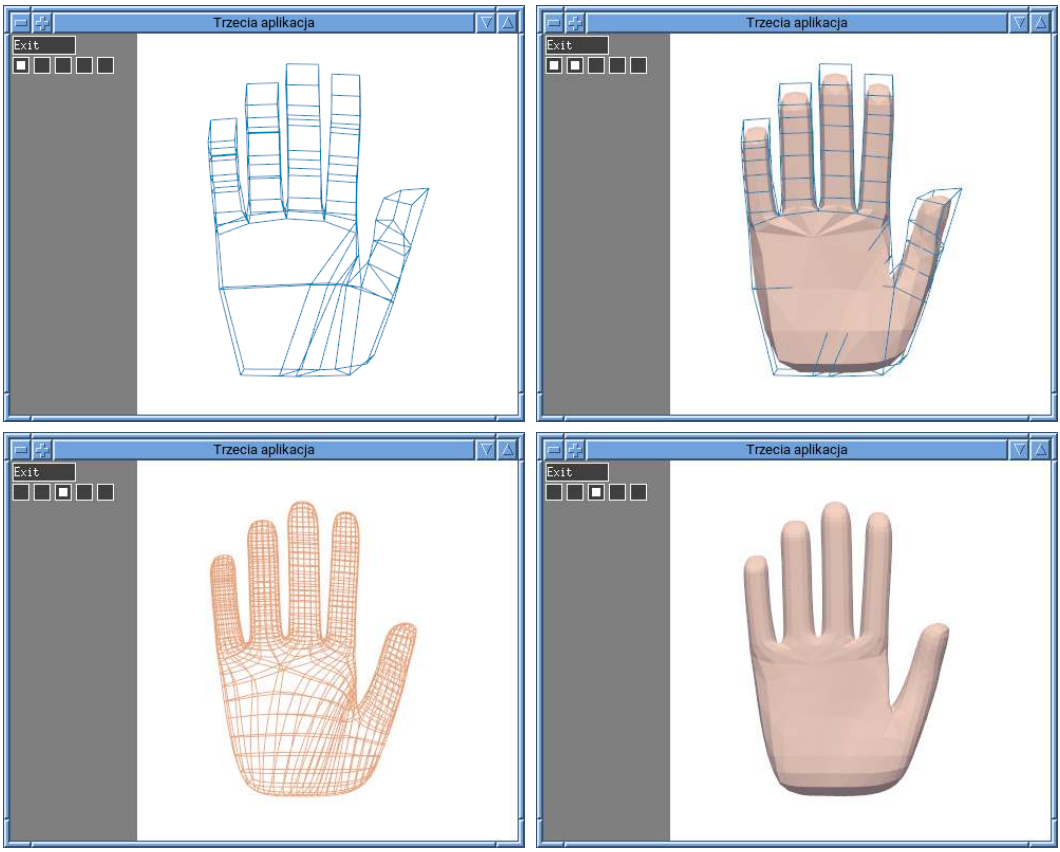
[5] Gdyby część okienkowa była zrealizowana na przykład przy użyciu biblioteki `FreeGLUT`, animacja mogłaby być wyłączana przez wykonanie instrukcji `glutIdleFunc ( NULL );`. Interfejs między częściami okienkową a graficzną zawiera polecenie wyłączenia animacji po to, aby to umożliwić bez uzależniania części graficznej od biblioteki okienkowej.

```
case ConfigureNotify:
    XMoveResizeWindow ( xdisplay, window[1], 0, 0,
                        MENU_WIDTH, window0_height = ev->xconfigure.height );
    XMoveResizeWindow ( xdisplay, window[2], MENU_WIDTH, 0,
        (window0_width = ev->xconfigure.width)-MENU_WIDTH, window0_height );
    break;
case ClientMessage:
    xclient = (XClientMessageEvent*)ev;
    if ( xclient->message_type == WMProtocols &&
         (Atom)xclient->data.l[0] == DeleteWindow ) {
      terminate = true;
      break;
    }
default:
    break;
  }
} /*Win0MessageProc*/

void MessageLoop ( void )
{
  XEvent ev;

  terminate = false;
  do {
    XNextEvent ( xdisplay, &ev );
    if ( ev.xany.window == window[0] )
      Win0MessageProc ( &ev );
    else if ( ev.xany.window == window[1] )
      WinMenuInput ( wm1, &ev );
    else if ( ev.xany.window == window[2] )
      WinMenuInput ( wm2, &ev );
  } while ( !terminate );
} /*MessageLoop*/
```

Użytkownik, za pośrednictwem menedżera okien, może wysłać do aplikacji polecenie zatrzymania jej. Wtedy w komunikacie `ClientMessage` jest przekazany atom `DeleteWindow`; zmienna `terminate` otrzymuje wartość `true` i aplikacja kończy działanie.

32.5. Ćwiczenia

1. Dodaj suwak umożliwiający zmienianie prędkości obracania dłoni podczas animacji.
2. *Napisz szader obliczeniowy, który dla siatki przechowywanej w pamięci GPU wpisze do tablicy w odpowiednim buforze pary indeksów wierzchołków końcowych krawędzi, aby można było te krawędzie narysować za pomocą procedury `glDrawElements`. W wersji łatwiejszej szader ma wpisywać do tablicy dane dla wszystkich półkrawędzi, a w wersji



Rysunek 32.1. Siatki wyświetlane przez trzecią aplikację

trudniejszej tylko dla półkrawędzi brzegowych i dla jednej półkrawędzi z pary reprezentującej krawędź wewnętrzną — trzeba zatem dla każdej półkrawędzi wyznaczyć miejsce w tablicy, w którym mają być zapamiętane indeksy jej końców, obliczając odpowiedni ciąg sum prefiksowych (zobacz rozdz. 31).

3. *Napisz szader obliczeniowy, który do tablicy w pamięci GPU wpisze indeksy trójkątów otrzymanych z podziału ścian, aby umożliwić jej narysowanie za pomocą procedury `glDrawElements`. W wersji łatwiejszej można przyjąć, że wszystkie ściany są czworokątne. W wersji trudniejszej należy dopuścić dowolne siatki, co wymaga obliczenia wieloetapowego, z obliczaniem sum prefiksowych.

4. *Napisz i uruchom szader obliczeniowy, który „oznaczy" ściany siatki w „warstwie brzegowej" (nadając odpowiednią wartość bitowi określonemu przez maskę `TAGMASK` na listingu 31.3).

5. *Napisz i uruchom program szaderów, który narysuje ściany nie „oznaczone" na przykład przez szader będący rozwiązaniem poprzedniego ćwiczenia.

33

Aplikacja trzecia A

Dodamy szader obliczeniowy, którego zadaniem jest obliczenie, dla każdego wierzchołka siatki, wektora normalnego. Siatki otrzymane przez zagęszczanie przybliżają gładką powierzchnię graniczną, której obraz chcemy uzyskać. Obliczone wektory normalne będą przybliżać wektory normalne powierzchni granicznej, a ich interpolacja doprowadzi do otrzymania obrazu cieniowanego, który wygląda jak obraz gładkiej powierzchni. Do dzieła.

33.1. Obliczanie wektorów normalnych

Współrzędne wierzchołków siatki są upakowane w tablicy liczb zmiennopozycyjnych, przy czym dopuszczamy 2, 3 lub 4 współrzędne wierzchołków — w pierwszym przypadku siatka leży w płaszczyźnie xy i dla wszystkich wierzchołków mamy wektor normalny $(0,0,1)$, zatem wszelkie jego obliczenia są zbędne, a w trzecim przypadku mamy współrzędne jednorodne wierzchołków. Obliczone wektory normalne umieścimy w tej samej tablicy. W tym celu utworzymy dodatkowy bufor z tablicą na wyniki; dla każdego wierzchołka szader skopiuje współrzędne wierzchołka i dopisze do nich współrzędne wektora normalnego. Ale to wymaga „porozsuwania" wierzchołków w tablicy, bo zamiast trzech lub czterech liczb mamy ich dla każdego wierzchołka sześć lub siedem. Możemy też zarezerwować w tablicy miejsce na dodatkowe atrybuty, takie jak kolor lub współrzędne tekstury[1].

Szader obliczeniowy, którego zadaniem jest obliczenie wektora normalnego, *zastąpi* tablicę, w której są tylko współrzędne położeń wierzchołków, przez tablicę, w której są także współrzędne wektorów normalnych i ewentualnie miejsce na współrzędne tekstury. Rozmieszczenie atrybutów wierzchołków siatki danej i siatki wynikowej opiszemy za pomocą zmiennych jednolitych umieszczonych w bloku o nazwie `MeshNV`. Zmienne `innsattr` i `outnsattr` opisują liczbę wszystkich skalarnych atrybutów wierzchołka siatki w tablicy danej i wynikowej. Zmienna `pdim` określa liczbę współrzędnych wierzchołka (2, 3 lub 4).

[1] To wymaga odpowiedniego rozbudowania struktur typu `CPUmesh` i `GPUmesh` oraz bloku magazynowego `meshsurf` o pola opisujące położenia tych atrybutów w tablicy. Na przykład pola przechowujące liczbę współrzędnych tekstury i numer pierwszej z nich dla wierzchołka można nazwać `tdim` i `tofs`.

Zmienne `inpofs` i `outpofs` opisują, który atrybut jest pierwszą współrzędną położenia wierzchołka w siatce danej i wynikowej. Zmienna `outnvofs` określa miejsce, od którego szader ma umieścić w tablicy wynikowej obliczone współrzędne wektora normalnego. Jeśli na przykład wierzchołki mają podane tylko 3 współrzędne swojego położenia, to zmienne `pdim` oraz `innsattr` mają wartość 3, a `inpofs` ma wartość 0. Jeśli wtedy w tablicy z wynikami mają być tylko współrzędne położenia i współrzędne wektora normalnego, to przypiszemy `outnsattr = 6`, `outpofs = 0` i `outnvofs = 3`.

Do punktów dowiązania 0, 1, 2 i 3 celu `GL_SHADER_STORAGE_BUFFER` przywiążemy odpowiednio bufory, z których pierwszy zawiera tablice wierzchołków i ścian oraz tablice z listami indeksów ich półkrawędzi, drugi zawiera tablicę półkrawędzi, trzeci tablicę z danymi współrzędnymi wierzchołków, a ostatni tablicę na wynik. Makrodefinicje w liniach 18–24 na listingu 33.1 ułatwiają dostęp do tablic w tych buforach.

Listing 33.1. Szader obliczeniowy wektorów normalnych wierzchołków siatki

GLSL

```
1: #version 450 core
2:
3: .... /* tu makrodefinicje FHEMASK i DEGSHIFT jak na listingu 31.3 */
4: .... /* oraz makrodefinicje V0, V1, FACN i OTHE jak na listingu 31.9 */
5:
6: layout(local_size_x=1) in;
7:
8: layout(std430,binding=0) buffer Inmvf { int   mvf[]; } mvf;
9: layout(std430,binding=1) buffer Inmhe { ivec4 mhe[]; } mhe;
10: layout(std430,binding=2) buffer Invc  { float vc[];  } inmvc;
11: layout(std430,binding=3) buffer Outvc { float vc[];  } outmvc;
12:
13: uniform MeshNV {
14:     int innsattr, pdim, inpofs, inv, inhe, infac,
15:         outnsattr, outpofs, outnvofs;
16:   };
17:
18: #define mv(I)    mvf.mvf[I]
19: #define mfac(I)  mvf.mvf[inv+(I)]
20: #define mvhei(I) mvf.mvf[inv+infac+(I)]
21: #define mfhei(I) mvf.mvf[inv+infac+inhe+(I)]
22: #define mhe(I)   mhe.mhe[I]
23: #define imvc(I)  inmvc.vc[I]
24: #define omvc(I)  outmvc.vc[I]
25:
26: vec3 GetVertexPos3f ( int i )
27: {
28:   i = innsattr*i + inpofs;
29:   return vec3 ( imvc(i), imvc(i+1), imvc(i+2) );
30: } /*GetVertexPos3f*/
31:
```

```
32: void MeshVertexNormal3f ( int i )
33: {
34:   int  d, fhe, j, m, f, fd, ffhe;
35:   vec3 p0, p1, p2, v1, v2, nv;
36:
37:   fhe = mv(i) & FHEMASK;
38:   d = mv(i) >> DEGSHIFT;
39:   p0 = GetVertexPos3f ( i );
40:   m = outnsattr*i + outpofs;
41:   omvc(m) = p0.x;  omvc(m+1) = p0.y;  omvc(m+2) = p0.z;
42:   nv = vec3 ( 0.0 );
43:   if ( mhe(mvhei(fhe+d-1)).OTHE < 0 ) {
44:     f = mhe(mvhei(fhe)).FACN;
45:     ffhe = mfac(f) & FHEMASK;
46:     fd = mfac(f) >> DEGSHIFT;
47:     for ( j = 0; j < fd; j++ )
48:       if ( mhe(mfhei(ffhe+j)).V1 == i )
49:         break;
50:     v1 = GetVertexPos3f ( mhe(mfhei(ffhe+j)).V0 ) - p0;
51:   }
52:   else
53:     v1 = GetVertexPos3f ( mhe(mvhei(fhe+d-1)).V1 ) - p0;
54:   for ( j = 0; j < d; j++ )  {
55:     v2 = GetVertexPos3f ( mhe(mvhei(fhe+j)).V1 ) - p0;
56:     nv += cross ( v1, v2 );
57:     v1 = v2;
58:   }
59:   nv = normalize ( nv );
60:   m = outnsattr*i + outnvofs;
61:   omvc(m) = nv.x;  omvc(m+1) = nv.y;  omvc(m+2) = nv.z;
62: } /*MeshVertexNormal3f*/
63:
64: void main ( void )
65: {
66:   switch ( pdim ) {
67: case 3:  MeshVertexNormal3f ( int(gl_GlobalInvocationID.x) );  break;
68: case 4:  break;  /* procedura do napisania jako ćwiczenie */
69: default: break;
70:   }
71: } /*main*/
```

Pomocnicza procedura `GetVertexPos3f` ma pobrać trzy współrzędne (kartezjańskie) położenia i-tego wierzchołka. Zasadnicze obliczenie wykonuje procedura `MeshVertexNormal3f`, która w linii 40 przepisuje współrzędne i-tego wierzchołka do tablicy wynikowej. Algorytm obliczania wektora normalnego jest następujący: i-ty wierzchołek jest końcem co najmniej dwóch (jeśli jest brzegowy) albo co najmniej trzech (jeśli jest wewnętrzny) krawędzi, które oby nie były równoległe. W każdej ścianie, która ma ten wierzchołek, znajdowane

są dwie półkrawędzie, takie że i-ty wierzchołek jest końcem pierwszej i początkiem drugiej z nich. Po odjęciu i-tego wierzchołka od pozostałych wierzchołków tych półkrawędzi otrzymujemy dwa wektory, $\boldsymbol{v}_1$ i $\boldsymbol{v}_2$, których iloczyn wektorowy jest wektorem normalnym płaszczyzny zawierającej krawędzie siatki reprezentowane przez te półkrawędzie. Wektory normalne płaszczyzn dla kolejnych ścian są sumowane (w celu uśrednienia kierunków), po czym suma jest poddawana normalizacji (tj. dzielona przez swoją długość) i oto mamy jednostkowy wektor normalny odpowiadający i-temu wierzchołkowi. Wystarczy jego współrzędne wpisać do tablicy wynikowej, co wykonują instrukcje w linii 61.

Dla wierzchołków brzegowych i wewnętrznych trzeba wykonać trochę inne instrukcje. Warunek w linii 43 jest spełniony dla wierzchołka brzegowego. Dla każdego wierzchołka mamy tylko listę indeksów półkrawędzi, których ten wierzchołek jest początkiem, nie ma w niej więc indeksu półkrawędzi brzegowej, której ten wierzchołek jest końcem. Dlatego dla wierzchołka brzegowego trzeba odnaleźć kończącą się w nim półkrawędź brzegową, przeszukując (w pętli w liniach 47–49) listę półkrawędzi *ściany pierwszej półkrawędzi wychodzącej* z wierzchołka. W linii 50 jest obliczany wektor $\boldsymbol{v}_1$ dla znalezionej półkrawędzi brzegowej wchodzącej do wierzchołka.

Jeśli wierzchołek jest wewnętrzny, to obliczenie jest znacznie prostsze; wektor $\boldsymbol{v}_1$ obliczony w linii 53 odpowiada ostatniej półkrawędzi wychodzącej z wierzchołka. W pętli w liniach 54–58 obliczane są wektory $\boldsymbol{v}_2$ dla kolejnych półkrawędzi wychodzących z i-tego wierzchołka, po czym iloczyn wektorowy $\boldsymbol{v}_1 \wedge \boldsymbol{v}_2$ jest dodawany do zmiennej `nv` w celu obliczenia sumy. Wektor $\boldsymbol{v}_2$ w następnym przebiegu pętli staje się wektorem $\boldsymbol{v}_1$.

Podobną parę procedur, o nazwach na przykład `GetVertexPos4f` i `MeshVertexNormal4f` należy napisać w celu umożliwienia obliczeń dla siatek, których wierzchołki mają położenia reprezentowane przez współrzędne jednorodne (linia 68). Nie napisałem tych procedur celowo, zostawiając to jako ćwiczenie dla Czytelników. Potrzebne do odnajdywania numerów odpowiednich wierzchołków w siatce instukcje mogą być takie same, a współrzędne wektora normalnego płaszczyzny w $\mathbb{R}^3$, w której leżą punkty (wierzchołki siatki) reprezentowane przez wektory współrzędnych jednorodnych $\boldsymbol{P}_i$, $\boldsymbol{P}_j$, $\boldsymbol{P}_k$ są pierwszymi trzema współrzędnymi iloczynu wektorowego tych trzech wektorów w $\mathbb{R}^4$, $\boldsymbol{P}_i \wedge \boldsymbol{P}_k \wedge \boldsymbol{P}_j$. Można do jego obliczenia użyć funkcji `cross4` z listingu 15.1.

Inna możliwość, to zamiana współrzędnych jednorodnych na kartezjańskie. Mogłoby to dać oszczędność miejsca w pamięci GPU zajmowanego przez wynikową reprezentację i trochę przyspieszyć rysowanie, ale trzeba pamiętać, że wynik ewentualnego dalszego zagęszczania siatki, której wierzchołki mają *różne* współrzędne wagowe byłby inny niż wynik zagęszczania siatki, której wierzchołki mają w $\mathbb{R}^3$ te same położenia reprezentowane przez współrzędne kartezjańskie (albo równoważnie przez współrzędne jednorodne z tą samą współrzędną wagową dla wszystkich wierzchołków).

Listing 33.2 przedstawia procedury, które przygotowują do pracy i likwidują program z opisanym wcześniej szaderem obliczeniowym. Identyfikator programu, bufor bloku zmiennych jednolitych i tablica przesunięć zmiennych względem początku bufora są pamiętane w globalnych zmiennych statycznych, niewidocznych poza plikiem źródłowym zawierającym te procedury i procedurę wywołującą szader z listingu 33.1.

Listing 33.2. Procedury kompilacji i likwidacji programu obliczania wektorów normalnych

C

```
1: static GLuint nvprogid;
2: static GLuint nvbuf, nvbbp;
3: static GLint  nvuvofs[9];
4:
5: void LoadMeshNormalVectorProgram ( void )
6: {
7:   const GLchar *filename[] =
8:     { "mnv.comp.glsl" };
9:   const GLchar *uvnames[] =
10:    { "MeshNV", "innsattr", "pdim", "inpofs", "inv", "inhe", "infac",
11:      "outnsattr", "outpofs", "outnvofs" };
12:  GLuint shader_id;
13:  GLint  size;
14:
15:  shader_id = CompileShaderFiles ( GL_COMPUTE_SHADER, 1, &filename[0] );
16:  nvprogid = LinkShaderProgram ( 1, &shader_id, "meshes normal" );
17:  GetAccessToUniformBlock ( nvprogid, 9, uvnames, &size, nvuvofs,
18:                            &nvbbp );
19:  glGenBuffers ( 1, &nvbuf );
20:  glBindBufferBase ( GL_UNIFORM_BUFFER, nvbbp, nvbuf );
21:  glBufferData ( GL_UNIFORM_BUFFER, size, NULL, GL_DYNAMIC_DRAW );
22:  glDeleteShader ( shader_id );
23:  ExitIfGLError ( "LoadMeshNormalVectorProgram" );
24: } /*LoadMeshNormalVectorProgram*/
25:
26: void DeleteMeshNormalVectorProgram ( void )
27: {
28:  glUseProgram ( 0 );
29:  glDeleteProgram ( nvprogid );
30:  glDeleteBuffers ( 1, &nvbuf );
31:  ExitIfGLError ( "DeleteMeshNormalVectorProgram" );
32: } /*DeleteMeshNormalVectorProgram*/
```

Procedura `LoadMeshNormalVectorProgram` wykonuje rutynowe działania: kompiluje szader i łączy program oraz odczytuje z niego przesunięcia pól w bloku zmiennych jednolitych `MeshNV`; informacje te są zapamiętywane w opakowaniu wskazywanym przez parametr. Ponadto procedura rezerwuje numer punktu dowiązania i tworzy bufor dla tego bloku zmiennych jednolitych; numer punktu i identyfikator bufora są również zapisywane w opakowaniu.

Procedura wywołująca program obliczający wektory normalne dla wierzchołków siatki jest pokazana na listingu 33.3. W liniach 11–14 dokonywana jest rezerwacja bufora na nową tablicę atrybutów wierzchołków siatki. W liniach 18–26 zmiennym jednolitym w bloku `MeshNV` są nadawane odpowiednie wartości, po czym program szaderów przystępuje do pracy — liczba uruchamianych wątków jest liczbą wierzchołków siatki. Po zakończeniu obliczeń

nowy bufor ze współrzędnymi położeń i wektorów normalnych jest „instalowany” w reprezentacji siatki w miejscu bufora dotychczasowego, który w linii 29 zostaje oddany do recyklingu.

Listing 33.3. Procedura obliczania wektorów normalnych

C

```
1: #define SETUVAR(n,type,x)
2:   glBufferSubData ( GL_UNIFORM_BUFFER, uvofs[n], sizeof(type), &x );
3:
4: void ComputeMeshNormalVectors ( GPUmesh *mesh, int nsattr, GLint nvofs )
5: {
6:   GLuint vcbuf;
7:
8:   glBindBufferBase ( SSB, 0, mesh->MVFBUF );
9:   glBindBufferBase ( SSB, 1, mesh->MHEBUF );
10:  glBindBufferBase ( SSB, 2, mesh->VCBUF );
11:  glGenBuffers ( 1, &vcbuf );
12:  glBindBufferBase ( SSB, 3, vcbuf );
13:  glBufferData ( SSB, mesh->nv*nsattr*sizeof(GLfloat), NULL,
14:                 GL_DYNAMIC_DRAW );
15:  ExitIfGLError ( "ComputeMeshNormalVectors 0" );
16:  glUseProgram ( nvprogid );
17:  glBindBufferBase ( GL_UNIFORM_BUFFER, nvbbp, nvbuf );
18:  SETUVAR ( 0, GLint, mesh->nsattr );
19:  SETUVAR ( 1, GLint, mesh->pdim );
20:  SETUVAR ( 2, GLint, mesh->pofs );
21:  SETUVAR ( 3, GLint, mesh->nv );
22:  SETUVAR ( 4, GLint, mesh->nhe );
23:  SETUVAR ( 5, GLint, mesh->nfac );
24:  SETUVAR ( 6, GLint, nsattr );
25:  SETUVAR ( 7, GLint, mesh->pofs );
26:  mesh->nvofs = nvofs;  SETUVAR ( 8, GLint, nvofs );
27:  COMPUTE ( mesh->nv, 1, 1 )
28:  ExitIfGLError ( "ComputeMeshNormalVectors 1" );
29:  glDeleteBuffers ( 1, &mesh->VCBUF );
30:  mesh->VCBUF = vcbuf;
31:  mesh->nsattr = nsattr;
32:  UploadMeshParams ( mesh );
33:  ExitIfGLError ( "ComputeMeshNormalVectors" );
34: } /*ComputeMeshNormalVectors*/
```

33.2. Rysowanie siatki

Opisane w poprzednim rozdziale szadery rysowania siatek trzeba dostosować do zmiany reprezentacji siatki z dodatkowymi atrybutami wierzchołków; chcemy zachować możliwość otrzymania takich samych obrazów jak poprzednio, ale jeśli są podane wektory normalne

wierzchołków, to chcemy ich używać podczas rysowania ścian siatki. Pierwszy program rysowania siatki, którego zadaniem jest wyświetlenie krawędzi, nie wymaga żadnych zmian.

Niezbędna modyfikacja szadera wierzchołków programu rysowania podzielonych na trójkąty ścian siatki jest również niewielka; prawie cała treść szadera jest pokazana na listingu 33.4. Oprócz położenia wierzchołka (w zawsze obecnej zmiennej `gl_Position`) i koloru szader wyprowadza wektor normalny w zmiennej `Normal`. Dla siatki płaskiej, leżącej w płaszczyźnie xy jest to zawsze wektor $(0, 0, 1)$. Jeśli liczba współrzędnych wierzchołka jest równa 3 albo 4, to wyprowadzany jest wektor normalny wzięty z tablicy, ale jeśli wartość zmiennej `nvofs` jest ujemna (co sygnalizuje nieobecność atrybutu — wektora normalnego), to wyprowadzony zostanie wektor zerowy. Nakłada to na szader geometrii dodatkowy obowiązek zbadania, czy wektor normalny jest zerowy, który w takim przypadku powinien w miejscu wektora normalnego powierzchni wyprowadzić wektor normalny płaszczyzny przetwarzanego trójkąta (takiego rozwiązania użyliśmy już w drugiej aplikacji, zobacz listing 15.5).

Listing 33.4. Szader wierzchołków programu rysowania ścian siatki

C

```
.... /* dyrektywę #version i wcześniej opisane makrodefinicje pominąłem */

layout(location=0) out vec3 Normal;
layout(location=1) out vec3 colour;

.... /* bloki magazynowe jak na listingu 31.4 */

#define mfac(I)  mmvf.mvf[nv+(I)]
#define mfhei(I) mmvf.mvf[nv+nfac+nhe+(I)]
#define mhe(I)   mmhe.mhe[I]
#define mvc(I)   mmvc.vc[I]

void main ( void )
{
  int i, j;

  i = mhe(mfhei((mfac(gl_InstanceID) & FHEMASK) + gl_VertexID)).V0;
  j = nsattr*i + pofs;
  switch ( pdim ) {
case 2:
    gl_Position = vec4 ( mvc(j), mvc(j+1), 0.0, 1.0 );
    Normal = vec3 ( 0.0, 0.0, 1.0 );
    return;
case 3:
    gl_Position = vec4 ( mvc(j), mvc(j+1), mvc(j+2), 1.0 );
    break;
default:
    gl_Position = vec4 ( mvc(j), mvc(j+1), mvc(j+2), mvc(j+3) );
    break;
```

```
30:   }
31:   if ( nvofs >= 0 ) {
32:     j = nsattr*i + nvofs;
33:     Normal = vec3 ( mvc(j), mvc(j+1), mvc(j+2) );
34:   }
35:   else
36:     Normal = vec3 ( 0.0 );
37:   colour = Colour;
38: } /*main*/
```

Szader geometrii (przetwarzający jeden trójkąt) jest pokazany na listingu 33.5, przedstawiającym różnice między tym szaderem a szaderem z listingu 32.9. Kolejne pola bloku wyjściowego `Out` reprezentują położenie wierzchołka w układzie współrzędnych świata, kolor, wektor normalny otrzymany od szadera wierzchołków i wektor normalny płaszczyzny trójkąta. W linii 29 następuje obliczanie i normalizacja współrzędnych w układzie świata otrzymanego na wejściu wektora normalnego (danego w układzie modelu). W linii 32 jest obliczany wektor normalny płaszczyzny trójkąta, który później jest przekazywany na wyjście szadera w linii 37. Jeśli otrzymany od szadera wierzchołków wektor normalny jest zerowy (tj. ma długość znikomą, a nie równą 1), to w jego miejscu jest również wyprowadzany wektor normalny płaszczyzny trójkąta. W linii 42 może być zmodyfikowany zwrot tego wektora, aby kąt między oboma wektorami normalnymi był ostry.

Listing 33.5. Szader geometrii programu rysowania ścian siatki

GLSL

```
1: #version 450 core
2:
3: layout(triangles) in;
4: layout(triangle_strip,max_vertices=3) out;
5:
6: layout(location=0) in vec3 Normal[];
7: layout(location=1) in vec3 colour[];
8:
9: out FVertex {
10:     vec3 Colour;
11:     vec3 Position;
12:     vec3 Normal, TNormal;
13:   } Out;
14:
15: uniform TransBlock {
16:     mat4 mm, mmti, vm, pm, vpm;
17:     vec4 eyepos;
18:   } trb;
19:
20: void main ( void )
21: {
22:   int  i;
```

```
23:   vec4 p[3];
24:   vec3 q[3], nvec[3], v1, v2, tnv;
25:
26:   for ( i = 0; i < 3; i++ ) {
27:     p[i] = trb.mm * gl_in[i].gl_Position;
28:     q[i] = p[i].xyz/p[i].w;
29:     nvec[i] = normalize ( mat3(trb.mmti) * Normal[i] );
30:   }
31:   v1 = q[1] - q[0];  v2 = q[2] - q[0];
32:   tnv = normalize ( cross ( v2, v1 ) );
33:   for ( i = 0; i < 3; i++ ) {
34:     gl_Position = trb.vpm * p[i];
35:     Out.Position = q[i];
36:     Out.Colour = colour[i];
37:     Out.tnv = tnv;
38:     if ( Normal[i], Normal[i] ) < 1.0e-10 )
39:       Out.Normal = Out.TNormal = tnv;
40:     else {
41:       Out.Normal = nvec[i];
42:       Out.TNormal = dot ( nvec[i], tnv ) > 0.0 ? tnv : -tnv;
43:     }
44:     EmitVertex ();
45:   }
46:   EndPrimitive ();
47: } /*main*/
```

Do rysowania ścian siatki jest użyty szader fragmentów pokazany na listingu 12.8. Mamy zatem *dwa* wektory normalne: jednostkowy wektor normalny płaszczyzny trójkąta (`In.TNormal`) i wektor normalny powierzchni gładkiej reprezentowanej przez siatkę (`In.Normal`), który trzeba unormować, bo jest on wynikiem interpolacji wektorów jednostkowych o różnych kierunkach. Do rozstrzygnięcia, czy obserwator jest po tej samej stronie powierzchni co źródło światła, trzeba użyć wektora normalnego płaszczyzny trójkąta, a do modelu oświetlenia powierzchni (tj. do wzoru opisującego ten model) trzeba podstawić wektor normalny powierzchni (zobacz s. 287).

Dostosowanie procedury `LoadMeshRenderingShaders` polega na zmienieniu nazw plików z tekstami źródłowymi szaderów, z których składa się program rysujący ściany siatki. Aby móc wybierać, czy w modelu oświetlenia mają być używane wektory normalne podane jako atrybuty wierzchołków siatki (czego skutkiem jest powstanie obrazu powierzchni gładkiej), czy wektory normalne trójkątów, trzeba zmienić procedurę `DrawMeshFacets` w sposób pokazany na listingu 33.6 — dodatkowy parametr steruje wybieraniem wektorów. Procedura `SetMeshNVS` przypisuje jego wartość polu `MeshNormals` bloku magazynowego `meshsurf` opisującego siatkę. Wywołanie procedury `glProvokingVertex` usunąłem, ale można je zostawić i nadać kwalifikator `flat` polu `TNormal` bloku interfejsu `FVertex`, aby wyeliminować niepotrzebne obliczenia związane z interpolacją wektora normalnego płaszczyzny trójkąta.

Listing 33.6. Zmiany w procedurze rysowania ścian siatki

C

```
void DrawMeshFacets ( MeshRenderPrograms *prog,
                      GPUmesh *mesh, GLfloat colour[3], char nvs )
{
  int i;

  for ( i = 0; i < 4; i++ )
    glBindBufferBase ( SSB, i, mesh->mbuf[i] );
  SetMeshColour ( mesh, colour );
  SetMeshNVS ( mesh, (GLint)nvs );
  glUseProgram ( prog->progid[1] );
  glBindVertexArray ( empty_vao );
  glDrawArraysInstanced ( GL_TRIANGLE_FAN, 0, 4, mesh->nfac );
  glBindVertexArray ( 0 );
  ExitIfGLError ( "DrawMeshFacets" );
} /*DrawMeshFacets*/

void SetMeshNVS ( GPUmesh *gmesh, GLint nvs )
{
  glBindBuffer ( SSB, gmesh->MSBUF );
  glBufferSubData ( SSB, mbofs[7], sizeof(GLint), &nvs );
  ExitIfGLError ( "SetMeshNVS" );
} /*SetMeshNVS*/
```

33.3. Zmiany w aplikacji

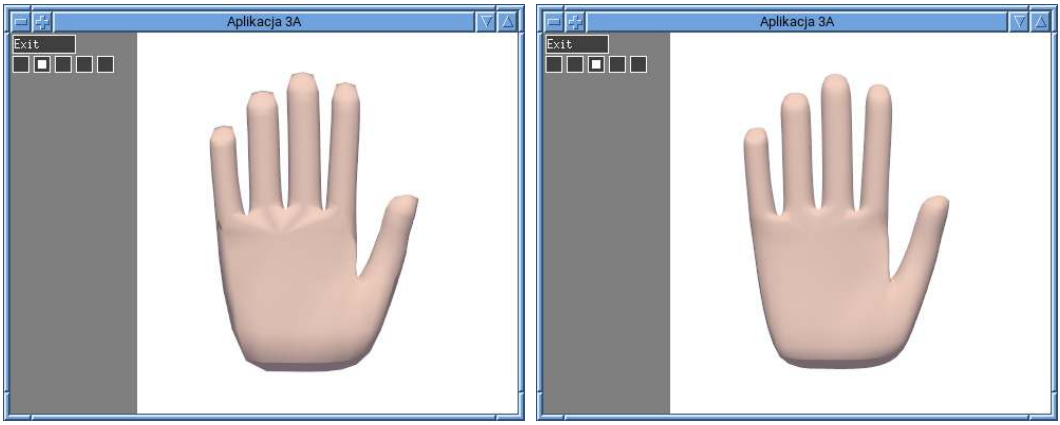
Do procedury `InitMyWorld` pokazanej na listingu 32.14 trzeba dodać wywołanie procedury `LoadMeshNormalVectorShader` z listingu 33.2, z kolei procedura sprzątająca `DeleteMyWorld` ma wywołać procedurę `DeleteMeshNormalVectorProgram`. Procedurę przygotowującą siatki trzeba uzupełnić o instrukcję, która oblicza wektory normalne dla wierzchołków siatek (listing 33.7). Zwróćmy uwagę, że procedurę `ComputeMeshNormalVectors` wykonujemy *po* otrzymaniu przez zagęszczanie wszystkich siatek — w ten sposób zagęszczamy siatkę, której wierzchołki mają tylko atrybut położenia opisany przez trójkę liczb (współrzędnych kartezjańskich), a obliczenie wektorów normalnych (które wprowadza dodatkowe atrybuty) jest przeprowadzane, gdy reprezentacja siatki w pamięci GPU jest potrzebna już tylko do wykonywania obrazów.

Pozostałe zmiany, które aplikację 3 zamieniły w 3A, to dodanie pola `mnv` typu `char` do struktury `AppData`, instrukcji (w procedurze `ProcessCharCommand`) nadających temu polu wartości `true` i `false` w odpowiedzi na naciskanie klawisza z literą `N` i przekazanie wartości tego pola jako dodatkowego parametru w wywołaniu procedury `DrawMeshFacets`.

Listing 33.7. Zmiany w procedurze `InitPalmMeshes`

C

```
void InitPalmMeshes ( AppData *ad )
{
  static const GLfloat edges_colour[3] = {0.0,0.5,0.7};
  static const GLfloat facets_colour[3] = {0.91,0.65,0.5};
  KLMesh *palm;
  int i;

  palm = &ad->palm;
  if ( (palm->mesh[0] = EnterPalmToGPU ()) ) {
    for ( i = 1; i < NPALMMESHES; i++ ) {
      .... /* tu instrukcje bez zmian */
    }
    for ( i = 1; i < NPALMMESHES; i++ )
      ComputeMeshNormalVectors ( palm->mesh[i] );
  }
  else
    ExitOnError ( "InitPalmMeshes" );
} /*InitPalmMeshes*/
```



Rysunek 33.1. Okno aplikacji trzeciej A

33.4. Ćwiczenia

1. Wykonaj obliczenie wektorów normalnych dla pewnej siatki, a następnie dokonaj zagęszczenia tej siatki prowadzącego do obliczenia wektorów normalnych przez interpolację (tak jak obliczane są położenia wierzchołków podczas zagęszczania). Porównaj obrazy otrzymane przy użyciu tych wektorów z obrazami wyświetlanymi przez aplikację opisaną w tym rozdziale.

2. Rozszerz zestaw możliwych atrybutów wierzchołków siatki o kolor, zmień aplikację tak, by radziła sobie z siatkami, w których wektor normalny i kolor są obecne lub nieobecne i użyj do wykonywania obrazów atrybutów dowolnie wybieranych spośród atrybutów obecnych w danej reprezentacji siatki.

34

Aplikacja trzecia B

Utworzymy łańcuch kinematyczny, z którego członami związemy punkty kontrolne siatki dłoni, umożliwiając poruszanie palcami. Po zmianie dowolnego parametru artykulacji aplikacja przy użyciu szadera obliczeniowego z listingu 23.1 obliczy wierzchołki odkształconej siatki, po czym zagęści tę siatkę i narysuje powierzchnię.

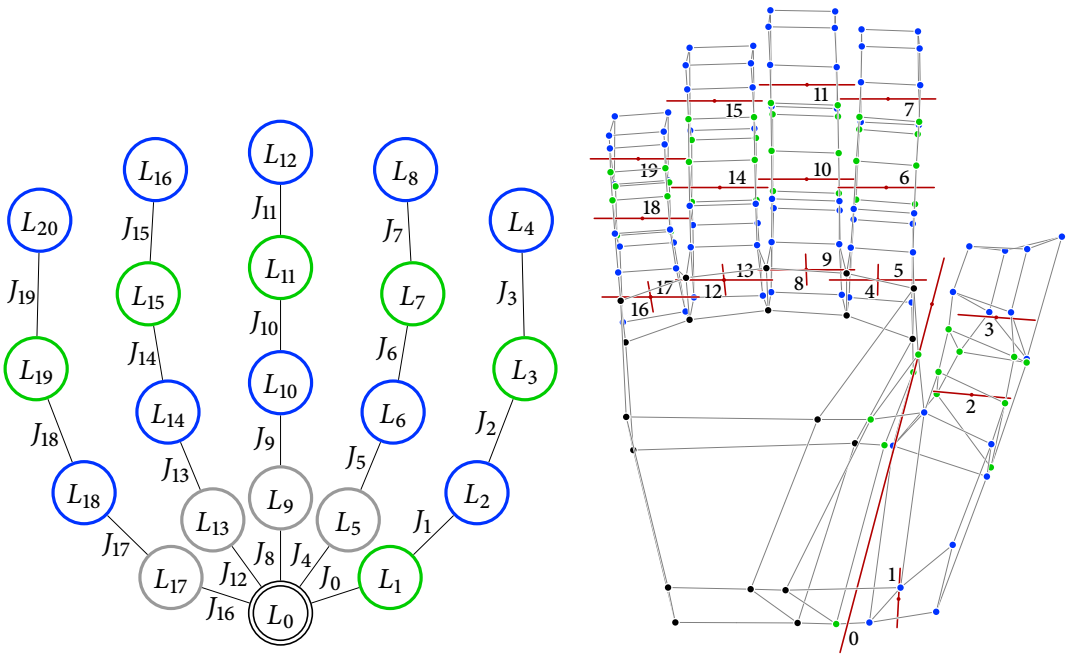
34.1. Łańcuch kinematyczny

Graf łańcucha kinematycznego dłoni jest drzewem, którego „gałęzie" reprezentują palce. Gałęzie na rysunku 34.1 od prawej do lewej reprezentują kolejno kciuk, palec wskazujący, środkowy, serdeczny i mały. Każda gałąź ma cztery krawędzie — pary kinematyczne odpowiadające poszczególnym stawom, przy czym pierwsze dwie pary wszystkich palców oprócz kciuka (np. pary J_4 i J_5 palca wskazującego) odpowiadają temu samemu stawowi; dzięki temu, choć wszystkie pary w łańcuchu są proste, pierwszy staw każdego z tych palców ma dwa stopnie swobody.

Wszystkie pary kinematyczne w tym łańcuchu są obrotowe; razem z siatką na rysunku są uwidocznione osie obrotów realizowanych przez te pary, a przy każdej osi jest podany numer pary. Ponadto wierzchołki siatki związane z poszczególnymi członami łańcucha są oznaczone kolorami użytymi także do przedstawienia członów — wierzchołków grafu z lewej strony. Zbiór wierzchołków związanych z członami L_5, L_9, L_{13} i L_{17} jest pusty.

Listing 34.1 przedstawia makrodefinicje i definicje typów strukturalnych potrzebne do zbudowania łańcucha kinematycznego w aplikacji. Struktura typu `KLMesh` opisuje obiekt — reprezentowaną przez siatkę powierzchnię będącą modelem dłoni. W nowym polu `tribuf` tej struktury będzie pamiętany identyfikator bufora z tablicą, w której dla każdego wierzchołka siatki znajduje się numer przekształcenia (indeks do tablicy macierzy przekształceń opisujących przejście od układu współrzędnych członu łańcucha do układu modelu), któremu ma być poddany ten wierzchołek, aby odkształcić siatkę.

Struktura typu `KLAppData` opisuje dane stanowiące część opisu łańcucha kinematycznego i umożliwiające narysowanie odkształconego za jego pomocą obiektu. Jej pole `wdg`,



Rysunek 34.1. Łańcuch kinematyczny dłoni

będące strukturą typu `AppWidgets`, zostało rozszerzona o tablicę `artp`, której elementy są „podłączone” do suwaków w menu. Zmienne te, przyjmujące wartości z przedziału $[0,1]$, są używane do obliczenia parametrów artykulacji łańcucha, tj. kątów ugięcia poszczególnych stawów.

Tablica `mesh` została wydłużona o jedno miejsce. Pierwszy jej element będzie wskazywał strukturę reprezentującą siatkę nieodkształconą, a drugi siatkę odkształconą, tj. otrzymaną w wyniku artykulacji. Zagęszczaniu będzie poddawana siatka odkształcona, otrzymane w ten sposób siatki będą wskazywane przez pozostałe elementy tablicy.

Do struktury typu `AppData` zostały dodane pola `linkage`, `lktrbuf` i `artprog`. Pierwsze z nich jest wskaźnikiem struktury łańcucha, której pole `usrdata` będzie wskazywać zmienną typu `AppData`. Pole `lktrbuf` służy do przechowania identyfikatora bufora z macierzami przekształceń, którym będą poddawane poszczególne wierzchołki siatki. Pole `artprog` jest opakowaniem programu artykulacji łańcucha kinematycznego, tego samego, który był użyty w aplikacji 2H (zobacz listingi 23.1, 23.2 i 23.3).

Listing 34.1. Makrodefinicje i struktura reprezentująca scenę

C

```
#define MESHDEG         3  /* liczba kroków uśredniania */
#define NPALMMESHES     4  /* liczba zagęszczonych siatek */
#define NKLOBJ          1  /* liczba obiektów w łańcuchu */
#define NKLINKS        21  /* liczba członów */
#define NKLREFS        17  /* liczba referencji */
```

```
6: #define NKLJOINTS     20  /* liczba par kinematycznych */
7: #define NKLARTPARAMS  20  /* liczba parametrów artykulacji */
8:
9: typedef struct {
10:     char  sw[NPALMMESHES+1];
11:     float artp[NKLARTPARAMS];
12:     char  animation;
13:   } AppWidgets;
14:
15: typedef struct {
16:     GPUmesh *mesh[NPALMMESHES+2];
17:     GLfloat ecolour[3], fcolour[3];
18:     GLuint  tribuf;
19:   } KLMesh;
20:
21: typedef struct {
22:     AppWidgets            wdg;
23:     KLMesh                palm;
24:     kl_linkage            *linkage;
25:     Camera                camera;
26:     TransBl               trans;
27:     LightBl               light;
28:     GLuint                lktrbuf;
29:     char                  lod, edges, mnv;
30:     float                 speed;
31:     float                 model_rot_axis[3];
32:     double                model_rot_angle;
33:     MeshRenderPrograms    mrprog;
34:     KLArticulationProgram artprog;
35:   } AppData;
```

Listing 34.2 przedstawia w skrócie procedurę `ConstructPalmLinkage`. Jej zadaniem jest skonstruowanie łańcucha kinematycznego opisującego chwytną dłoń. Wywoływane przez nią procedury pomocnicze i metody obiektu — dłoni są przedstawione na kolejnych listingach. Parametr procedury `ConstructPalmLinkage` jest adresem struktury opisującej scenę.

Wywołana w liniach 22–23 procedura `kl_NewLinkage` rezerwuje pamięć na łańcuch kinematyczny o podanych limitach liczb tworzących go elementów. W linii 24 do tablicy zmiennych sterowanych przez suwaki są wpisywane liczby (zapisane w tablicy `palmartp0`), które określą wartości początkowe parametrów artykulacji.

Człony łańcucha są wprowadzane w pętli w liniach 25–26, po czym procedura `kl_NewObject` wprowadza obiekt — siatkę dłoni, rejestrując podane jako parametry metody tego obiektu i wywołując procedurę `KLInitPalmMesh`, czyli konstruktor, który przesyła reprezentację siatki do pamięci GPU i inicjalizuje dane potrzebne do jej przetwarzania (tj. zagęszczania i rysowania).

Listing 34.2. Procedura `ConstructPalmLinkage`

C

```
kl_linkage *ConstructPalmLinkage ( AppData *ad )
{
  static int jtype[NKLJOINTS] =
    { KL_ART_ROT_Y, KL_ART_ROT_Z, KL_ART_ROT_X, KL_ART_ROT_X,
      KL_ART_ROT_Z, KL_ART_ROT_X, KL_ART_ROT_X, KL_ART_ROT_X,
      KL_ART_ROT_Z, KL_ART_ROT_X, KL_ART_ROT_X, KL_ART_ROT_X,
      KL_ART_ROT_Z, KL_ART_ROT_X, KL_ART_ROT_X, KL_ART_ROT_X,
      KL_ART_ROT_Z, KL_ART_ROT_X, KL_ART_ROT_X, KL_ART_ROT_X };
  static int jpnum[NKLARTPARAMS] =
    {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19};
  static float artp0[NKLARTPARAMS] =
      {0.0,0.0,0.0,0.0,0.2,0.0,0.0,0.0,0.5,0.0,0.0,
       0.0,0.5,0.0,0.0,0.0,0.8,0.0,0.0};
  static float pp0[3] = {-0.2, -0.86, 0.5},
               vp0[3] = {0.0,1.0,0.0}, vp1[3] = {0.22,0.83,0.1};

  kl_linkage *lkg;
  int        lnk[NKLINKS], jnt[NKLJOINTS];
  GLfloat    tra[16];
  int        i, j, k, l;

  if ( (ad->linkage = lkg = kl_NewLinkage ( NKLOBJ, NKLINKS, NKLREFS,
                              NKLJOINTS, NKLARTPARAMS, (void*)ad )) ) {
    memcpy ( ad->artp, artp0, NKLARTPARAMS*sizeof(float) );
    for ( i = 0; i < NKLINKS; i++ )
      lnk[i] = kl_NewLink ( lkg );
    kl_NewObject ( lkg, 0, 3, PALM_NV, NULL, (void*)&ad->palm,
                   KLInitPalmMesh, KLTransformVertices,
                   KLPostprocessMesh, KLRedrawMesh, KLDeletePalmMesh );
    for ( k = j = 0;  k < 5;  k++ )
      for ( i = 0, l = -1;  i < 4;  i++, l = j++ )
        jnt[j] = kl_NewJoint ( lkg, lnk[l+1], lnk[j+1],
                               jtype[j], jpnum[j] );
        /* kciuk */
    M4x4RotateP2Vf ( tra, pp0, vp0, vp1 );
    kl_SetJointFtr ( lkg, jnt[0], tra, true );
    M4x4Translatef ( tra, 0.125, -0.842, 0.0 );
    kl_SetJointFtr ( lkg, jnt[1], tra, true );
    M4x4Translatef ( tra, 0.329, -0.275, 0.05 );
    M4x4MRotateZf ( tra, -0.1*PI );
    M4x4MRotateYf ( tra, -0.2*PI );
    kl_SetJointFtr ( lkg, jnt[2], tra, true );
    M4x4Translatef ( tra, 0.4, -0.05, 0.05 );
    M4x4MRotateZf ( tra, -0.1*PI );
    M4x4MRotateYf( tra, -0.2*PI );
```

```
46:     kl_SetJointFtr ( lkg, jnt[3], tra, true );
47:         /* wskazujący */
48:     M4x4Translatef ( tra, 0.070, 0.05, 0.08 );
49:     kl_SetJointFtr ( lkg, jnt[4], tra, true );
50:     kl_SetJointFtr ( lkg, jnt[5], tra, true );
51:     M4x4Translatef ( tra, 0.094, 0.324, 0.08 );
52:     kl_SetJointFtr ( lkg, jnt[6], tra, true );
53:     M4x4Translatef ( tra, 0.1, 0.591, 0.08 );
54:     kl_SetJointFtr ( lkg, jnt[7], tra, true );
55:         /* podobnie środkowy, serdeczny i mały */
56:     ....
57:     glGenBuffers ( 1, &ad->lktrbuf );
58:     glBindBuffer ( GL_SHADER_STORAGE_BUFFER, ad->lktrbuf );
59:     glBufferData ( GL_SHADER_STORAGE_BUFFER, lkg->norefs*16*sizeof(GLfloat),
60:                    NULL, GL_DYNAMIC_DRAW );
61:   }
62:   return lkg;
63: } /*ConstructPalmLinkage*/
```

W liniach 30–33 do łańcucha są dodawane pary kinematyczne; z uwagi na dosyć prostą budowę grafu łańcucha (wszystkie „gałęzie" mają w nim tyle samo par), zamiast „wyliczać" każdą parę osobno, można to zrobić w podwójnej pętli. Rodzaje kolejnych par (tj. określenia osi obrotu dla każdej pary) są brane z tablicy `jtype`, z kolei w tablicy `jpnum` są podane numery parametrów artykulacji. W tym przypadku, ponieważ wszystkie pary są proste, numery te są kolejnymi liczbami całkowitymi.

Po utworzeniu par kinematycznych trzeba jeszcze dla każdej z nich określić macierze stałe, F_i oraz B_i. W każdym przypadku jest $B_i = F_i^{-1}$, dzięki czemu w położeniu wyjściowym (w którym kąty obrotów wszystkich par są równe 0) układy współrzędnych wszystkich członów pokrywają się i siatka nie jest odkształcona (zobacz podrozdz. 13.1). Przekształcenia opisane przez te macierze mają na celu końcowe określenie osi obrotu pary, przez podanie (dowolnego) punktu tej osi i ewentualne zmodyfikowanie jej kierunku. Dla wszystkich palców z wyjątkiem kciuka osie obrotów są równoległe do osi z i x, a odpowiednie macierze F_i są macierzami przesunięć. Na przykład dla palca wskazującego osie obrotów par J_4 i J_5 przechodzą przez punkt $(0.07, 0.05, 0.08)$, oś pary J_6 przechodzi przez punkt $(0.094, 0.324, 0.08)$, a para J_7 realizuje obrót wokół osi przechodzącej przez punkt $(0.1, 0.591, 0.08)$. Macierze $F_4 = F_5$, F_6 i F_7 są konstruowane przez instrukcje w liniach 48, 51 i 53.

Osie obrotów niektórych par kinematycznych kciuka nie są równoległe do osi układów współrzędnych. Na przykład kierunek osi obrotu pary J_0 jest wyznaczony przez macierz F_0, która opisuje obrót wokół przechodzącej przez punkt $\boldsymbol{p}_0 = (-0.2, -0.86, 0.5)$ osi prostopadłej do wektorów $\boldsymbol{v}_0 = (0, 1, 0)$ i $\boldsymbol{v}_1 = (0.22, 0.83, 0.1)$, przy czym kąt tego obrotu jest dobrany tak, aby obraz wektora $\boldsymbol{v}_0$ miał kierunek i zwrot wektora $\boldsymbol{v}_1$. Użyta w linii 35 do obliczenia tej macierzy procedura `M4x4RotateP2Vf` jest zamieszczona na listingu 5.3. Macierz F_1 reprezentuje tylko przesunięcie, z kolei macierz $F_2 = TR_1R_2$ opisuje złożenie trzech przekształceń: przesunięcia T o wektor $(0.329, -0.275, 0.05)$, obrotu R_1 wokół osi z i obrotu R_2 wokół osi y.

W liniach 57–60 jest tworzony bufor w pamięci GPU z tablicą, do której procedura KLTransformVertices, będąca metodą obiektu — siatki, będzie wpisywać macierze przekształceń obliczone przez procedurę artykulacji łańcucha.

Listing 34.3 przedstawia metody siatki. Konstruktor obiektu, czyli procedura KLInitPalmMesh, ma przygotować wszystkie dane potrzebne do artykulacji i do rysowania obiektu. Zatem, w linii 26 (oryginalna) siatka dłoni jest przesyłana do pamięci GPU. Pętla w liniach 27–31 wprowadza referencje obiektu, wiążące odpowiednie podzbiory zbioru wierzchołków z członami łańcucha, oraz zapisuje (w roboczej tablicy cpi) dla każdego wierzchołka numer referencji tego wierzchołka — jest to numer przekształcenia, któremu wierzchołek będzie poddany w procesie artykulacji. Parametr procedury kl_NewObjRef określający liczbę wierzchołków tworzonej referencji jest równy 0, ponieważ przekstałcanie wierzchołków wykona szader obliczeniowy wywołany przez procedurę KLPostprocessMesh. W liniach 32–34 jest tworzony bufor, do którego przesyłana jest zawartość tablicy roboczej.

W liniach 36–40 jest rezerwowana pamięć na opisy siatki odkształconej i siatek zagęszczonych. Ponieważ siatka oryginalna i siatka odkształcona będą różnić się tylko położeniami wierzchołków, w linii 41 dane opisujące siatkę oryginalną są kopiowane do struktury siatki odkształconej, po czym w liniach 42–45 jest rezerwowany nowy bufor, w którym będą przechowywane wierzchołki siatki odkształconej. W liniach 46 i 47 w strukturze opisującej siatkę są zapisywane kolory, jakimi mają być rysowane krawędzie siatki odkształconej i ściany siatek otrzymanych z jej zagęszczania.

Listing 34.3. Metody obiektu — siatki dłoni

C

```
1:  static char KLInitPalmMesh ( kl_linkage *lkg, kl_object *obj )
2:  {
3:    static const GLfloat edges_colour[3] = {0.0,0.5,0.7};
4:    static const GLfloat facets_colour[3] = {0.91,0.65,0.5};
5:    static GLint r0[144] =
6:     {0,1,2,3,6,7,8,9,10,13,16,17,135,138,139,140,141,142,143,
7:       4,11,134,136,137, 5,12,14,15,18,19,20,21, 22,23,24,25,26,27,28,29,
8:      30,31,32,33,34,35,36,37, 38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,
9:      53,54,55,56,57,58,59,60,61, 62,63,64,65,66,67,68,69, 70,71,72,73,74,75,
10:     76,77,78,79,80,81,82,83,84,85, 86,87,88,89,90,91,92,93, 94,95,96,97,98,
11:     99,100,101,102,103,104,105,106,107,108,109, 110,111,112,113,114,115,116,
12:     117,118,119,120,121,122,123,124,125, 126,127,128,129,130,131,132,133};
13:   static const int r1[17] = {19,5,8,8,8,8,8,8, 8, 8, 8, 8, 8, 8, 8, 8, 8};
14:   static const int r2[17] = { 0,1,2,3,4,6,7,8,10,11,12,14,15,16,18,19,20};
15:   KLMesh  *md;
16:   GPUmesh **palms;
17:   GLuint  *cpi;
18:   int     on, rn, nv, i, j, k;
19:
20:   on = obj - lkg->obj;
21:   nv = obj->nvert;
22:   if ( (cpi = malloc ( nv*sizeof(GLuint) )) ) {
```

```
23:     memset ( cpi, 0, nv*sizeof(GLuint) );
24:     md = (KLMesh*)obj->usrdata;
25:     palms = md->mesh;
26:     palms[0] = EnterPalmToGPU ();
27:     for ( j = k = 0;  j < 17;  k += r1[j++] ) {
28:       rn = kl_NewObjRef ( lkg, r2[j], on, 0, NULL );
29:       for ( i = 0; i < r1[j]; i++ )
30:         cpi[r0[k+i]] = rn;
31:     }
32:     md->tribuf = NewStorageBuffer ( nv*sizeof(GLuint),
33:                                     ad->artprog.ctribp );
34:     glBufferData ( SSB, nv*sizeof(GLuint), cpi, GL_STATIC_DRAW );
35:     free ( cpi );
36:     for ( i = 1; i <= NPALMMESHES+1; i++ ) {
37:       if ( !(palms[i] = malloc ( sizeof(GPUmesh) )) )
38:         ExitOnError ( "KLInitPalmMesh" );
39:       memset ( palms[i], 0, sizeof(GPUmesh) );
40:     }
41:     memcpy ( palms[1], palms[0], sizeof(GPUmesh) );
42:     glGenBuffers ( 1, &palms[1]->mbuf[2] );
43:     glBindBuffer ( SSB, palms[1]->mbuf[2] );
44:     glBufferData ( SSB, PALM_NV*3*sizeof(GLfloat),
45:                    NULL, GL_DYNAMIC_DRAW );
46:     memcpy ( md->ecolour, edges_colour, 3*sizeof(GLfloat) );
47:     memcpy ( md->fcolour, facets_colour, 3*sizeof(GLfloat) );
48:   }
49:   else
50:     ExitOnError ( "KLInitPalmMesh" );
51:   return true;
52: } /*KLInitPalmMesh*/
53:
54: static void KLDeletePalmMesh ( kl_linkage *lkg, kl_object *obj )
55: {
56:   KLMesh  *md;
57:   int     i;
58:
59:   md = (KLMesh*)obj->usrdata;
60:   for ( i = 0; i <= NPALMMESHES+1; i++ )
61:     DeleteGPUmesh ( md->mesh[i] );
62:   glDeleteBuffers ( 1, &md->tribuf );
63: } /*KLDeletePalmMesh*/
64:
65: static void KLTransformVertices ( kl_linkage *lkg, kl_object *obj,
66:                                   int refn, GLfloat *tr, int nv, int *vn )
67: {
68:   AppData *ad;
69:
```

```
70:   ad = (AppData*)lkg->usrdata;
71:   glBindBuffer ( GL_UNIFORM_BUFFER, ad->lktrbuf );
72:   glBufferSubData ( GL_UNIFORM_BUFFER, refn*16*sizeof(GLfloat),
73:                     16*sizeof(GLfloat), tr );
74:   ExitIfGLError ( "KLTransformVertices" );
75: } /*KLTransformVertices*/
76:
77: static void KLPostprocessMesh ( kl_linkage *lkg, kl_object *obj )
78: {
79:   AppData *ad;
80:   KLMesh  *md;
81:   KLArticulationProgram *prog;
82:   GPUmesh **mesh;
83:   int     i;
84:
85:   ad = (AppData*)lkg->usrdata;
86:   prog = &ad->artprog;
87:   md = (KLMesh*)obj->usrdata;
88:   mesh = md->mesh;
89:   glUseProgram ( prog->progid );
90:   glBindBufferBase ( SSB, prog->ctrbp, ad->lktrbuf );
91:   glBindBufferBase ( SSB, prog->ctribp, md->tribuf );
92:   glBindBufferBase ( SSB, prog->cpibp, mesh[0]->mbuf[2] );
93:   glBindBufferBase ( SSB, prog->cpobp, mesh[1]->mbuf[2] );
94:   glUniform1i ( prog->dim_loc, obj->nvc );
95:   glUniform1i ( prog->trnum_loc, -1 );
96:   glUniform1i ( prog->ncp_loc, (GLint)obj->nvert );
97:   COMPUTE ( obj->nvert, 1, 1 )
98:   if ( ad->lod >= 1 ) {
99:     for ( i = 1; i <= ad->lod; i++ ) {
100:      if ( !GPUmeshRefinement ( MESHDEG, mesh[i], mesh[i+1] ) )
101:        ExitOnError ( "KLPostprocessMesh" );
102:    }
103:    ComputeMeshNormalVectors ( &ad->mcnprog, mesh[ad->lod+1], 6, 3 );
104:  }
105:  ExitIfGLError ( "KLPostprocessMesh" );
106: } /*KLPostprocessMesh*/
107:
108: static void KLRedrawMesh ( kl_linkage *lkg, kl_object *obj )
109: {
110:   AppData *ad;
111:   KLMesh  *md;
112:
113:   ad = (AppData*)lkg->usrdata;
114:   md = (KLMesh*)obj->usrdata;
115:   if ( ad->meshsw[0] )
116:     DrawMeshEdges ( &ad->mrprog, md->mesh[1], md->ecolour );
```

```
117:   if ( ad->lod >= 1 ) {
118:     if ( ad->edges )
119:       DrawMeshEdges ( &ad->mrprog, md->mesh[ad->lod+1], md->fcolour );
120:     else
121:       DrawMeshFacets ( &ad->mrprog, md->mesh[ad->lod+1], md->fcolour,
122:                        ad->mnv );
123:   }
124: } /*KLRedrawMesh*/
```

Procedura `KLDeletePalmMesh` będzie wywoływana podczas sprzątania — jej zadaniem jest likwidacja reprezentacji siatek w pamięci GPU (i opisujących je struktur w pamięci GPU) oraz zwolnienie bufora z numerami przekształceń wierzchołków.

Procedura `KLTransformVertices`, wywoływana przez procedurę artykulacji łańcucha, zamiast przekształcać wierzchołki siatki, przesyła podaną macierz przekształcenia do bufora zarezerwowanego przez procedurę `ConstructPalmLinkage` (listing 34.2, linie 57–60). Przekształcaniem wierzchołków zajmuje się procedura `KLPostprocessMesh`, która (podobnie jak procedura `KLPostprocessBP` z listingu 23.7) przywiązuje do odpowiednich punktów dowiązania bufor z macierzami przekształceń (linia 90), bufor z numerami przekształceń dla poszczególnych wierzchołków (linia 91), bufor ze współrzędnymi wierzchołków siatki oryginalnej (linia 92) i bufor na przekształcone wierzchołki (linia 93). W liniach 94–96 zmiennym jednolitym `dim`, `trnum` i `ncp` zostają nadane odpowiednie wartości, po czym wykonywany jest program artykulacji.

Po zakończeniu jego działania (czyli po powrocie z procedury `glMemoryBarrier` wywołanej przez makrodefinicję `COMPUTE`) następuje zagęszczanie odkształconej siatki. Wartość pola `lod` struktury `*ad`, jeśli jest dodatnia, jest wybranym przez użytkownika poziomem szczegółowości obrazu, tj. liczbą iteracji zagęszczania. Ściany siatki będącej wynikiem ostatniego zagęszczania mają być narysowane; opisana w poprzednim rozdziale procedura `ComputeMeshNormalVectors` oblicza wektory normalne, które będą użyte do „optycznego wygładzenia" powierzchni na obrazie.

Ostatnia metoda, `KLRedrawMesh`, jest wywoływana przez procedurę `kl_Redraw` (listing 13.9). Procedura ta, zależnie od stanu przełączników w menu, rysuje krawędzie siatki odkształconej i krawędzie albo ściany siatki zagęszczonej określonej przez wartość pola `lod`.

Procedury na listingu 34.4 służą do wprowadzenia jednego lub wszystkich parametrów artykulacji. Każdy staw w dłoni ma pewien, w ogólności inny, zakres kątów, a suwaki (opisane w rozdziale 30) „dostarczają" liczby z przedziału $[0,1]$ (przechowywane w tablicy `ad->wdg.artp`, której elementy są „podłączone" do poszczególnych suwaków). Dlatego każda z tych procedur odwzorowuje ten przedział na odpowiednie przedziały dla poszczególnych stawów (par kinematycznych), na podstawie liczb podanych w tablicy `palmartprange`. Drugim parametrem procedury `SetArticulationParameter` jest numer (jednego) parametru artykulacji. Procedura `ArticulatePalmLinkage` po obliczeniu wartości wszystkich parametrów artykulacji dokonuje artykulacji łańcucha, wywołując procedurę `kl_Articulate`.

Listing 34.4. Procedury obsługi parametrów artykulacji

C

```
static float palmartprange[NKLARTPARAMS][2] =
  {{0.0,-0.5*PI},{0.0,-0.5},{0.0,0.4*PI},{0.0,0.4*PI},
   {0.025*PI,-0.1*PI},{0.0,0.5*PI},{0.0,0.5*PI},{0.0,0.5*PI},
   {0.025*PI,-0.025*PI},{0.0,0.5*PI},{0.0,0.5*PI},{0.0,0.5*PI},
   {0.025*PI,-0.025*PI},{0.0,0.5*PI},{0.0,0.5*PI},{0.0,0.5*PI},
   {0.12*PI,-0.03*PI},{0.0,0.5*PI},{0.0,0.5*PI},{0.0,0.5*PI}};

void SetArticulationParameter ( AppData *ad, int pnum )
{
  float x, par;

  x = ad->artp[pnum];
  par = (1.0-x)*artprange[pnum][0] + x*artprange[pnum][1];
  kl_SetArtParam ( ad->linkage, pnum, 1, &par );
} /*SetArticulationParameter*/

void ArticulatePalmLinkage ( AppData *ad )
{
  float   x;
  GLfloat par[NKLARTPARAMS];
  int     i;

  for ( i = 0; i < NKLARTPARAMS; i++ ) {
    x = ad->artp[i];
    par[i] = (1.0-x)*artprange[i][0] + x*artprange[i][1];
  }
  kl_SetArtParam ( ad->linkage, 0, NKLARTPARAMS, par );
  kl_Articulate ( ad->linkage );
} /*ArticulatePalmLinkage*/
```

34.2. Przygotowanie i rysowanie sceny

W procedurze `InitMyWorld` potrzebne są tylko dwie zmiany (porównaj listing 34.5 z 32.14): trzeba dodać wywołanie procedury `LoadLinkageArticulationProgram`, która kompiluje program artykulacji łańcucha kinematycznego i zastąpić wywołanie procedury `InitPalmMeshes` wywołaniem procedury `ConstructPalmLinkage`, a po niej `ArticulatePalmLinkage`. Aby poprawić czytelność kodu, przeniosłem wywołania wszystkich procedur kompilujących szadery do osobnej procedury `LoadMyShaders`.

Inaczej niż w aplikacjach 2H–2K, które wyświetlają obiekty (czajnik, torus i lustro) „z pominięciem" procedury `kl_Redraw`, aplikacja 3B wywołuje tę procedurę, a ona w pętli wywołuje metody rysowania wszystkich obiektów (czyli jednego; metodą tą jest procedura `KLRedrawMesh` z listingu 34.3). Przedtem trzeba tylko skasować tło i uaktywnić test widoczności.

Listing 34.5. Procedury `InitMyWorld` i `RedrawMyWorld`

C

```
void LoadMyShaders ( AppData *ad )
{
  LoadMeshRefinementProgram ( true, false );
  LoadMeshNormalVectorProgram ();
  LoadMeshRenderingPrograms ( &ad->mrprog );
  LoadLinkageArticulationProgram ( &ad->artprog );
} /*LoadMyShaders*/

AppWidgets *InitMyWorld ( int argc, char *argv[], int width, int height )
{
  static const float model_rot_axis[3] = {0.0,1.0,0.0};

  memset ( &appdata, 0, sizeof(AppData) );
  LoadMyShaders ( &appdata );
  ConstructEmptyVAO ();
  appdata.trans.trbuf = NewUniformTransBlock ();
  appdata.light.lsbuf = NewUniformLightBlock ();
  TimerInit ();
  memcpy ( appdata.model_rot_axis, model_rot_axis, 3*sizeof(float) );
  appdata.speed = 0.5*3.1415926;
  SetupModelMatrix ( &appdata );
  InitCamera ( &appdata, width, height );
  InitLights ( &appdata );
  appdata.mnv = true;
  appdata.wdg.sw[0] = appdata.wdg.sw[2] = true;
  appdata.lod = 2;
  appdata.edges = appdata.wdg.animation = false;
  if ( ConstructPalmLinkage ( &appdata ) )
    ArticulatePalmLinkage ( &appdata );
  else
    ExitOnError ( "InitMyWorld" );
  return &appdata.wdg;
} /*InitMyWorld*/

void DrawMyScene ( AppData *ad )
{
  glClearColor ( 1.0, 1.0, 1.0, 1.0 );
  glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glEnable ( GL_DEPTH_TEST );
  kl_Redraw ( ad->lkg );
} /*DrawMyScene*/

void RedrawMyWorld ( void )
{
  DrawMyScene ( &appdata );
```

```
46: } /*RedrawMyWorld*/
```

Listing 34.6 przedstawia procedurę dodaną do interfejsu części graficznej i okienkowej; jest ona wywoływana po zmianie położenia suwaka. Procedura wywołuje procedurę obliczającą nową wartość parametru artykulacji i procedurę dokonującą artykulacji łańcucha, a potem przekazuje wartość `true`, wskazującą, że należy wykonać nowy obraz.

Listing 34.6. Procedura `ProcessSlidebarCommand`

C

```
1: char ProcessSlidebarCommand ( int sln )
2: {
3:   if ( sln >= SL_ID_ARTP0 && sln < SL_ID_ARTP0+NKLARTPARAMS ) {
4:     SetArticulationParameter ( &appdata, sln-SL_ID_ARTP0 );
5:     kl_Articulate ( appdata.linkage );
6:     return true;
7:   }
8:   else
9:     return false;
10: } /*ProcessSlidebarCommand*/
```

Oczywiście, do procedury `DeleteMyWorld` trzeba dodać instrukcje likwidujące program szaderów dokonujący artykulacji oraz bufor z indeksami przekształceń i cały łańcuch kinematyczny.

34.3. Interfejs użytkownika

Listing 34.7 przedstawia zmiany dokonane w procedurach tworzenia menu i obsługi komunikatów wysyłanych do aplikacji przez wihajstry. Wihajstrów tych jest więcej, bo do guzika zatrzymującego program i przełączników wybierających wyświetlaną siatkę doszły suwaki umożliwiające nadawanie wartości parametrom artykulacji.

W linii 4 jest makrodefinicja wprowadzająca identyfikator pierwszego z tych suwaków. Suwaków jest 20 (tyle, ile parametrów artykulacji, zobacz listing 34.1).

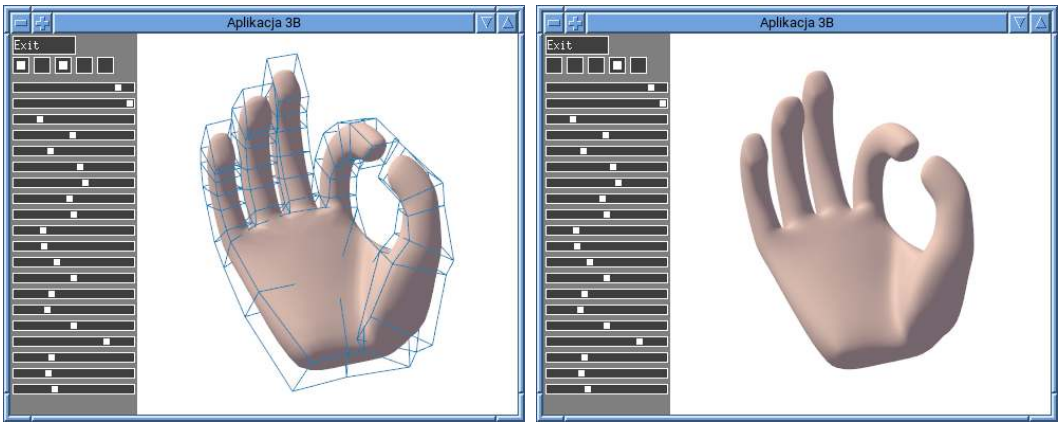
Procedura inicjalizacji menu ma nową nazwę i trzy nowe linie, 41–43, które opisują pętlę tworzącą suwaki. Do poszczególnych suwaków są przywiązywane zmienne typu `float` przechowywane w tablicy `appwdg->artp`. Komunikaty przysyłane przez te suwaki są obsługiwane przez instrukcje w liniach 17–22.

Po otrzymaniu komunikatu od suwaka procedura `Win1CallBack` wywołuje procedurę `ProcessSlidebarCommand`, która realizuje odpowiednią reakcję części graficznej na to zdarzenie. Zarówno po zmianie stanu przełącznika, jak i suwaka, wysyłany jest komunikat powodujący wykonanie nowego obrazu.

Listing 34.7. Procedury tworzenia menu i obsługi jego komunikatów

C

```
#define GLWIN_ID_VIEW 1
#define BTN_ID_EXIT    2
.... /* identyfikatory przełączników bez zmian */
#define SL_ID_ARTP0    8

void Win1Callback ( struct xwidget *wdg, int msg, int key, int x, int y )
{
  switch ( msg ) {
case WDGMSG_BUTTON_PRESS:
    .... /* zatrzymywanie programu po naciśnięciu guzika bez zmian */
    break;

case WDGMSG_SWITCH_CHANGE:
    ProcessSwitchCommand ( wdg->id );
    goto redraw_win2;

case WDGMSG_SLIDEBAR_CHANGE:
    ProcessSlidebarCommand ( wdg->id );
redraw_win2:
    wm2->changed = true;
    PostMenuExposeEvent ( wm2 );
    break;

default:
    break;
  }
} /*Win1CallBack*/

xwinmenu *SetupApp3BMenu ( void )
{
  xwinmenu *wm;
  int      i;

  if ( !(wm = NewWinMenu ( window[1], MENU_WIDTH, WIN0_HEIGHT, 0, 0,
                           NULL, NULL, Win1Callback )) )
    ExitOnError ( "SetupApp3BMenu" );
  NewButton ( wm, BTN_ID_EXIT, 60, 18, 2, 2, str_EXIT );
  for ( i = 0; i < NPALMMESHES; i++ )
    NewSwitch ( wm, SW_ID_MESH0+i, 16, 16, 2+20*i, 22, NULL,
                &appwdg->sw[i] );
  for ( i = 0; i < NARTPARAMS; i++ )
    NewSlidebarf ( wm, SL_ID_ARTP0+i, 116, 10, 2, 46+15*i,
                   &appwdg->artp[i] );
  return wm;
} /*SetupApp3BMenu*/
```



Rysunek 34.2. Okno aplikacji trzeciej B

34.4. Ćwiczenia

1. Zastanów się, jak skrócić procedurę `ConstructPalmLinkage`, zastępując instrukcje w liniach 34–55 instrukcjami wywoływanymi w pętli. Po zastanowieniu weź się do dzieła.
2. *Zastanów się nad możliwością napisania procedury, która skonstruuje kompletny łańcuch kinematyczny (z wieloma obiektami) na podstawie danych opisanych przez swoje parametry, zastępując procedury takie jak `ConstructPalmLinkage` lub `ConstructMyLinkage` z aplikacji 2H i 2I. Niektóre parametry muszą być wskaźnikami procedur (metod wirtualnych) wykonujących obliczenia specyficzne dla obiektów poszczególnych rodzajów.

Motywacją do tych ćwiczeń jest umożliwienie konstruowania różnych łańcuchów kinematycznych na podstawie danych odczytanych przez aplikację z plików.

35

Aplikacja trzecia C

Aplikacja 3C pokaże pazurki (zrobione z płatów Béziera). Ponadto użyjemy w niej wypróbowanego wcześniej modelu oświetlenia Blinna-Phonga i sprawimy, by na obrazach pojawiły się cienie.

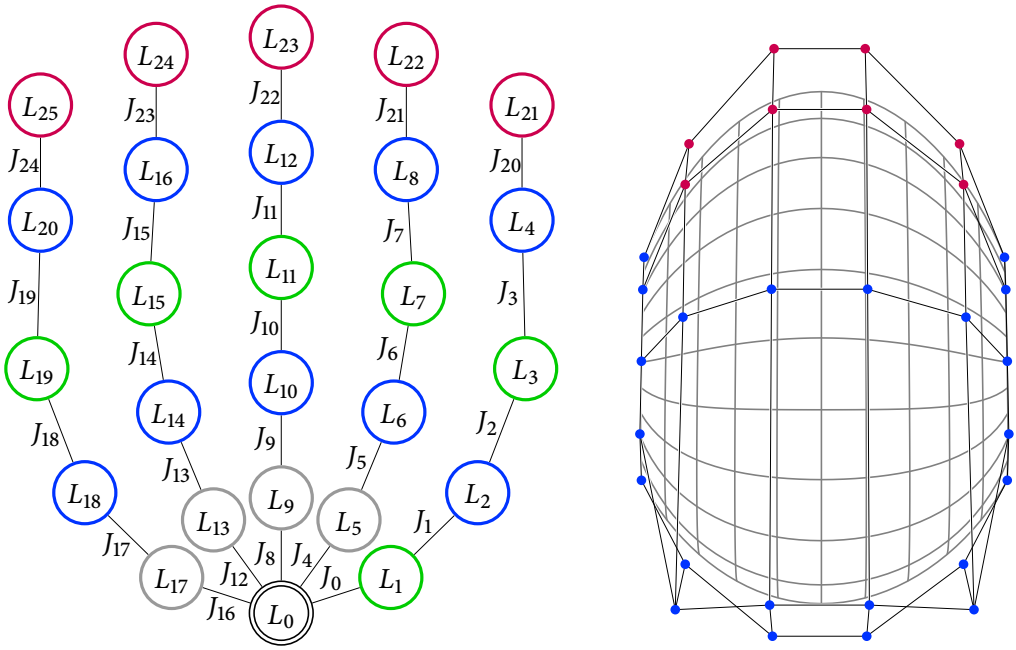
35.1. Łańcuch kinematyczny

Rozbudujemy łańcuch kinematyczny z aplikacji 3B — dodamy do niego 5 nowych członów i par kinematycznych (które wydłużą każdą z „gałęzi" drzewa — grafu opisującego łańcuch). Będą teraz dwa obiekty: siatka dłoni i zestaw pięciu płatów Béziera będących modelami paznokci.

Na rysunku 35.1 jest pokazany graf łańcucha i płat Béziera będący modelem paznokcia. Każda „gałąź" łańcucha jest wydłużona o jedną krawędź i jeden wierzchołek (czyli o jedną parę kinematyczną i jeden człon). Płat ma stopień $(5, 4)$, zatem w każdym wierszu jego siatki jest 6 punktów kontrolnych, a w każdej kolumnie jest ich 5. Wszystkie punkty kontrolne z pierwszych trzech wierszy oraz wszystkie z pierwszej i ostatniej kolumny są związane z przedostatnim członem odpowiedniej gałęzi (np. członem L_4 w przypadku kciuka), a pozostałe punkty będą miały położenia ustalone w układzie współrzędnych ostatniego członu (L_{21} dla kciuka).

Na listingu 35.1 są pokazane makrodefinicje opisujące nowe liczby elementów łańcucha i typ struktury `AppData`, do której zostało dodane pole `nails`, nowe zmienne `shadows` i `final`, potrzebne podczas wykonywania obrazów sceny, oraz pole `brprog`, które jest opisanym dalej opakowaniem programów szaderów używanych do rysowania płatów Béziera, tj. paznokci. Zamiast jednego obiektu są dwa (siatka dłoni i paznokcie), a że każdy paznokieć jest związany z dwoma członami łańcucha, liczba referencji obiektów wzrosła o 10.

Pole `nails` jest strukturą, w której są przechowywane wskaźniki reprezentacji dwóch zestawów pięciu płatów Béziera (oryginalnego i odkształconego), identyfikatory buforów z opisem materiału i z macierzami przekształceń artykulacji oraz kolor siatki kontrolnej.



Rysunek 35.1. Łańcuch kinematyczny aplikacji 3C i model paznokcia

Listing 35.1. Nowe liczby elementów łańcucha i zmieniona struktura `AppData`

C

```
#define MESHDEG          3  /* liczba kroków uśredniania */
#define NPALMMESHES      4  /* liczba zagęszczonych siatek */
#define NKLOBJ           2  /* liczba obiektów w łańcuchu */
#define NKLINKS         26  /* liczba członów */
#define NKLREFS         27  /* liczba referencji */
#define NKLJOINTS       25  /* liczba par kinematycznych */
#define NKLARTPARAMS    21  /* liczba parametrów artykulacji */

typedef struct {
    GPUmesh *mesh[NPALMMESHES+2];
    GLfloat ecolour[3], fcolour[3];
    GLuint  tribuf, mtn;
  } KLMesh;

typedef struct {
    BezierPatchObjf *bpatches[2];
    GLuint          tribuf, mtn;
  } KLBezPatches;

typedef struct {
    AppWidgets              wdg;
    KLMesh                  palm;
```

```
    KLBezPatches          nails;
    kl_linkage            *linkage;
    Camera                camera;
    TransBl               trans;
    LightBl               light;
    MatBl                 mat;
    GLuint                lktrbuf;
    char                  lod, edges, mnv, shadows, final;
    float                 speed;
    float                 model_rot_axis[3];
    double                model_rot_angle;
    MeshRenderPrograms    mrprog;
    BPRenderPrograms      brprog;
    KLArticulationProgram artprog;
  } AppData;
```

Listing 35.2 przedstawia procedurę wprowadzającą reprezentację paznokci do pamięci GPU. Makrodefinicje w liniach 1–5 określają stopień płata, liczbę palców, a także liczby punktów kontrolnych jednego płata (paznokcia) i wszystkich paznokci jednej dłoni.

Listing 35.2. Procedura `EnterFingernailsToGPU`

C

```
#define FINGERNAIL_UDEG 5  /* stopień płata ze względu na parametr u */
#define FINGERNAIL_VDEG 4  /* stopień płata ze względu na parametr v */
#define FINGER_NUM      5  /* liczba palców */
#define FINGERNAIL_NCP  ((FINGERNAIL_UDEG+1)*(FINGERNAIL_VDEG+1))
#define FINGERNAIL_NV   (FINGERNAIL_NCP*FINGER_NUM)

static GLfloat fingernail_cp[][3] = {
  {-0.65655,-0.54132,-0.62502}, {-0.65655,-0.40329,-0.48714},
  .... /* z 30 punktów kontrolnych 27 tu pominąłem */
  { 0.68104, 0.28418,-0.52111}};

void EnterFingernailsToGPU ( BezierPatchObjf *nails[2], GLfloat colour[3] )
{
  GLfloat *nailcp;
  int     i, j, k;
  GLfloat nsc[FINGER_NUM] = {0.09,0.09,0.095,0.085,0.075};
  GLfloat nrot[FINGER_NUM][3] =
    {{-0.05*PI,0.85*PI,-0.07*PI},{-0.03*PI,PI,0.0},{-0.03*PI,PI,0.02*PI},
     {-0.03*PI,PI,0.02*PI},{-0.03*PI,PI,0.02*PI}};
  GLfloat ntrv[FINGER_NUM][3] =
    {{0.48,0.095,-0.088},{0.105,0.75,-0.12},{-0.15,0.8,-0.12},
     {-0.38,0.7,-0.12},{-0.61,0.5,-0.12}};
  GLfloat tr[16];

  if ( !(nailcp = malloc ( FINGERNAIL_NV*3*sizeof(GLfloat) )) )
```

```
26:     ExitOnError ( "EnterFingernailsToGPU 0" );
27:   for ( i = k = 0; i < FINGER_NUM; i++ ) {
28:     M4x4Translatef ( tr, ntrv[i][0], ntrv[i][1], ntrv[i][2] );
29:     M4x4MRotateZf ( tr, nrot[i][2] );
30:     M4x4MRotateYf ( tr, nrot[i][1] );
31:     M4x4MRotateXf ( tr, nrot[i][0] );
32:     M4x4MScalef ( tr, nsc[i], nsc[i], nsc[i] );
33:     for ( j = 0;  j < FINGERNAIL_NCP;  j++, k += 3 )
34:       M4x4MultMP3f ( &nailcp[k], tra, fingernail_cp[j] );
35:   }
36:   nails[0] = EnterBezierPatches ( FINGERNAIL_UDEG, FINGERNAIL_VDEG, 3,
37:                       FINGER_NUM, 1, FINGERNAIL_NV, nailcp,
38:                       (FINGERNAIL_UDEG+1)*(FINGERNAIL_VDEG+1)*3, 0,
39:                       (FINGERNAIL_VDEG+1)*3, 3, colour );
40:   free ( nailcp );
41:   if ( (nails[1] = malloc ( sizeof(BezierPatchObjf) )) ) {
42:     memcpy ( nails[1], nails[0], sizeof(BezierPatchObjf) );
43:     glGenBuffers ( 1, &nails[1]->buf[1] );
44:     glBindBuffer ( GL_SHADER_STORAGE_BUFFER, nails[1]->buf[1] );
45:     glBufferData ( GL_SHADER_STORAGE_BUFFER,
46:             FINGERNAIL_NV*3*sizeof(GLfloat), NULL, GL_DYNAMIC_DRAW );
47:     ExitIfGLError ( "EnterFingernailsToGPU" );
48:   }
49:   else
50:     ExitOnError ( "EnterFingernailsToGPU" );
51: } /*EnterFingernailsToGPU*/
```

Wszystkie paznokcie są obrazami jednego płata Béziera w odpowiednio dobranych (indywidualnie dla każdego palca) przekształceniach afinicznych. Punkty kontrolne tego płata są podane w tablicy `fingernail_cp`. W linii 25 procedura `EnterFingernailsToGPU` rezerwuje bufor, w którym zapisze punkty kontrolne wszystkich paznokci, po czym w pętli w liniach 27–35 konstruuje odpowiednie przekształcenie dla każdego palca i (w wewnętrznej pętli w liniach 33–34) poddaje mu punkty z tablicy `fingernail_cp`.

Przekształcenia są konstruowane przez instrukcje w liniach 28–32; każde z nich jest opisane przez macierz $A_i = T_i R_{zi} R_{yi} R_{xi} S_i$ — iloczyn macierzy przesunięcia T_i, trzech macierzy obrotów (wokół osi z, y, x) i macierzy skalowania S_i. Parametry tych przekształceń są zapisane w tablicach `ntrv`, `nrot` i `nsc`; może warto zwrócić uwagę, że wielkości poszczególnych paznokci są różne, za co odpowiadają współczynniki skalowania (równomiernego dla wszystkich osi) w tablicy `nsc`.

Po zakończeniu zewnętrznej pętli tablica `nailcp` zawiera współrzędne punktów, które procedura `EnterBezierPatches` (z listingu 15.8) razem z pozostałymi elementami opisu płatów przesyła do pamięci GPU. W linii 40 pamięć zajmowana przez niepotrzebną już tablicę jest zwalniana, po czym w liniach 41–46 konstruowana jest struktura danych przeznaczona do reprezentowania paznokci po artykulacji.

Listing 35.3 przedstawia zmiany w procedurze `ConstructPalmLinkage`. W liniach 6–7 są dodatkowe elementy opisujące rodzaj nowych par kinematycznych — wszystkie one reali-

Listing 35.3. Zmiany w procedurze `ConstructPalmLinkage`

C

```
kl_linkage *ConstructPalmLinkage ( mypalmscene *scene )
{
  static int jtype[NKLJOINTS] =
    { KL_ART_ROT_Y, KL_ART_ROT_Z, KL_ART_ROT_X, KL_ART_ROT_X,
      ....
      KL_ART_TRANS_Y, KL_ART_TRANS_Y, KL_ART_TRANS_Y, KL_ART_TRANS_Y,
      KL_ART_TRANS_Y };
  static int jpnum[NKLJOINTS] =
    {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,20,20,20,20};
  static float pp0[3] = { ... }, vp0[3] = { ... }, vp1[3] = { ... };
  kl_linkage *lkg;
  int        lnk[NKLINKS], jnt[NKLJOINTS];
  GLfloat    tra[16];
  int        i, j, k, l;

  if ( (lkg = kl_NewLinkage ( NKLOBJ, NKLINKS, NKLREFS, NKLJOINTS,
                              NKLARTPARAMS, (void*)scene )) ) {
    memcpy ( scene->palmartp, palmartp0, NKLARTPARAMS*sizeof(float) );
    for ( i = 0; i < NKLINKS; i++ )
      lnk[i] = kl_NewLink ( lkg );
    kl_NewObject ( lkg, 0, 3, PALM_NV, NULL, (void*)&ad->palm,
                   KLInitPalmMesh, KLTransformVertices,
                   KLPostprocessMesh, KLRedrawMesh, KLDeletePalmMesh );
    kl_NewObject ( lkg, 0, 3, FINGERNAIL_NV, NULL, (void*)&ad->nails,
           KLInitFingernails, KLTransformVertices, KLPostprocessFingernails,
           KLRedrawBezPatches, KLDeleteFingernails );
    for ( k = j = 0;  k < 5;  k++ ) {
      for ( i = 0, l = -1;  i < 4;  i++, l = j++ )
        jnt[j] = kl_NewJoint ( lkg, lnk[l+1], lnk[j+1],
                               jtype[j], jpnum[j] );
    }
    for ( k = 0;  k < 5;  k++, j++ )
      jnt[j] = kl_NewJoint ( lkg, lnk[4*(k+1)], lnk[j+1],
                             jtype[j], jpnum[j] );
        /* kciuk */
    M4x4RotateP2Vf ( tra, pp0, vp0, vp1 );
    ....
    kl_SetJointFtr ( lkg, jnt[3], tra, true );
    M4x4RotateZf ( tra, -0.1*PI );
    kl_SetJointFtr ( lkg, jnt[20], tra, true );
    ....
    glGenBuffers ( 1, &ad->lktrbuf );
    glBindBuffer ( GL_SHADER_STORAGE_BUFFER, ad->lktrbuf );
    glBufferData ( GL_SHADER_STORAGE_BUFFER, lkg->norefs*16*sizeof(GLfloat),
                   NULL, GL_DYNAMIC_DRAW );
```

```
46:   }
47:   return scene->lkg = lkg;
48: } /*ConstructPalmLinkage*/
```

zują przesunięcia wzdłuż osi y. W linii 9 jest wydłużona tablica z numerami parametrów artykulacji dla poszczególnych par — wszystkie nowe pary mają *jeden wspólny* parametr artykulacji, który określa wielkość przesunięcia. Początkowa wartość tego parametru to 0.

Metody obiektu — siatki dłoni — zostały tylko rozszerzone o instrukcje wprowadzające opis materiału używanego do obliczeń oświetlenia. Jego numer jest zapamiętywany w polu `mtn` struktury typu `KLMesh`. W liniach 24–26 jest dodane wywołanie procedury `kl_NewObject`, której celem jest utworzenie i dołączenie do łańcucha obiektu paznokci. Metody tego obiektu są przedstawione na listingu 35.4. Pętla w liniach 27–28, wprowadzająca pary kinematyczne obecne w aplikacji 3B, też nie uległa zmianie, natomiast nowe pary, $J_{20}, \ldots, J_{24}$, wprowadzane są w pętli dodanej w liniach 32–34.

Prawie każda z nowych par kinematycznych realizuje przesunięcie wzdłuż osi y przedostatniego członu odpowiedniego palca, przy czym w położeniu początkowym oś ta ma kierunek osi y układu związanego z członem L_0. W związku z tym macierze stałe $F_{21}, \ldots, F_{24}$ tych par są macierzą jednostkową — taka macierz jest przyjmowana domyślnie dla każdej pary przez procedurę `kl_NewJoint` i nie trzeba jej zmieniać. W wyjątkowy sposób trzeba potraktować kciuk, którego paznokieć w położeniu wyjściowym jest obrócony inaczej niż pozostałe i przesunięcia punktów kontrolnych jego siatki też powinny mieć inny kierunek. Odpowiednia macierz, F_{20}, jest konstruowana przez instrukcję w linii 39.

Procedura `KLInitFingernails`, która jest konstruktorem obiektu paznokci, w linii 8 oblicza numer obiektu[1]. W linii 10 następuje rezerwacja tablicy roboczej, w której dla każdego punktu kontrolnego zostanie (w liniach 13–24) obliczony numer przekształcenia artykulacji dla tego wierzchołka i z której numery te (w liniach 25–26) zostaną przesłane do utworzonego w linii 11 bufora w pamięci GPU. W linii 28 jest wywołana procedura z listingu 35.2. W liniach 29–30 zostaje utworzony opis materiału, którego kolor będzie ustalany bezpośrednio przed rysowaniem (zmiany parametrów artykulacji będą powodować zmiany tego koloru). W linii 31 w opisie paznokci zostaje zapamiętany kolor siatek kontrolnych płatów Béziera na obrazie.

Procedura `KLDeleteFingernails`, czyli destruktor wywoływany podczas likwidacji łańcucha, zwalnia pamięć CPU i GPU zajmowaną przez reprezentację paznokci. Drugi zestaw płatów ma z pierwszym wspólny bufor z blokiem `BezPatch`, więc jego likwidacja wymaga tylko zwolnienia bufora z blokiem `CPoints` i struktury w pamięci CPU (linie 43–44).

Metodą `transform` obiektu paznokci jest ta sama procedura `KLTransformVertices` co dla powierzchni siatkowej (listing 34.3). Otrzymaną jako parametr macierz przesyła ona do bufora, którego identyfikator jest pamiętany w polu `lktrbuf` struktury typu `AppData` opakowującej całą reprezentację sceny do narysowania.

[1]Drugi obiekt wprowadzony do łańcucha ma oczywiście numer 1, ale lepiej jest obliczyć go w taki sposób, aby łatwiej było rozbudować aplikację dalej.

Listing 35.4. Metody obiektu paznokci

C

```
static char KLInitFingernails ( kl_linkage *lkg, kl_object *obj )
{
  static const GLfloat cnetcolour[3] = { 0.0, 1.0, 0.0 };
  KLBezPatches *pd;
  int          i, j, l, m, on, rn;
  GLuint       *cpi;

  on = obj - lkg->obj;
  pd = (KLBezPatches*)obj->usrdata;
  if ( (cpi = malloc (FINGERNAIL_NV*sizeof(GLuint))) ) {
    glGenBuffers ( 1, &pd->tribuf );
    glBindBuffer ( SSB, pd->tribuf );
    for ( l = 0; l < FINGER_NUM; l++ ) {  /* kolejno dla każdego palca */
      rn = kl_NewObjRef ( lkg, 4*(l+1), on, 0, NULL );
      for ( i = 0;  i <= FINGERNAIL_UDEG;  i++ ) {
        m = i == 0 || i == FINGERNAIL_UDEG ? FINGERNAIL_VDEG : 2;
        for ( j = 0;  j <= m;  j++ )
          cpi[l*FINGERNAIL_NCP + (FINGERNAIL_VDEG+1)*i + j] = rn;
      }
      rn = kl_NewObjRef ( lkg, 21+l, on, 0, NULL );
      for ( i = 1;  i < FINGERNAIL_UDEG;  i++ )
        for ( j = 3;  j <= FINGERNAIL_VDEG;  j++ )
          cpi[l*FINGERNAIL_NCP + (FINGERNAIL_VDEG+1)*i + j] = rn;
    }
    glBufferData ( SSB, FINGERNAIL_NV*sizeof(GLuint),
                   cpi, GL_STATIC_DRAW );
    free ( cpi );
    EnterFingernailsToGPU ( pd->bpatches, cnetcolour );
    pd->mtn = SetupMaterial ( &ad->mat, -1, cnetcolour, cnetcolour,
                              1.0, 1.0, 1.0 );
  }
  else
    ExitOnError ( "KLInitFingernails" );
  return true;
} /*KLInitFingernails*/

static void KLDeleteFingernails ( kl_linkage *lkg, kl_object *obj )
{
  KLBezPatches *pd;

  pd = (KLBezPatches*)obj->usrdata;
  DeleteBezierPatches ( pd->bpatches[0] );
  glDeleteBuffers ( 1, &pd->bpatches[1]->buf[1] );
  free ( pd->bpatches[1] );
  glDeleteBuffers ( 1, &pd->tribuf );
```

```
} /*KLDeleteFingernails*/

static void V4Interpolatef ( GLfloat v[4],
                         const GLfloat v0[4], const GLfloat v1[4], float t )
{
  int   i;
  float s;

  for ( i = 0, s = 1.0-t;  i < 3;  i++ )
    v[i] = s*v0[i] + t*v1[i];
} /*V4Interpolatef*/

static void KLPostprocessFingernails ( kl_linkage *lkg, kl_object *obj )
{
  const GLfloat diffr0[4] = { 0.91, 0.65, 0.5, 1.0 };
  const GLfloat specr0[4] = { 0.15, 0.1, 0.12, 1.0 };
  const GLfloat diffr1[4] = { 0.7, 0.1, 0.25, 1.0 };
  const GLfloat specr1[4] = { 0.4, 0.4, 0.4, 1.0 };
  const GLfloat shn = 60.0, wa = 5.0, we = 5.0;
  AppData               *ad;
  KLArticulationProgram *prog;
  KLBezPatches          *pd;
  BezierPatchObjf       **nails;
  GLfloat               diffr[4], specr[4];
  float                 t;

  ad = (AppData*)lkg->usrdata;
  prog = &ad->artprog;
  pd = (KLBezPatches*)obj->usrdata;
  nails = pd->bpatches;
  glUseProgram ( ad->artprog.progid );
  glBindBufferBase ( SSB, prog->ctrbp, ad->lktrbuf );
  glBindBufferBase ( SSB, prog->ctribp, pd->tribuf );
  glBindBufferBase ( SSB, prog->cpibp, nails[0]->buf[1] );
  glBindBufferBase ( SSB, prog->cpobp, nails[1]->buf[1] );
  glUniform1i ( prog->dim_loc, obj->nvc );
  glUniform1i ( prog->trnum_loc, -1 );
  glUniform1i ( prog->ncp_loc, (GLint)obj->nvert );
  glDispatchCompute ( obj->nvert, 1, 1 );
  t = ad->artp[NKLARTPARAMS-1];
  t = t > 0.2 ? 1 : 5.0*t;
  V4Interpolatef ( diffr, diffr0, diffr1, t );
  V4Interpolatef ( specr, specr0, specr1, t );
  SetupMaterial ( &ad->mat, mt->mtn, diffr, specr, shn, wa, we );
  glMemoryBarrier ( GL_UNIFORM_BARRIER_BIT );
  ExitIfGLError ( "KLPostprocessFingernails" );
} /*KLPostprocessFingernails*/
```

```
93:
94: typedef struct {
95:     GLuint progid[3];
96:     GLint  LightingModelLoc;
97:   } BPRenderPrograms;
98:
99: static void KLRedrawBezPatches ( kl_linkage *lkg, kl_object *obj )
100: {
101:   static const int TessLevel[4] = {4,8,16,32};
102:   AppData      *ad;
103:   KLBezPatches *bezp;
104:
105:   ad = (AppData*)lkg->usrdata;
106:   bezp = (KLBezPatches*)obj->usrdata;
107:   glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
108:   SetBezierPatchTessLevel ( bezp->bpatches[1], TessLevel[ad->lod-1] );
109:   SetBezierPatchNVS ( bezp->bpatches[1], (GLint)ad->mnv );
110:   if ( ad->edges )
111:     glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
112:   else
113:     glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
114:   if ( ad->final ) {
115:     glUseProgram ( ad->brprog.progid[1] );
116:     ChooseMaterial ( &bezp->mat, bezp->mtn );
117:   }
118:   else
119:     glUseProgram ( ad->brprog.progid[2] );
120:   DrawBezierPatches ( bezp->bpatches[1] );
121:   if ( ad->meshsw[0] ) {
122:     glUseProgram ( ad->brprog.progid[0] );
123:     DrawBezierNets ( bezp->bpatches[1] );
124:   }
125:   ExitIfGLError ( "KLRedrawBezPatches" );
126: } /*KLRedrawBezPatches*/
```

Procedura `KLPostprocessFingernails` dokonuje artykulacji paznokci przy użyciu szadera obliczeniowego z listingu 23.1, który dokonuje też artykulacji siatki dłoni. Metody artykulacji siatki i płatów Béziera są oczywiście różne, ale w każdej z nich do odpowiednich punktów dowiązania (w celu `GL_SHADER_STORAGE_BUFFER`) są przywiązywane bufory z tablicami macierzy przekształceń, numerów przekształceń poszczególnych punktów, wierzchołków, które trzeba przekształcić, i miejsc, w których przekształcone punkty mają być zapisane, a potem uruchamiany jest program szaderów.

Artykulacja paznokci oprócz przekształcania punktów kontrolnych określa kolor. Obliczenie wykonywane w liniach 86–88 polega na interpolacji parametrów materiału za pomocą procedury `V4Interpolatef`[2]. Opis materiału jest przesyłany do pamięci GPU, gdzie będzie

[2]będącej odpowiednikiem procedury `mix` w GLSL-u

gotowy do użycia podczas rysowania. Zauważmy, że te obliczenia nie kolidują z działaniami szadera uruchomionego w linii 84, zatem procedura `glMemoryBarrier`, która czeka na zakończenie tych działań, może być wywołana na końcu postprocesingu.

Procedura `KLRedrawBezPatches` jest metodą rysowania płatów Béziera, której wywołanie może mieć na celu znalezienie obszaru cienia lub wykonanie końcowego obrazu. Pole `brprog` jest strukturą typu `BPRenderPrograms`, tj. opakowaniem trzech programów szaderów, których identyfikatory są pamiętane w tablicy `progid`. Pierwszy z nich (`progid[0]`) służy w obu przypadkach do rysowania siatki kontrolnej. Drugi program (`progid[1]`) służy do znajdowania obszaru cienia, a trzeci (`progid[2]`) rysuje końcowy obraz. Kolor odcinków siatki kontrolnej na obrazie jest brany z pola `Colour` w opisie płatów, a pole w linii 96 służy do wyboru modelu oświetlenia (Lamberta lub Blinna-Phonga) używanego podczas rysowania płatów Béziera[3]. Procedury kompilujące programy rysowania płatów Béziera i siatek są opisane dalej.

Metoda rysowania obiektu paznokci, `KLRedrawBezPatches`, zależnie od wartości zmiennej `ad->lod` określa stopień rozdrobnienia płatów Béziera dostosowany do stopnia zagęszczenia wyświetlanej siatki, wywołując procedurę `SetBezierPatchOptions`. Zmienna ta określa poziom szczegółowości wyświetlanej siatki, tj. liczbę iteracji rozdrabniania. Liczba ścian podczas każdego rozdrabniania siatki rośnie czterokrotnie, zatem liczba trójkątów płatów Béziera, na każdym następnym poziomie szczegółowości, powinna być dobrana zgodnie z tą samą regułą. Zatem, liczby podane w tablicy `TessLevel` powodują, że zależnie od poziomu każdy płat Béziera (czyli każdy paznokieć) będzie podzielony na to $2 \cdot 4^2 = 32$, $2 \cdot 8^2 = 128$, $2 \cdot 16^2 = 512$ albo $2 \cdot 32^2 = 2048$ trójkąty.

Podczas znajdowania obszaru cienia procedura wykonuje instrukcję w linii 119, a podczas rysowania obrazu końcowego instrukcje w liniach 115–116; w obu przypadkach są wybierane różne programy szaderów, a dla końcowego obrazu jest także przyczepiany bufor z opisem materiału. Jeśli zmienna `ad->meshsw[0]` ma wartość niezerową, to następuje jeszcze rysowanie siatki kontrolnej.

35.2. Szadery rysujące i ich przygotowanie

Aplikacja wykonuje obrazy przy użyciu sześciu programów szaderów: trzech dla obiektu siatki i trzech dla paznokci. Wszystkie szadery używane do wykonywania obrazu przez aplikację 3C były wypróbowane wcześniej, choć do niektórych z nich trzeba było wprowadzić drobne modyfikacje. Identyfikatory tych programów razem z położeniami zmiennych jednolitych są pamiętane w tablicach `progid` w strukturach typu `MeshRenderPrograms` dla siatek i `BPRenderPrograms` dla płatów Béziera.

Listing 35.5 przedstawia zmiany wprowadzone do szadera wierzchołków z listingu 33.4, w celu otrzymania cieni na końcowym obrazie siatki. W liniach 29–31 następuje obliczenie współrzędnych wierzchołka w układach związanych z wszystkimi włączonymi źródłami

[3] Takie samo pole zostało dodane do struktury `MeshRenderPrograms`, tj. opakowania trzech programów do rysowania siatek.

światła. Do szadera geometrii trzeba dodać przekazywanie tych współrzędnych z wejścia na wyjście; uznałem, że listing tego szadera jest zbędny.

Listing 35.5. Szader wierzchołków programu rysowania ścian siatki z cieniami

GLSL

```
.... /* początek szadera bez zmian */

out GVertex {
    vec4 Normal;
    vec4 ShadowPos[MAX_NLIGHTS];
  } Out;

.... /* bloki magazynowe z reprezentacją siatki jak na listingu 31.4 */

uniform TransBlock { .... } trb;  /* tak, jak na listingu 12.3 */
struct LSPar { .... };            /* tak, jak na listingu 22.3 */
uniform LSBlock { .... } light;   /* tak, jak na listingu 22.3 */


void main ( void )
{
  vec4 pos, wpos;
  uint i, j, l, mask;

  i = mhe(mfhei((mfac(gl_InstanceID) & FHEMASK) + gl_VertexID)).V0;
  j = nsattr*i + pofs;
  switch ( pdim ) {
    .... /* odczytywanie położenia wierzchołka bez zmian, */
    .... /* ale przypisujemy je zmiennej pos */
  }
  gl_Position = pos;
  .... /* obliczanie wektora normalnego bez zmian */
  wpos = trb.mm * pos;
  for ( l = 0, mask = 0x00000001;  l < light.nls;  l++, mask <<= 1 )
    if ( (light.mask & mask) != 0 )
      Out.ShadowPos[l] = light.ls[l].shadow_vpm * wpos;
} /*main*/
```

Nie zamieściłem tu również listingu szadera fragmentów wykonującego końcowy obraz płatów Béziera i powierzchni siatkowej; szader ten powstał z przedstawionego na listingu 22.5 szadera aplikacji 2G przez usunięcie instrukcji związanych z nakładaniem tekstury na powierzchnie[4]. Bez zmian pozostawiłem modele oświetlenia (Lamberta i Blinna-Phonga) i opisy świateł oraz własności materiału używane do obliczania kolorów fragmentów.

[4]Powierzchnia reprezentowana przez siatkę nieregularną zazwyczaj nie jest płatem (tj. powierzchnią o parametryzacji, której dziedzina jest obszarem płaskim), przez co wygenerowanie sensownych współrzędnych tekstury dwuwymiarowej dla jej wierzchołków, choć wykonalne, jest zadaniem dosyć trudnym. Pozostawiam je jako przedmiot dalszych studiów.

Program do znajdowania obszaru cienia rzucanego przez powierzchnię siatkową jest znacznie uproszczony, bo nie ma w nim potrzeby przetwarzania wektora normalnego powierzchni. Program ten składa się z dwóch szaderów, wierzchołków i fragmentów. Szader wierzchołków powstał przez uproszczenie szadera z listingu 33.4 — instrukcje odczytujące z tablic wektor normalny zostały usunięte. Ponieważ w tym programie szader geometrii jest nieobecny, szader wierzchołków musi wykonać dodatkowe zadanie — dokonać przejścia do układu kostki standardowej i przypisać zmiennej `gl_Position` współrzędne wierzchołka w tym układzie. Macierze przejścia od układu modelu do układu świata i dalej do układu kostki standardowej są brane z bloku zmiennych jednolitych `TransBlock`, przy czym zawsze, gdy ten program jest wykonywany, układ kostki standardowej jest związany ze źródłem światła.

Listing 35.6 przedstawia procedury, których zadaniem jest przygotowanie do pracy programów rysujących. W celu poprawienia czytelności kodu aplikacji programy rysujące siatki i płaty Béziera są przygotowywane przez osobne procedury, `LinkMeshRenderingProgams` i `LinkBPRenderingPrograms`, ale programy te mają wiele szaderów wspólnych. W związku z tym pomocnicza procedura `LoadRenderingShaders` dokonuje kompilacji wszystkich szaderów, z których składają się te programy; trzeba ją wywołać przed wspomnianymi procedurami, a po złączeniu programów można sprzątnąć szadery, aby nie zajmowały miejsca.

Pomocnicza procedura `LinkMyShaderProgram` umożliwia skrócenie kodu aplikacji; pierwszym jej parametrem jest tablica liczb całkowitych, z których pierwsza jest liczbą szaderów do połączenia w program, a kolejne liczby są indeksami do tablicy `shader` zawierającej identyfikatory skompilowanych szaderów. W liniach 28–29 identyfikatory szaderów, które mają być połączone w program, są przepisywane do pomocniczej tablicy, która jest następnie przekazywana procedurze `LinkShaderProgram` z listingu 4.7. Ostatni parametr jest napisem, który zostanie wyświetlony w razie wystąpienia błędu łączenia programu, w celu ułatwienia znalezienia jego przyczyny.

Listing 35.6. Procedury kompilacji i łączenia programów rysujących

C

```
1:  void LoadRenderingShaders ( GLuint *shid )
2:  {
3:    static const GLchar *filename[] =
4:      { "app3C0.vert.glsl", "app3C1.vert.glsl", "app3C2.vert.glsl",
5:        "app3C3.vert.glsl", "app3C4.vert.glsl", "app2.tesc.glsl",
6:        "app3C0.tese.glsl", "app3C1.tese.glsl", "app3C0.geom.glsl",
7:        "app3C1.geom.glsl", "app3C2.geom.glsl", "app3C0.frag.glsl",
8:        "app3C1.frag.glsl" };
9:    static const GLuint shtype[] =
10:     { GL_VERTEX_SHADER, GL_VERTEX_SHADER, GL_VERTEX_SHADER,
11:       GL_VERTEX_SHADER, GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER,
12:       GL_TESS_EVALUATION_SHADER, GL_TESS_EVALUATION_SHADER,
13:       GL_GEOMETRY_SHADER, GL_GEOMETRY_SHADER, GL_GEOMETRY_SHADER,
14:       GL_FRAGMENT_SHADER, GL_FRAGMENT_SHADER };
15:   int i;
```

```
16:
17:   for ( i = 0; i < 13; i++ )
18:     shid[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
19: } /*LoadRenderingShaders*/
20:
21: static const GLchar *UVNames[] = { "LightingModel" };
22:
23: static GLuint LinkMyShaderProgram ( const int *shn, const GLuint *shaders,
24:                                      const char *name )
25: {
26:   GLuint sh[5], i;
27:
28:   for ( i = 0; i < shn[0]; i++ )
29:     sh[i] = shaders[shn[i+1]];
30:   return LinkShaderProgram ( shn[0], sh, name );
31: } /*LinkMyShaderProgram*/
32:
33: void LinkMeshRenderingPrograms ( MeshRenderPrograms *prog, GLuint *shid )
34: {
35:   static const int p0sh[] = {3,0,8,11};
36:   static const int p1sh[] = {3,1,9,12};
37:   static const int p2sh[] = {2,2,11};
38:   int i;
39:
40:   prog->progid[0] = LinkMyShaderProgram ( p0sh, shid, "0" );
41:   prog->progid[1] = LinkMyShaderProgram ( p1sh, shid, "1" );
42:   prog->progid[2] = LinkMyShaderProgram ( p2sh, shid, "2" );
43:   GetAccessToMeshSurfBlock ( prog->progid[1] );
44:   prog->LightingModelLoc =
45:           glGetUniformLocation ( prog->progid[1], UVNames[0] );
46:   GetAccessToTransBlockUniform ( prog->progid[0] );
47:   GetAccessToLightMatUniformBlocks ( prog->progid[1] );
48:   for ( i = 1; i < 3; i++ )
49:     AttachUniformTransBlockToBP ( prog->progid[i] );
50:   ExitIfGLError ( "LinkMeshRenderingShaders" );
51: } /*LinkMeshRenderingPrograms*/
52:
53: void LinkBPRenderingPrograms ( BPRenderPrograms *prog, GLuint *shid )
54: {
55:   static const int p3sh[] = {2,4,11};
56:   static const int p4sh[] = {5,3,5,6,10,12};
57:   static const int p5sh[] = {4,3,5,7,11};
58:   static const GLchar *UVnames[] = { "colour" };
59:   int i;
60:
61:   prog->progid[0] = LinkMyShaderProgram ( p3sh, shid, "3" );
62:   prog->progid[1] = LinkMyShaderProgram ( p4sh, shid, "4" );
```

```
63:   prog->progid[2] = LinkMyShaderProgram ( p5sh, shid, "5" );
64:   GetAccessToBezPatchStorageBlocks ( prog->progid[1], false, false );
65:   prog->LightingModelLoc =
66:           glGetUniformLocation ( prog->progid[1], UVNames[0] );
67:   GetAccessToTransBlockUniform ( prog->progid[0] );
68:   for ( i = 1; i < 3; i++ )
69:     AttachUniformTransBlockToBP ( prog->progid[i] );
70:   GetAccessToLightMatUniformBlocks ( prog->progid[1] );
71:   prog->ucolour_loc = glGetUniformLocation ( prog->progid[0], UVnames[0] );
72:   ExitIfGLError ( "LoadBPRenderingPrograms" );
73: } /*LinkBPRenderingPrograms*/
```

Obie procedury przygotowania programów rysujących dla aplikacji wykonują rutynowe działania, tzn. łączą programy ze skompilowanych szaderów i odczytują z nich położenia zmiennych jednolitych i informacje o przesunięciach pól w blokach zmiennych jednolitych `TransBlock`, `LSBlock` i `MatBlock`. We wszystkich programach rysujących bloki te mają identyczą budowę.

Listing 35.7 przedstawia procedury likwidujące programy używane do rysowania siatek i płatów Béziera. Wywołanie tych procedur trzeba dopisać do procedury sprzątającej po zatrzymaniu aplikacji przez użytkownika.

Listing 35.7. Procedury likwidacji programów rysujących

C

```
1: void DeleteMeshRenderingPrograms ( MeshRenderPrograms *prog )
2: {
3:   int i;
4:
5:   glUseProgram ( 0 );
6:   for ( i = 0; i < 3; i++ )
7:     glDeleteProgram ( prog->progid[i] );
8:   ExitIfGLError ( "DeleteMeshRenderingPrograms" );
9: } /*DeleteMeshRenderingPrograms*/
10:
11: void DeleteBPRenderingPrograms ( BPRenderPrograms *prog )
12: {
13:   int i;
14:
15:   glUseProgram ( 0 );
16:   for ( i = 0; i < 3; i++ )
17:     glDeleteProgram ( prog->progid[i] );
18:   ExitIfGLError ( "DeleteBPRenderingPrograms" );
19: } /*DeleteBPRenderingPrograms*/
```

Listing 35.8 przedstawia nową procedurę rysowania ścian siatki. Procedura ta przywiązuje bufory z reprezentacją siatki do odpowiednich punktów dowiązania w celu `GL_SHADER_STORAGE_BUFFER`, a następnie zależnie od wartości parametru `final` procedura

wybiera program szaderów dla obrazu końcowego (`rprog_id[1]`) albo dla wyznaczania obszaru cienia (`rprog_id[2]`). Przed wykonywaniem końcowego obrazu procedura wybiera materiał, przywiązuje pusty obiekt tablicy wierzchołków, po czym, wywołując procedurę `glDrawArraysInstanced`, uruchamia potok przetwarzania grafiki.

Listing 35.8. Procedura rysowania ścian siatki

C

```
void DrawMeshFacets ( MeshRenderPrograms *prog, GPUmesh *mesh,
                      MatBl *mat, GLint mtn, char nvs, char final )
{
  int i;

  for ( i = 0; i < 4; i++ )
    glBindBufferBase ( SSB, i, mesh->mbuf[i] );
  if ( final ) {
    glUseProgram ( prog->progid[1] );
    SetMeshNVS ( mesh, (GLint)nvs );
    ChooseMaterial ( mat, mtn );
  }
  else
    glUseProgram ( prog->progid[2] );
  glBindVertexArray ( empty_vao );
  glDrawArraysInstanced ( GL_TRIANGLE_FAN, 0, 4, mesh->nfac );
  glBindVertexArray ( 0 );
  ExitIfGLError ( "DrawMeshFacets" );
} /*DrawMeshFacets*/
```

Listing 35.9 przedstawia procedurę `RedrawMyWorld` i procedury wywoływane przez nią w celu znalezienia obszarów cienia i wykonania obrazu końcowego. Nadanie (na polecenie użytkownika) wartości `false` polu `shadows` powoduje pominięcie znajdowania cieni, jeśli jednak mają one być na obrazie, to procedura `DrawSceneToShadows` nadaje wartość `false` polu `final`, powodując wybranie (przez procedury rysujące siatkę i płaty Béziera) odpowiednich programów szaderów. Działanie tej procedury i procedur OpenGL-a przez nią wywoływanych jest opisane szczegółowo w rozdziale 22.

Listing 35.9. Procedury rysowania sceny

C

```
void DrawMyScene ( AppData *ad )
{
  kl_Redraw ( ad->linkage );
} /*DrawMyScene*/

void DrawSceneToShadows ( AppData *ad )
{
  int    l;
  GLuint mask;
```

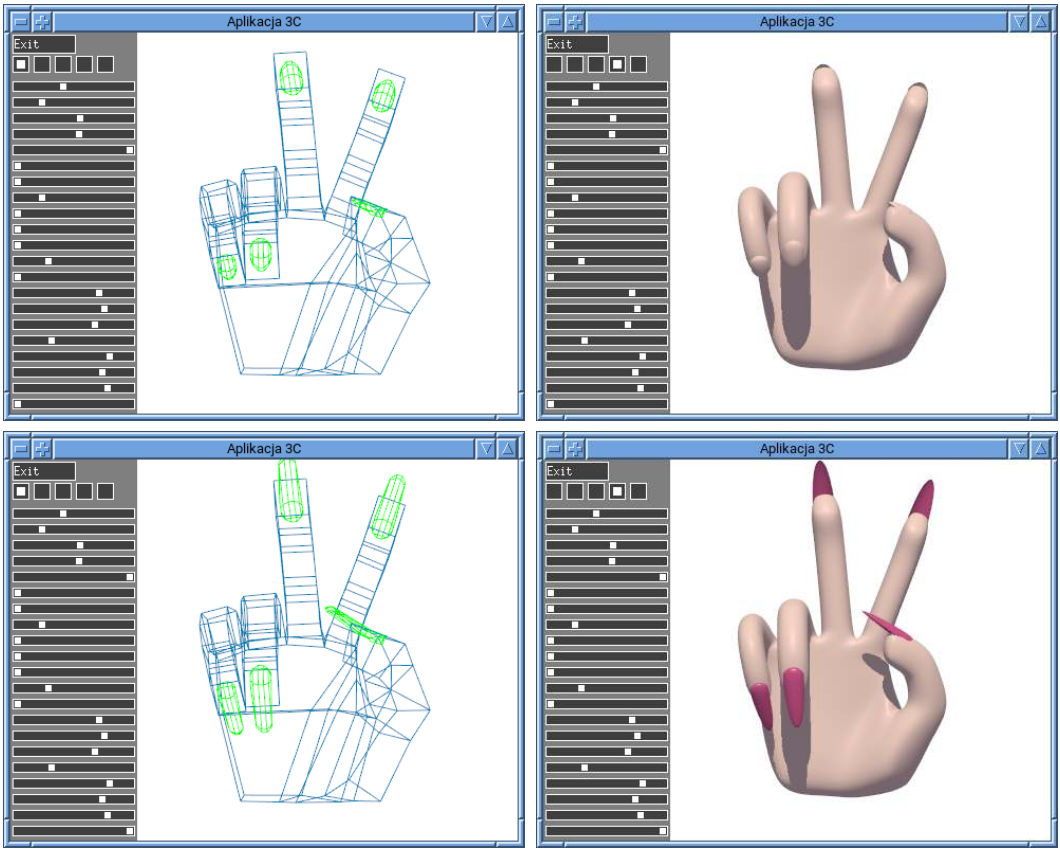
```
10:
11:   appdata.final = false;
12:   glViewport ( 0, 0, SHADOW_MAP_SIZE, SHADOW_MAP_SIZE );
13:   glEnable ( GL_POLYGON_OFFSET_FILL );
14:   glPolygonOffset ( 2.0f, 4.0f );
15:   for ( l = 0, mask = 0x00000001;  l < ad->light.nls;  l++, mask <<= 1 )
16:     if ( ad->light.shmask & mask ) {
17:       BindShadowTxtFBO ( &ad->trans, &ad->light, l );
18:       glClear ( GL_DEPTH_BUFFER_BIT );
19:       DrawMyScene ( ad );
20:     }
21:   glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
22:   glDisable ( GL_POLYGON_OFFSET_FILL );
23:   for ( l = 0, mask = 0x00000001;  l < ad->light.nls;  l++, mask <<= 1 )
24:     if ( ad->light.shmask & mask ) {
25:       glActiveTexture ( GL_TEXTURE0+l );
26:       glBindTexture ( GL_TEXTURE_2D, ad->light.ls[l].shadow_txt );
27:     }
28: } /*DrawSceneToShadows*/
29:
30: void DrawSceneToWindow ( AppData *ad )
31: {
32:   appdata.final = true;
33:   glViewport ( 0, 0, ad->camera.win_width, ad->camera.win_height );
34:   glClearColor ( 1.0, 1.0, 1.0, 1.0 );
35:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
36:   LoadVPMatrix ( &ad->trans );
37:   DrawMyScene ( &appdata );
38: } /*DrawSceneToWindow*/
39:
40: void RedrawMyWorld ( void )
41: {
42:   glEnable ( GL_DEPTH_TEST );
43:   if ( appdata.shadows )
44:     DrawSceneToShadows ( &appdata );
45:   DrawSceneToWindow ( &appdata );
46:   glFlush ();
47: } /*RedrawMyWorld*/
48:
49: void ToggleLightModel ( AppData *ad )
50: {
51:   ad->lighting_model = !ad->lighting_model;
52:   glUseProgram ( ad->mrprog.progid[1] );
53:   glUniform1i ( ad->mrprog.LightingModelLoc, ad->lighting_model );
54:   glUseProgram ( ad->brprog.progid[1] );
55:   glUniform1i ( ad->brprog.LightingModelLoc, ad->lighting_model );
56: } /*ToggleLightModel*/
```

Procedura `ToggleLightModel` na polecenie użytkownika przełącza model oświetlenia. Identyfikator bieżącego modelu (liczba `0` albo `1`) jest pamiętany w dodanym do struktury `AppData` polu `lighting_model`. Jego nowa wartość musi być nadana zmiennym jednolitym `LightingModel` w *obu* programach rysujących: dla siatek i dla płatów Béziera.

35.3. Pozostałe zmiany w aplikacji

Pozostałe zmiany w aplikacji są drobne i nieliczne. W części okienkowej trzeba utworzyć dodatkowy suwak, do sterowania długością paznokci, ale to wymaga tylko zmiany makrodefinicji `NKLARTPARAMS` (listing 35.1) opisującej liczbę parametrów artykulacji, zatem (poza drobną modyfikacją nazw procedur odzwierciedlającą zmianę nazwy aplikacji) część okienkowa pozostała niezmieniona.

Procedura `ProcessCharCommand` w części graficznej reaguje na polecenia wydawane przez napisanie liter `B` i `U`, które wybierają model oświetlenia (wywołując procedurę `Toggle-`



Rysunek 35.2. Sceny wyświetlane przez aplikację trzecią C

`LightModel`, listing 35.9) i przełączają obecność cieni na obrazie. Wreszcie, do procedury sprzątającej `DeleteMyWorld` są dodane instrukcje likwidujące dodatkowe programy szaderów. Zamieszczenie listingów tych procedur uznałem za zbędne.

35.4. Ćwiczenia

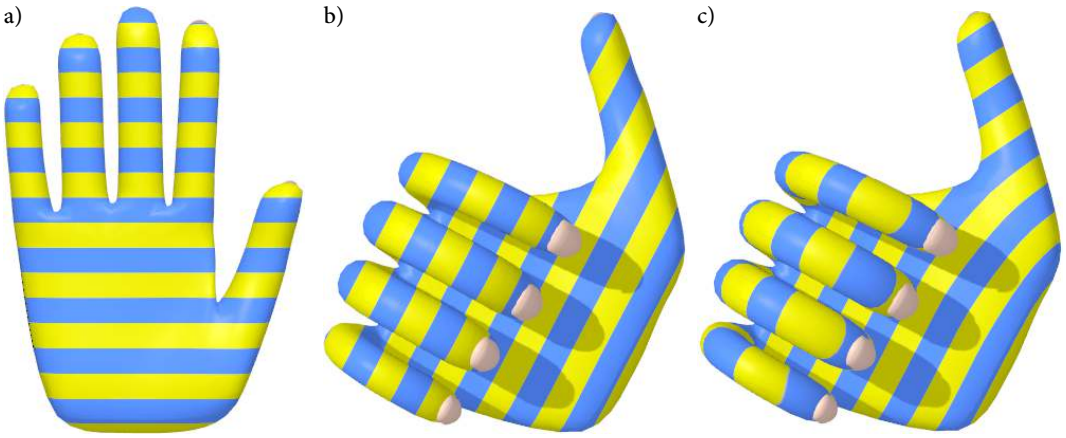
1. Zrób drugą (lewą) dłoń do kompletu i skonstruuj łańcuch kinematyczny umożliwiający niezależną artykulację palców obu dłoni. Paznokcie obu dłoni mogą mieć w każdej chwili tę samą długość.

   W menu nie ma miejsca na większą liczbę suwaków. Aby je pomieścić, najprościej jest utworzyć jeszcze jedno okno (podokno okna menu) o odpowiedniej wielkości i wyświetlać jego część wybieraną na przykład przy użyciu pionowego suwaka (a więc trzeba by dorobić taki rodzaj wihajstra).

2. Powierzchnia zbudowana ze ścian siatki jest zamknięta — jest brzegiem bryły. Korzystając z wiadomości podanych w podrozdziale 7.6, włącz przed rysowaniem siatki pomijanie ścian odwróconych tyłem do obserwatora (a przed rysowaniem paznokci je wyłącz).

35.5. Uzupełnienia — określanie parametrów tekstury

Rozważmy przykład nałożonej na dłoń tekstury. Szader fragmentów potrzebuje współrzędnych tekstury, aby obliczyć jej wartość. Jeśli te współrzędne określimy na podstawie współrzędnych punktu powierzchni w przestrzeni (np. w układzie świata), to po odkształceniu i zagęszczeniu siatki otrzymamy efekt taki jak na rysunku 35.3b: tekstura „przepłynie” po powierzchni, która będzie inaczej „pomalowana” niż przed odkształceniem.



Rysunek 35.3. Tekstura jednowymiarowa na powierzchni

Aby związać teksturę z powierzchnią tak jak farbę, trzeba określić współrzędne tekstury na podstawie współrzędnych punktów powierzchni nieodkształconej. W przykładzie pokazanym na rysunku 35.3 jest użyta tekstura jednowymiarowa, której argument (współrzędna tekstury) jest określony na podstawie współrzędnej y punktu powierzchni odkształconej na rysunku b i nieodkształconej na rysunku c. W przedstawionym tu eksperymencie każdy wierzchołek siatki miał 7 współrzędnych (tj. atrybutów skalarnych): trzy współrzędne wierzchołka siatki odkształconej w wyniku artykulacji, trzy współrzędne wektora normalnego i jedną dodatkową liczbę, która dla siatki niezagęszczonej (wygenerowanej przez szader artykulacji) była współrzędną y wierzchołka siatki oryginalnej, interpolowaną (tak samo jak współrzędne wierzchołków siatki) w kolejnych operacjach zagęszczania.

Pozostawiając Czytelnikowi do „rozgryzienia" modyfikacje szaderów wytwarzających reprezentację siatki do narysowania z wszystkimi potrzebnymi atrybutami wierzchołków, przedstawię sposób przygotowania i użycia tekstury jednowymiarowej. Tekstura użyta w opisanym tu eksperymencie została utworzona za pomocą instrukcji

```
glGenTextures ( 1, &mytxt );
glActiveTexture ( GL_TEXTURE0 );
glBindTexture ( GL_TEXTURE_1D, mytxt );
glTexParameteri ( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri ( GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri ( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexImage1D ( GL_TEXTURE_1D, 0, GL_RGBA, 64, 0, GL_RGBA,
               GL_FLOAT, mytexture );
```

W zmiennej `mytxt` jest zapamiętany identyfikator tekstury, natomiast tablica `mytexture` zawiera 64 liczby zmiennopozycyjne opisujące 16 tekseli; 8 w kolorze żółtym i 8 niebieskich. Argumentem tekstury jednowymiarowej jest jedna współrzędna, oznaczana literą s, przy czym podany parametr `GL_REPEAT` wybiera okresowe rozszerzenie tekstury określonej w przedziale $[0,1)$ na całą oś rzeczywistą[5]. Nie zastosowałem tu mipmapingu, choć to jest możliwe (i byłoby wskazane dla większych tekstur).

Szader fragmentów zawiera deklaracje

```
in FVertex {
  vec3 Position;
  ....
  float txtc;
} In;
layout(binding=0) uniform sampler1D mytxt;
```

a obliczenie tekstury wykonuje dodana do niego instrukcja

```
mm.dirref = texture ( mytxt, 5.0*In.txtc );
```

[5] W ten sposób powstały paski na obrazach.

która zapamiętuje kolor farby (tj. wartość tekstury) w odpowiednich polach zmiennej `mm` opisującej materiał (zobacz listing 18.1) na potrzeby obliczeń koloru oświetlonej powierzchni. Polu `txtc` bloku `FVertex` nadał wartość szader wierzchołków, wybierając odpowiednią współrzędną na podstawie wartości zmiennej jednolitej (a następnie etap rasteryzacji obliczył wartość tego pola dla każdego fragmentu). Nie należy zapomnieć o przywiązaniu tekstury do celu `GL_TEXTURE_1D` przed rysowaniem powierzchni i o posprzątaniu, gdy aplikacja kończy działanie.

36

Aplikacja trzecia D

Ostatnim tematem do przerabiania w tym kursie jest aplikacja 3D. Powstała ona z aplikacji 3C przez dodanie bardziej zaawansowanej animacji; na podstawie zadanych wartości parametrów artykulacji w wybranych chwilach aplikacja umożliwia określenie funkcji interpolacyjnej, której argumentem jest czas. Funkcję tę można „odgrywać", otrzymując ruch obiektu.

Nie ma w tej aplikacji nowych elementów OpenGL-a, ale za to jest bardziej rozbudowany interfejs użytkownika — konieczny element większości niebanalnych projektów. Dla części Czytelników będzie to przykład, jak można zrealizować taki interfejs, a dla innych, być może, będzie to przykład, jak tego robić nie należy. W ten sposób każdy znajdzie tu coś dla siebie.

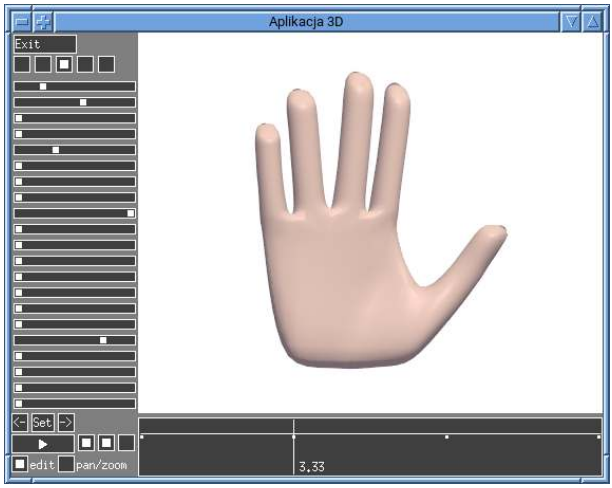
36.1. Działanie interfejsu użytkownika

Inaczej niż poprzednio, widok okna aplikacji umieściłem na początku rozdziału (rys. 36.1), aby najpierw przedstawić scenariusz działania interfejsu użytkownika, a implementację opisać dalej. Obszar głównego okna jest podzielony między trzy podokna, z których pierwsze i trzecie zawierają wihajstry rysowane przez procedury biblioteki X11, a okno drugie zawiera wihajster z obrazem wygenerowanym przy użyciu OpenGL-a. Wihajstry w pierwszym i drugim podoknie są przeniesione bez zmian z aplikacji 3C. Wihajstry w podoknie trzecim (zajmującym dolną część okna głównego) służą do konstruowania i odtwarzania animacji.

Guziki i przełączniki z lewej strony są „standardowymi" wihajstrami obsługiwanymi przez procedury opisane w podrozdziale 30.4, przy czym jeden z guzików ma podmienioną procedurę rysowania, bo zamiast opisu tekstowego ma na nim widnieć ikona (trójkąt lub dwa prostokąty) wskazująca na możliwość uruchomienia albo zatrzymania animacji. Prawą część podokna (poniżej obrazu sceny trójwymiarowej) zajmuje wihajster realizujący **oś czasu** aplikacji. Wihajster ten jest zaprojektowany specjalnie dla tej aplikacji i ma kilka trybów działania.

Obraz wihajstra składa się z linii poziomej (będącej wizualizacją osi czasu), zaznaczonych **węzłów**, w których są zadane wartości parametrów artykulacji[1], i dwóch pionowych

[1] W terminologii filmów animowanych chwilom tym odpowiadają tzw. klatki kluczowe animacji.



Rysunek 36.1. Okno aplikacji trzeciej D

kresek, z których jedna uwidocznia wybrany węzeł, a druga, opatrzona liczbą, określa chwilę wskazywaną przez położenie kursora lub bieżący czas odtwarzanej animacji.

Dwa przełączniki opatrzone napisami `edit` i `pan/zoom` włączają tryby **edycji** i **zmiany zakresu**. W trybie edycji użytkownik może wskazać kursorem węzeł i nacisnąć lewy przycisk myszy, co spowoduje przejście wihajstra do stanu, w którym przesuwanie myszy będzie powodować odpowiednie zmiany węzła. Wyjście z tego stanu następuje po zwolnieniu przycisku. Użytkownik może też nacisnąć prawy przycisk, co spowoduje dodanie nowego węzła w miejscu wskazanym przez kursor. Wihajster również wtedy wchodzi w stan przesuwania (nowego) węzła i pozostaje w tym stanie, dopóki prawy przycisk jest naciśnięty.

Naciśnięcie lewego przycisku w trybie zmiany zakresu powoduje wejście w stan przesuwania osi czasu — przesuwanie myszy w tym stanie powoduje przesuwanie w oknie początku układu współrzędnych (czyli zera na osi czasu). Naciśnięcie prawego przycisku wprowadza wihajster w stan najeżdżania lub odjeżdżania, czyli zmiany skali osi czasu. Można w ten sposób wybrać widoczny przedział osi czasu, aby w nim porozmieszczać węzły odpowiednio do potrzeb.

W trybie edycji, w stanie przesuwania węzła, użytkownik może spowodować usunięcie tego węzła, naciskając klawisz `Del`. Nie można w ten sposób pozostawić mniej niż czterech węzłów, ponieważ jest to minimalna liczba wymagana w używanych w tej aplikacji konstrukcjach interpolacyjnych krzywych sklejanych, opisanych w dodatku B.

Jeśli użytkownik nacisnął lewy przycisk, nie wskazując (wystarczająco dokładnie) węzła w trybie edycji lub gdy wihajster osi czasu nie jest w żadnym z opisanych wyżej trybów, to wihajster wchodzi w stan przesuwania chwili. W tym stanie aplikacja ma obliczać wartości parametrów artykulacji dla chwili odpowiadającej położeniu kursora i wyświetlać obraz obiektu odpowiadającego tej chwili. Przesuwając mysz, użytkownik może oglądać ruch w dowolnym tempie, w tym także cofać go.

Trzy guziki z napisami <-, Set i -> służą do zadawania warunków interpolacyjnych. Guziki ze strzałkami umożliwiają wybranie węzła. Po wybraniu go użytkownik powinien nadać (za pomocą suwaków w pierwszym podoknie) odpowiednie wartości parametrów artykulacji, a następnie pstryknąć guzik Set, co spowoduje zapamiętanie tych wartości dla bieżącego węzła. Dla chwili odpowiadającej temu węzłowi oprócz parametrów artykulacji zostanie zapamiętany bieżący obrót obserwatora wokół sceny.

Obok guzika uruchamiającego i zatrzymującego animację znajdują się trzy przełączniki. Służą one do niezależnego wybierania trzech animowanych elementów sceny: łańcucha kinematycznego, macierzy modelu i położenia obserwatora wokół sceny. Animacja łańcucha kinematycznego polega na obliczaniu, dla kolejnych chwil, wartości parametrów artykulacji i dokonywaniu artykulacji łańcucha. Parametry te opisuje wektorowa sklejana krzywa interpolacyjna trzeciego stopnia. Macierz modelu jest macierzą obrotu, którego kąt jest zmieniany ze stałą szybkością, jednej ósmej obrotu na sekundę. Animacja położenia obserwatora polega na interpolacji kwaternionów reprezentujących obroty w chwilach odpowiadających węzłom (zobacz podrozdz. A.4 i B.4).

36.2. Wihajster osi czasu

Wihajster osi czasu należy do części okienkowej aplikacji; przetwarza komunikaty otrzymywane od systemu X Window i jest rysowany za pomocą procedur z biblioteki X11. Aby przeniesienie aplikacji do innego systemu, na przykład Windows, było łatwiejsze, ciąg punktów na osi czasu oraz sposób jego zmieniania przez użytkownika nie powinien zależeć od środowiska. Uniezależnieniu części graficznej aplikacji, która musi mieć dostęp do danych i pewnych metod wihajstra, służy wprowadzenie dwóch plików nagłówkowych wihajstra, z których pierwszy (listing 36.1) zawiera dane widoczne dla części graficznej (w tym makrodefinicje wprowadzające nazwy i numery poleceń wydawanych przez wihajster osi czasu), a w drugim (listing 36.2) są zdefiniowane stany wihajstra, jego struktura danych związana ze środowiskiem i prototyp konstruktora (wywoływanego przez część okienkową).

Kolejne pola struktury KnotsWidgetf na listingu 36.1 przechowują odpowiednio minimalną i maksymalną dopuszczalną oraz bieżącą liczbę węzłów, numer węzła przesuwanego w danej chwili oraz numer poprzedniego przesuwanego węzła, poprzednią i bieżącą współrzędną położenia kursora w oknie, wskaźnik tablicy węzłów, odpowiadające sobie zakresy współrzędnych „świata", tj. osi czasu i okna, punkt osi czasu odpowiadający bieżącemu położeniu kursora i wreszcie objaśnione dalej przełączniki trybu działania wihajstra.

Struktura typu KnotsWidgetf będzie widocznym dla części okienkowej polem struktury AppWidgets, która jest opakowaniem wszystkich danych części graficznej. Struktura danych wihajstra osi czasu, xKnotsWidgetf zawiera wskaźnik tej struktury, a oprócz tego pole wdg typu xwidget, które umożliwia funkcjonowanie wihajstra w menu opisanym w rozdziale 30, i pole thebutton, które jest (stosowanym w systemie X Window) identyfikatorem aktualnie naciśniętego przycisku myszy[2].

[2]W języku C++ typ xKnotsWidgetf powinien być podklasą *klasy* xwidget, ale napisanie takiej implementacji pozostawiam amatorom tego języka.

Listing 36.1. Plik `knotswidget.h`

```C
#define WDGMSG_KNOT_CHANGE          20
#define WDGMSG_KNOT_INSERT          21
#define WDGMSG_KNOT_DELETE          22
#define WDGMSG_KNOT_MCLICK          23
#define WDGMSG_KNOT_MMOVE           24
#define WDGMSG_KNOT_CHANGE_MAPPING  25
#define WDGMSG_KNOT_ERROR           26

typedef struct {
    int   minknots, maxknots, nknots, current, prevc;
    int   prevxi, curxi;
    float *knots;
    float xmin, xmax, ximin, ximax, xc;
    char  editswitch, panswitch, motion_off;
  } KnotsWidgetf;

int KnotsWidgetXtoXi ( KnotsWidgetf *knw, float x );
float KnotsWidgetXitoX ( KnotsWidgetf *knw, int xi );
int FindKnotInterval ( int n, float *knots, float x );
```

Listing 36.2. Plik `xknotswidget.h`

```C
#define WDGSTATE_KW_MOVING_KNOT      10
#define WDGSTATE_KW_MOVING_MOUSE     11
#define WDGSTATE_KW_PANNING          12
#define WDGSTATE_KW_ZOOMING          13

typedef struct xKnotsWidgetf {
    xwidget      wdg;
    int          thebutton;
    KnotsWidgetf *knw;
  } xKnotsWidgetf;

xKnotsWidgetf *NewKnotsWidget ( xwinmenu *wm, KnotsWidgetf *knw,
                          int id, int w, int h, int x, int y,
                          int minknots, int maxknots,
                          int nknots, float *knots, float xmin, float xmax );
```

Dane „podłączone” do wihajstra osi czasu reprezentują rosnący ciąg (tzw. **węzłów**) liczb i przedział między najmniejszą a największą z nich, przy czym powiązanie liczb z tego przedziału z czasem jest pozostawione aplikacji. Liczby są typu `float`; pojedyncza precyzja jest wystarczająca, ponieważ czas będzie mierzony od początku animacji (która ma trwać od kilku sekund do najwyżej kilku minut), a nie od początku działania aplikacji, czego implementacja wymagałaby użycia podwójnej precyzji (zobacz uwagi na s. 77).

Liczby przechowywane $x_{\min}$, $x_{\max}$, $\xi_{\min}$ i $\xi_{\max}$ w polach `xmin`, `xmax`, `ximin` i `ximax` określają przejścia między układami współrzędnych węzłów i okna. Przejścia te są opisane wzorami

$$\xi = \xi_{\min} + \frac{\xi_{\max} - \xi_{\min}}{x_{\max} - x_{\min}}(x - x_{\min}), \tag{36.1}$$

$$x = x_{\min} + \frac{x_{\max} - x_{\min}}{\xi_{\max} - \xi_{\min}}(\xi - \xi_{\min}). \tag{36.2}$$

Listingi 36.3 i 36.4 przedstawiają procedurę przetwarzania komunikatów wihajstra osi czasu oraz (ze skrótami) wywoływane przez tę procedurę podprogramy pomocnicze. Obliczenie wykonane przez procedurę `KnotsWidgetInput` zależy od jednego z pięciu stanów wihajstra, rodzaju zdarzenia i trybu. Przyjrzyjmy się temu po kolei.

Listing 36.3. Procedura `KnotsWidgetInput`

C

```
1: static char KnotsWidgetInput ( struct xwidget *wdg,
2:                                int msg, int key, int x, int y )
3: {
4:   KnotsWidgetf  *knw;
5:   xKnotsWidgetf *xknw;
6:   int           dxi;
7:   float         dx;
8:
9:   xknw = (xKnotsWidgetf*)wdg;
10:  knw = xknw->knw;
11:  switch ( wdg->state ) {
12: case WDGSTATE_KW_MOVING_MOUSE:
13:    switch ( msg ) {
14:  case XWMSG_BUTTON_RELEASE:
15:      if ( key == xknw->thebutton )
16:        goto exit_active_state;
17:      break;
18:  case XWMSG_MOUSE_MOTION:
19:      knw->xc = XitoX ( knw, knw->curxi = x );
20:      wdg->wm->callback ( wdg, WDGMSG_KNOT_MMOVE, knw->current, x, y );
21:      wdg->wm->changed = true;
22:      break;
23:  default:
24:      break;
25:    }
26:    break;
```

W stanie podstawowym, jeśli przełącznik `editswitch` ma wartość `true` i kursor wskazuje pewien węzeł na osi czasu, to po naciśnięciu lewego przycisku myszy wihajster przechodzi w stan przesuwania węzła (linia 128), a w przeciwnym razie przechodzi w stan przesuwania myszy (linia 132 lub 148).

Instrukcje przetwarzania komunikatów w stanie przesuwania myszy są w liniach 12–26; zwolnienie przycisku powoduje powrót do stanu podstawowego, a przemieszczenie kursora powoduje, w linii 20, przekazanie aplikacji polecenia WDGMSG_KNOT_MMOVE. Część graficzna aplikacji reaguje na to polecenie, dokonując artykulacji łańcucha kinematycznego dla chwili odpowiadającej nowemu położeniu kursora i wyświetlając nowy obraz.

Listing 36.3. (cd.) Procedura KnotsWidgetInput

C

```
27: case WDGSTATE_KW_MOVING_KNOT:
28:     switch ( msg ) {
29:   case XWMSG_BUTTON_RELEASE:
30:       if ( key == xknw->thebutton )
31:         goto exit_active_state;
32:       break;
33:   case XWMSG_MOUSE_MOTION:
34:       knw->xc = XitoX ( knw, knw->curxi = x );
35:       UpdateTheKnot ( knw, x );
36:       wdg->wm->callback ( wdg, WDGMSG_KNOT_CHANGE, knw->current, x, y );
37:       wdg->wm->changed = true;
38:       break;
39:   case XWMSG_KEY_PRESS:
40:       switch ( key ) {
41:     case 0x007f:  /* ASCII Del, not defined in keysymdef.h */
42:         goto delete_knot;
43:     default:
44:         break;
45:       }
46:       break;
47:   case XWMSG_SPECIAL_KEY_PRESS:
48:       switch ( key ) {
49:     case XK_KP_Delete:
50: delete_knot:
51:         if ( DeleteTheKnot ( knw ) ) {
52:           wdg->wm->callback ( wdg, WDGMSG_KNOT_DELETE, knw->current, x, y );
53:           goto exit_active_state;
54:         }
55:         break;
56:     default:
57:         break;
58:       }
59:       break;
60:     }
61:     break;
```

Wihajster może przejść w stan przesuwania węzła także po naciśnięciu prawego przycisku (linia 141); wtedy w miejscu wskazywanym przez kursor jest wstawiany nowy węzeł (przez

procedurę `InsertKnot`, linia 138) i ten węzeł staje się bieżący. Instrukcje przetwarzające komunikaty wejściowe są zapisane w liniach 27–61. Zwolnienie przycisku powoduje powrót do stanu podstawowego (linia 31). Przemieszczenie kursora powoduje przesunięcie węzła przez procedurę `UpdateTheKnot` i wysłanie aplikacji polecenia `WDGMSG_KNOT_CHANGE` (linie 35, 36). Naciśnięcie klawisza `Del` lub `Delete` (które może spowodować wysłanie kodu ASCII `Del` lub komunikatu o naciśnięciu klawisza specjalnego) powoduje usunięcie bieżącego węzła (przez procedurę `DeleteKnot`), zawiadomienie aplikacji (poleceniem `WDGMSG_KNOT_DELETE`) i powrót wihajstra do stanu podstawowego. Procedura `DeleteTheKnot` nie usuwa węzła, jeśli miałoby to spowodować zmniejszenie liczby węzłów poniżej dopuszczalnego minimum.

Listing 36.3. (cd.) Procedura `KnotsWidgetInput`

C

```
62: case WDGSTATE_KW_PANNING:
63:     switch ( msg ) {
64:   case XWMSG_MOUSE_MOTION:
65:       if ( (dxi = x - knw->prevxi) ) {
66:         knw->prevxi = x;
67:         dx = dxi*(knw->xmax-knw->xmin)/(knw->ximax-knw->ximin);
68:         knw->xmin -= dx;  knw->xmax -= dx;
69:         wdg->wm->callback ( wdg, WDGMSG_KNOT_CHANGE_MAPPING, 0, x, y );
70:         wdg->wm->changed = true;
71:       }
72:       break;
73:   case XWMSG_BUTTON_RELEASE:
74:       if ( key == xknw->thebutton )
75:         goto exit_active_state;
76:       break;
77:   default:
78:       break;
79:     }
80:     break;
```

Pozostałym dwóm stanom aktywnym wihajstra dałem nazwy `WDGSTATE_KW_PANNING` i `WDGSTATE_KW_ZOOMING`. Stany te umożliwiają zmianę widocznego w obrazie wihajstra przedziału na osi czasu. Wihajster wchodzi w te stany odpowiednio po naciśnięciu lewego i prawego przycisku myszy, gdy przełącznik `panswitch` ma wartość `true`. Zwolnienie przycisku powoduje powrót do stanu podstawowego. Przesunięcie kursora w stanie `WDGSTATE_KW_PANNING` powoduje dodanie do obu końców widocznego przedziału przyrostu odpowiadającego przyrostowi współrzędnej ξ położenia kursora (linia 68). Skutkiem przesunięcia kursora w stanie `WDGSTATE_KW_ZOOMING` jest zmiana długości przedziału o czynnik obliczany w linii 86 na podstawie przyrostu współrzędnej ξ oraz bieżącej szerokości obrazu wihajstra. W obu przypadkach aplikacja jest zawiadamiana o zmianie odwzorowania współrzędnych ekranowych na czas, przez wysłanie polecenia `WDGMSG_KNOT_CHANGE_MAPPING`.

Listing 36.3. (cd.) Procedura `KnotsWidgetInput`

C

```
case WDGSTATE_KW_ZOOMING:
    switch ( msg ) {
  case XWMSG_MOUSE_MOTION:
      if ( (dxi = knw->prevxi - x) ) {
        knw->prevxi = x;
        dx = exp ( (float)dxi/(knw->ximax-knw->ximin) );
        knw->xmax = knw->xmin + dx*(knw->xmax-knw->xmin);
        wdg->wm->callback ( wdg, WDGMSG_KNOT_CHANGE_MAPPING, 1, x, y );
        wdg->wm->changed = true;
      }
      break;
  case XWMSG_BUTTON_RELEASE:
      if ( key == xknw->thebutton ) {
exit_active_state:
        wdg->state = WDGSTATE_DEFAULT;
        UngrabInput ( wdg );
        wdg->wm->changed = true;
      }
      break;
  default:
      break;
    }
    break;
```

Przejście ze stanu podstawowego do każdego z czterech stanów aktywnych wiąże się z przejęciem komunikatów przez wihajster (za pomocą procedury `GrabInput`, linia 152). Podczas powrotu do stanu podstawowego jest wywoływana procedura `UngrabInput` (linia 96). W stanie podstawowym wihajster reaguje na naciśnięcie klawisza z literą `R`, przywracając wyświetlanie domyślnego przedziału $[0, 10]$, i klawisza z literą `F`, które powoduje wyświetlanie przedziału między pierwszym a ostatnim węzłem.

Przesuwanie kursora w stanie podstawowym powoduje zmiany obrazu wihajstra, w którym zmienia się pionowa linia i liczba opisująca wskazywaną chwilę czasową. Ale ta reakcja na przesuwanie kursowa może być wyłączona przez aplikację (przez przypisanie wartości `true` przełącznikowy `motion_off`), co jest pożądane podczas „odgrywania” animacji.

Listing 36.3. (cd.) Procedura `KnotsWidgetInput`

C

```
default:
    switch ( msg ) {
  case XWMSG_MOUSE_MOTION:
      if ( !knw->motion_off ) {
        knw->xc = XitoX ( knw, knw->curxi = x );
        wdg->wm->changed = true;
      }
```

```
111:       break;
112:   case XWMSG_BUTTON_PRESS:
113:       knw->xc = XitoX ( knw, knw->curxi = x );
114:       xknw->thebutton = key;
115:       if ( knw->panswitch ) {
116:         if ( key == Button1 )
117:           wdg->state = WDGSTATE_KW_PANNING;
118:         else if ( key == Button3 )
119:           wdg->state = WDGSTATE_KW_ZOOMING;
120:         else
121:           break;
122:         goto enter_active_state;
123:       }
124:       else if ( knw->editswitch ) {
125:         if ( key == Button1 ) {
126:           wdg->wm->callback ( wdg, WDGMSG_KNOT_MCLICK, 0, x, y );
127:           if ( FindNearestKnot ( knw, x ) ) {
128:             wdg->state = WDGSTATE_KW_MOVING_KNOT;
129:             goto enter_active_state;
130:           }
131:           else {
132:             wdg->state = WDGSTATE_KW_MOVING_MOUSE;
133:             goto enter_active_state;
134:           }
135:         }
136:         else if ( key == Button3 ) {
137:           wdg->wm->callback ( wdg, WDGMSG_KNOT_MCLICK, 1, x, y );
138:           if ( InsertKnot ( knw, x ) ) {
139:             wdg->wm->callback ( wdg, WDGMSG_KNOT_INSERT,
140:                                 knw->current, x, y );
141:             wdg->state = WDGSTATE_KW_MOVING_KNOT;
142:             goto enter_active_state;
143:           }
144:         }
145:       }
146:       else {
147:         if ( key == Button1 ) {
148:           wdg->state = WDGSTATE_KW_MOVING_MOUSE;
149:           wdg->wm->callback ( wdg, WDGMSG_KNOT_MCLICK, 2, x, y );
150: enter_active_state:
151:           knw->prevxi = x;
152:           GrabInput ( wdg );
153:           wdg->wm->changed = true;
154:         }
155:       }
156:       break;
157:   case XWMSG_KEY_PRESS:
```

```
158:       switch ( key ) {
159:     case 'R':  case 'r':
160:         knw->xmin = 0.0;  knw->xmax = 10.0;
161:         wdg->wm->callback ( wdg, WDGMSG_KNOT_CHANGE_MAPPING, 1, x, y );
162:         wdg->wm->changed = true;
163:         break;
164:     case 'F':  case 'f':
165:         if ( knw->knots[knw->nknots-1] != knw->knots[0] ) {
166:           knw->xmin = knw->knots[0];  knw->xmax = knw->knots[knw->nknots-1];
167:           wdg->wm->callback ( wdg, WDGMSG_KNOT_CHANGE_MAPPING, 1, x, y );
168:           wdg->wm->changed = true;
169:         }
170:         break;
171:     default:
172:         return false;
173:       }
174:       break;
175:   case WDGMSG_RECONFIGURE:
176:       if ( key )
177:         { wdg->r.x = x;  wdg->r.y = y; }
178:       else
179:         { wdg->r.width = x;  wdg->r.height = y; }
180:       knw->ximin = wdg->r.x + 3;  knw->ximax = wdg->r.x + wdg->r.width - 4;
181:       wdg->wm->changed = true;
182:     }
183:     break;
184:   }
185:   return true;
186: } /*KnotsWidgetInput*/
```

W stanie podstawowym wihajster reaguje na komunikat `WDGMSG_RECONFIGURE`, który zawiadamia o zmianie wymiarów — zależnie od wartości (`true` albo `false`) parametru `key`, parametry `x` i `y` określają współrzędne górnego lewego narożnika albo nowe wymiary wihajstra.

Zobaczmy teraz podprogramy pomocnicze wihajstra pokazane na listingu 36.4. Funkcje `KnotsWidgetXtoXi` i `KnotsWidgetXitoX` (listing 36.4) realizują odpowiednio wzory (36.1) i (36.2), przy czym w pierwszym przypadku wynik jest zaokrąglany do najbliższej liczby całkowitej. Lokalnie (w procedurze `KnotsWidgetInput`) funkcje te występują pod krótszymi nazwami `XtoXi` i `XitoX`.

Funkcja `FindKnotInterval` dla tablicy `knots` o długości $n+1$, zawierającej rosnący ciąg liczb $x_0, \ldots, x_n$, i liczby x znajduje (metodą bisekcji) liczbę i, taką że $x_i \leqslant x < x_{i+1}$, przy czym jeśli $x < x_0$, to $i = -1$, a jeśli $x \geqslant x_n$, to $i = n$.

Procedura `FindNearestKnot` znajduje węzeł, którego obraz (wyświetlony przez wihajster) znajduje się najbliżej kursora. Jeśli różnica współrzędnych ξ kursora i obrazu węzła nie jest większa niż próg tolerancji 5 pikseli, to procedura zapamiętuje numer tego węzła w polu

Listing 36.4. Pomocnicze procedury wihajstra osi czasu

C

```
int KnotsWidgetXtoXi ( xKnotsWidgetf *knw, float x ) { .... }
float KnotsWidgetXitoX ( xKnotsWidgetf *knw, int xi ) { .... }
#define XtoXi KnotsWidgetXtoXi
#define XitoX KnotsWidgetXitoX

int FindKnotInterval ( float n, float *knots, float x ) { .... }

static char FindNearestKnot ( xKnotsWidgetf *knw, int xi )
{
#define TOL 5
  int   i;
  float x, *knots;

  i = FindKnotInterval ( knw->nknots, knots = knw->knots,
                         x = XitoX ( knw, xi) );
  if ( i < 0 )
    i = 0;
  if ( i < knw->nknots-1 )
    if ( x-knots[i] > knots[i+1]-x )
      i++;
  if ( fabs ( x-knots[i] )*
       (knw->ximax-knw->ximin)/(knw->xmax-knw->xmin) <= TOL ) {
    knw->current = i;
    return true;
  }
  return false;
#undef TOL
} /*FindNearestKnot*/

static void ModifyTheKnot ( int nknots, float *knots, int i )
{
#define TOL 0.02
  float x, x0, x1, h;

  x = knots[i];
  x0 = i == 0 ? x : knots[i-1];
  x1 = i == nknots-1 ? x : knots[i+1];
  h = x1-x0;
  if ( x < x0+TOL*h )      knots[i] = x0+TOL*h;
  else if ( x > x1-TOL*h ) knots[i] = x1-TOL*h;
#undef TOL
} /*ModifyTheKnot*/

static void UpdateTheKnot ( xKnotsWidgetf *knw, int xi )
{
```

```
46:   float x, px, *knots;
47:   int   n, c;
48:
49:   knots = knw->knots;
50:   x = XitoX ( knw, xi );
51:   px = knots[c = knw->prevc = knw->current];
52:   if ( x < px ) {
53:     while ( c > 0 && x < knots[c-1] ) {
54:       knots[c] = knots[c-1];
55:       c --;
56:     }
57:   }
58:   else if ( x > px ) {
59:     n = knw->nknots;
60:     while ( c < n-1 && x > knots[c+1] ) {
61:       knots[c] = knots[c+1];
62:       c ++;
63:     }
64:   }
65:   knots[knw->current = c] = x;
66:   ModifyTheKnot ( knw->nknots, knots, c );
67: } /*UpdateTheKnot*/
68:
69: static char InsertKnot ( xKnotsWidgetf *knw, int xi )
70: {
71:   float x, *knots;
72:   int   i;
73:
74:   if ( knw->nknots >= knw->maxknots )
75:     return false;
76:   i = FindKnotInterval ( knw->nknots, knots = knw->knots,
77:                          x = XitoX ( knw, xi ) );
78:   if ( i < knw->nknots-1 )
79:     memmove ( &knots[i+2], &knots[i+1], (knw->nknots-1-i)*sizeof(float) );
80:   knots[knw->prevc = knw->current = i+1] = x;
81:   ModifyTheKnot ( ++knw->nknots, knots, i+1 );
82:   return true;
83: } /*InsertKnot*/
84:
85: static char DeleteTheKnot ( xKnotsWidgetf *knw )
86: {
87:   float *knots;
88:
89:   if ( knw->nknots > knw->minknots &&
90:        knw->current >= 0 && knw->current < knw->nknots ) {
91:     if ( knw->current < knw->nknots-1 ) {
92:       knots = knw->knots;
```

```
93:       memmove ( &knots[knw->current], &knots[knw->current+1],
94:                 (knw->nknots-1-knw->current)*sizeof(float) );
95:     }
96:     knw->nknots --;
97:     return true;
98:   }
99:   return false;
100: } /*DeleteTheKnot*/
```

current i zawiadamia procedurę KnotsWidgetInput, że węzeł został wkazany wystarczająco dokładnie, co umożliwi wejście w stan przesuwania węzła.

Procedura UpdateTheKnot oblicza liczbę x odpowiadającą współrzędnej ξ punktu w oknie i wpisuje ją do tablicy węzłów z zachowaniem uporządkowania ciągu liczb w tej tablicy. W tym celu, w liniach 53–56 albo 59–63 może przesunąć liczby w tablicy w odpowiednią stronę. Zadaniem procedury ModifyTheKnot, wywoływanej po wpisaniu węzła do tablicy, jest niedopuszczenie do nałożenia się węzłów, ponieważ ciąg węzłów interpolacyjnych musi być ściśle rosnący. Co więcej, długości sąsiednich przedziałów między węzłami nie powinny zbytnio się różnić. Dlatego procedura ModifyTheKnot dokonuje takiej modyfikacji, aby odległość węzła od jego sąsiadów nie była mniejsza niż 0.02 długości przedziału między tymi sąsiadami.

Procedura InsertKnot wstawia węzeł do ciągu, a DeleteTheKnot usuwa węzeł, dokonując odpowiednich przesunięć w tablicy. Procedura InsertKnot dodatkowo wywołuje procedurę ModifyTheKnot, aby uniknąć sytuacji, w której węzły pokrywają się lub są położone zbyt blisko siebie.

Listing 36.5 przedstawia nagłówek procedury rysowania wihajstra osi czasu i jego konstruktor, przy czym treść procedury rysowania pominąłem; procedura ta kolejno wywołuje procedury z biblioteki X11, które wyświetlają wihajster (w tym obrazy węzłów) w trybie natychmiastowym. Ewentualną ciekawość można zaspokoić, czytając plik źródłowy.

Listing 36.5. Procedury KnotsWidgetRedraw i NewKnotsWidget

C

```
1: static void KnotsWidgetRedraw ( struct xwidget *wdg ) { .... }
2:
3: xKnotsWidgetf *NewKnotsWidget ( xwinmenu *wm, KnotsWidgetf *knw,
4:                                 int id, int w, int h, int x, int y,
5:                                 int minknots, int maxknots, int nknots,
6:                                 float *knots, float xmin, float xmax )
7: {
8:   xKnotsWidgetf *wdg;
9:
10:  wdg = (xKnotsWidgetf*)NewWidget ( wm, sizeof(xKnotsWidgetf), id, w, h,
11:                    x, y, KnotsWidgetInput, KnotsWidgetRedraw, NULL, NULL );
12:  wdg->knw = knw;
13:  knw->minknots = minknots;
14:  knw->maxknots = maxknots;
```

```
15:   knw->nknots = nknots;
16:   knw->knots = knots;
17:   knw->xmin = xmin;
18:   knw->xmax = xmax;
19:   knw->ximin = x+3;
20:   knw->ximax = x+w-4;
21:   knw->panswitch = knw->motion_off = false;
22:   return wdg;
23: } /*NewKnotsWidget*/
```

Procedura `NewKnotsWidget`, czyli konstruktor wihajstra, za pomocą procedury `NewWidget` tworzy i włącza wihajster do menu przekazanego jako parametr. Parametr `knw` wskazuje strukturę typu `KnotsWidget`, która jest widoczną dla części graficznej aplikacji i niezależną od środowiska okienkowego częścią wihajstra osi czasu. Po zarezerwowaniu pamięci na wihajster działający w menu, wartości początkowe określone przez parametry, w tym adres i długość tablicy z węzłami, są przypisywane polom tej struktury.

36.3. Procedury obsługi animacji

Listing 36.6 przedstawia zmiany struktur `AppWidgets` i `AppData` części graficznej aplikacji 3C, w wyniku których powstały struktury aplikacji 3D. Do struktury `AppWidgets`, widocznej w części okienkowej, są dodane przełączniki elementów animacji oraz pole `kw`, przechowujące dane wihajstra osi czasu. Struktura `AppData` ma nowe pola `lastkeyframe` i `lastbsknot`, których wartości to numer M ostatniego węzła interpolacyjnego i numer N ostatniego węzła krzywej sklejanej skonstruowanej w celu interpolowania parametrów artykulacji. Pola `keyknots`, `keyparams` i `qparams` wskazują tablice węzłów interpolacyjnych, wyznaczających chwile klatek kluczowych, i tablice parametrów artykulacji łańcucha kinematycznego i kwaternionów opisujących obroty obserwatora w tych chwilach. Pola `bsknots`, `bsparams` i `qbsparams` wskazują tablice węzłów krzywych sklejanych i punktów kontrolnych tych krzywych, konstruowanych na podstawie danych przechowywanych we wcześniej opisanych tablicach.

Pola `animate_mm`, `animate_vp` i `animate_kl` struktury `AppWidgets` mają wartości nadawane przez przełączniki w menu. Jeśli pierwsze z nich ma wartość `true`, to będzie animowana macierz przekształcenia modelu (czyli ruch obrotowy sceny w układzie świata ze stałą prędkością kątową). Wartość `true` pola `animate_vp` oznacza życzenie animowania ruchu obserwatora wokół sceny, a jeśli pole `animate_kl` ma wartość `true`, to będzie animowany łańcuch kinematyczny.

Procedury obsługi animacji aplikacji 3D są odpowiedzialne za przetwarzanie parametrów artykulacji łańcucha kinematycznego i (reprezentowanych przez kwaterniony) obrotów obserwatora wokół sceny odpowiadających klatkom kluczowym, a także za konstruowanie sklejanych funkcji interpolacyjnych i obliczanie, dla podanego czasu, wartości tych funk-

Listing 36.6. Struktury danych aplikacji 3D

C

```
#define MAXKEYFRAMES  100

typedef struct {
    char          sw[NPALMMESHES+1];
    float         artp[NKLARTPARAMS];
    char          animate_mm, animate_vp, animate_kl;
    KnotsWidgetf kw;
    char          animation;
  } AppWidgets;

typedef struct {
    .... /* pola takie, jak w liniach 16-27 na listingu 35.1 */
    int                   lastkeyframe, lastbsknot;
    float                 *keyknots, *keyparams, *qparams,
                          *bsknots, *bsparams, *qbsparams;
    char                  bs_ok, qs_ok;
    .... /* opakowania programów szaderów */
         /* identyczne jak w liniach 28-30 na listingu 35.1 */
  } AppData;
```

cji, czyli parametrów artykulacji podczas animacji. Procedury te są niezależne zarówno od środowiska okienkowego, jak i od sposobu tworzenia grafiki; ich głównym zadaniem jest przechowywanie danych i zorganizowanie odpowiednich obliczeń numerycznych[3].

Listing 36.7 przedstawia procedury wywoływane (za pośrednictwem menu) w odpowiedzi na komunikaty przesyłane przez wihajster osi czasu. Procedury `UpdateKeyInterpSpline` i `UpdateKeyInterpQSpline`, wywoływane po zmianie węzłów lub warunków interpolacyjnych, konstruują (przy użyciu procedur opisanych w podrozdz. B.3 i B.4) reprezentacje krzywych interpolacyjnych: B-sklejanej i kwaternionowej. Zwróćmy uwagę, że konstruowana krzywa B-sklejana leży w przestrzeni o wymiarze równym liczbie parametrów artykulacji łańcucha kinematycznego (`NKLARTPARAMS`, czyli 21). Taką liczbę współrzędnych mają zarówno punkty tej krzywej (czyli wektory parametrów artykulacji), jak i punkty kontrolne obliczane przez procedurę `ConstructCubicInterpBSplinef`.

Instrukcje w liniach 12–15 mają na celu sprawienie, aby kąty między wektorami (kwaternionami) reprezentującymi kolejne zadane położenia w ruchu obrotowym nie przekraczały $\pi/2$; w tym celu zwroty pewnych wektorów mogą być zamienione na przeciwne. Dowolny obrót w przestrzeni jest reprezentowany przez dwa kwaterniony o przeciwnych znakach. Dokonany wybór znaków ma na celu określenie ruchu tak, aby obiekt między kolejnymi zadanymi położeniami obracał się o mniejszy kąt (zobacz rys. A.4 i uwagę na s. 1053).

Procedura `InitKeyFrames`, wywołana na początku działania aplikacji, rezerwuje pamięć i nadaje wartości początkowe zmiennym opisującym węzły i warunki interpolacyjne.

[3] Dlatego procedury te są umieszczone w osobnym pliku źródłowym, w którym podczas dostosowania aplikacji do standardu Vulkan lub DirectX nie trzeba byłoby wprowadzać *żadnych* zmian.

Listing 36.7. Procedury edycji klatek kluczowych

C

```
void UpdateKeyInterpSpline ( AppData *ad )
{
  ad->bs_ok = ConstructCubicInterpBSplinef ( &ad->lastbsknot,
                    ad->bsknots, ad->bsparams, ad->lastkeyframe,
                    ad->keyknots, NKLARTPARAMS, ad->keyparams );
} /*UpdateKeyInterpSpline*/

void UpdateKeyInterpQSpline ( AppData *ad )
{
  int i, j, k;

  for ( i = 1, k = 4;  i <= ad->lastkeyframe;  i++, k += 4 )
    if ( V4DotProductf ( &ad->qparams[k-4], &ad->qparams[k] ) < 0.0 )
      for ( j = k; j < k+4; j++ )
        ad->qparams[j] = -ad->qparams[j];
  ad->qs_ok = ConstructQuaternionInterpSplinef ( &ad->lastbsknot,
                    ad->bsknots, ad->qbsparams, ad->lastkeyframe,
                    ad->keyknots, ad->qparams );
} /*UpdateKeyInterpQSpline*/

char InitKeyFrames ( AppData *ad )
{
  int i;

  ad->keyknots = malloc ( MAXKEYFRAMES*sizeof(float) );
  ad->keyparams = malloc ( MAXKEYFRAMES*NKLARTPARAMS*sizeof(float) );
  ad->bsknots = malloc ( (MAXKEYFRAMES+6)*sizeof(float) );
  ad->bsparams = malloc ( (MAXKEYFRAMES+2)*NKLARTPARAMS*sizeof(float) );
  ad->qparams = malloc ( MAXKEYFRAMES*4*sizeof(float) );
  ad->qbsparams = malloc ( (MAXKEYFRAMES+2)*4*sizeof(float) );
  if ( ad->keyknots && ad->keyparams && ad->qparams &&
       ad->bsknots && ad->bsparams && ad->qbsparams ) {
    ad->keyknots[0] = 0.0;
    ad->keyknots[1] = 10.0/3.0;
    ad->keyknots[2] = 20.0/3.0;
    ad->keyknots[3] = 10.0;
    ad->wdg.kw.nknots = (ad->lastkeyframe = 3) + 1;
    ad->wdg.kw.knots = ad->keyknots;
    for ( i = 0; i <= ad->lastkeyframe; i++ )
      memcpy ( &ad->keyparams[i*NKLARTPARAMS], ad->wdg.artp,
               NKLARTPARAMS*sizeof(float) );
    memset ( ad->qparams, 0, (ad->lastkeyframe+1)*4*sizeof(float) );
    for ( i = 0; i <= ad->lastkeyframe; i++ )
      ad->qparams[4*i] = 1.0;
    UpdateKeyInterpSpline ( ad );
```

```
46:     UpdateKeyInterpQSpline ( ad );
47:     ad->wdg.animate_mm = ad->wdg.animate_kl = true;
48:     ad->wdg.animate_vp = false;
49:     return true;
50:   }
51:   else {
52:     CleanupKeyFrames ( ad );
53:     return false;
54:   }
55: } /*InitKeyFrames*/
56:
57: void CleanupKeyFrames ( AppData *ad )
58: {
59:   .... /* zwolnij wszystkie 6 tablic */
60: } /*CleanupKeyFrames*/
61:
62: void FindKeyFrame ( AppData *ad, char right )
63: {
64:   AppWidgets   *aw;
65:   KnotsWidgetf *kw;
66:   int          c;
67:   float        v[3];
68:   double       a;
69:
70:   aw = &ad->wdg;
71:   kw = &aw->kw;
72:   c = FindKnotInterval ( kw->nknots, ad->keyknots, kw->xc );
73:   if ( c < 0 )
74:     c = 0;
75:   else if ( c >= kw->nknots )
76:     c = kw->nknots-1;
77:   if ( !right && c > 0 && kw->xc == ad->keyknots[c] )
78:     c --;
79:   else if ( right && c < kw->nknots-1 )
80:     c ++;
81:   kw->current = c;
82:   kw->curxi = KnotsWidgetXtoXi ( kw, kw->xc = ad->keyknots[c] );
83:   memcpy ( aw->artp, &ad->keyparams[c*NKLARTPARAMS],
84:            NKLARTPARAMS*sizeof(float) );
85:   ArticulatePalmLinkage ( ad );
86:   RotVQuatf ( v, &a, &ad->qparams[c*4] );
87:   SetupViewMatrix ( ad, v, a );
88: } /*FindKeyFrame*/
89:
90: void SetKeyFrame ( AppData *ad )
91: {
92:   AppWidgets   *aw;
```

```
93:   KnotsWidgetf *kw;
94:   int           c;
95:
96:   aw = &ad->wdg;
97:   kw = &aw->kw;
98:   c = kw->current;
99:   if ( (c = kw->current) >= 0 && c < kw->nknots ) {
100:    memcpy ( &ad->keyparams[c*NKLARTPARAMS], aw->artp,
101:             NKLARTPARAMS*sizeof(float) );
102:    UpdateKeyInterpSpline ( ad );
103:    QuatRotVf ( &ad->qparams[c*4],
104:                ad->camera.viewer_rvec, ad->camera.viewer_rangle );
105:    UpdateKeyInterpQSpline ( ad );
106:  }
107: } /*SetKeyFrame*/
108:
109: void ClampArtParams ( float *params )
110: {
111:   int i;
112:
113:   for ( i = 0; i < NKLARTPARAMS; i++ )
114:     if ( params[i] < 0.0 ) params[i] = 0.0;
115:     else if ( params[i] > 1.0 ) params[i] = 1.0;
116: } /*ClampArtParams*/
117:
118: void InsertKeyFrame ( AppData *ad )
119: {
120:   AppWidgets   *aw;
121:   KnotsWidgetf *kw;
122:   int           c, n;
123:
124:   aw = &ad->wdg;
125:   kw = &aw->kw;
126:   ad->lastkeyframe = kw->nknots-1;
127:   if ( (c = kw->current) < (n = kw->nknots)-1 ) {
128:     memmove ( &ad->keyparams[(c+1)*NKLARTPARAMS],
129:               &ad->keyparams[c*NKLARTPARAMS],
130:               (n-c-1)*NKLARTPARAMS*sizeof(float) );
131:     BSCdeBoorf ( 3, ad->lastbsknot, ad->bsknots, NKLARTPARAMS,
132:                  ad->bsparams, ad->keyknots[c], aw->artp );
133:     ClampArtParams ( aw->artp );
134:     memcpy ( &ad->keyparams[c*NKLARTPARAMS], aw->artp,
135:              NKLARTPARAMS*sizeof(float) );
136:     memmove ( &ad->qparams[(c+1)*4], &ad->qparams[c*4],
137:               (n-c-1)*4*sizeof(float) );
138:     if ( ad->qs_ok )
139:       QuatSlerpdeBoorf ( 3, ad->lastbsknot, ad->bsknots, ad->qbsparams,
```

```
140:                            ad->keyknots[c], &ad->qparams[c*4] );
141:   }
142:   else {
143:     memcpy ( &ad->keyparams[c*NKLARTPARAMS],
144:            &ad->keyparams[(c-1)*NKLARTPARAMS], NKLARTPARAMS*sizeof(float) );
145:     memcpy ( &ad->qparams[c*4], &ad->qparams[(c-1)*4], 4*sizeof(float) );
146:   }
147:   UpdateKeyInterpSpline ( ad );
148:   ArticulatePalmLinkage ( ad );
149:   QuatRotVf ( &ad->qparams[c*4],
150:               ad->camera.viewer_rvec, ad->camera.viewer_rangle );
151:   UpdateKeyInterpQSpline ( ad );
152: } /*InsertKeyFrame*/
153:
154: void SwapKeyFrames ( AppData *ad, int i, int j )
155: {
156:   int   k, l, m;
157:   float *kp, a;
158:
159:   for ( l = i*NKLARTPARAMS, m = j*NKLARTPARAMS, k = 0;
160:         k < NKLARTPARAMS;
161:         k ++ )
162:     { a = kp[l+k];  kp[l+k] = kp[m+k];  kp[m+k] = a; }
163:   kp = ad->qparams;
164:   for ( l = i*4, m = j*4, k = 0;  k < 4; k++ )
165:     { a = kp[l+k];  kp[l+k] = kp[m+k];  kp[m+k] = a; }
166: } /*SwapKeyFrames*/
167:
168: void ChangeKeyFrame ( AppData *ad )
169: {
170:   AppWidgets   *aw;
171:   KnotsWidgetf *kw;
172:   int          i, p, c;
173:
174:   aw = &ad->wdg;
175:   kw = &aw->kw;
176:   if ( (p = kw->prevc) < (c = kw->current) ) {
177:     for ( i = p; i < c; i++ )
178:       SwapKeyFrames ( ad, i, i+1 );
179:   }
180:   else if ( p > c ) {
181:     for ( i = p; i > c; i-- )
182:       SwapKeyFrames ( ad, i, i-1 );
183:   }
184:   UpdateKeyInterpSpline ( ad );
185:   UpdateKeyInterpQSpline ( ad );
186: } /*ChangeKeyFrame*/
```

```
187:
188: void DeleteKeyFrame ( AppData *ad )
189: {
190:   AppWidgets   *aw;
191:   KnotsWidgetf *kw;
192:   int           c, n;
193:
194:   aw = &ad->wdg;
195:   kw = &aw->kw;
196:   ad->lastkeyframe = kw->nknots-1;
197:   if ( (c = kw->current) < (n = kw->nknots)-1 ) {
198:     memmove ( &ad->keyparams[c*NKLARTPARAMS],
199:               &ad->keyparams[(c+1)*NKLARTPARAMS],
200:               (n-c-1)*NKLARTPARAMS*sizeof(float) );
201:     memmove ( &ad->qparams[c*4], &ad->qparams[(c+1)*4],
202:               (n-c-1)*4*sizeof(float) );
203:   }
204:    UpdateKeyInterpSpline ( ad );
205:    UpdateKeyInterpQSpline ( ad );
206: } /*DeleteKeyFrame*/
```

Początkowo są 4 węzły $(t_0, \ldots, t_3)$ dzielące na równe części przedział o długości 10 s. Dla wszystkich tych węzłów, w liniach 39–41, warunki interpolacyjne mają nadawane wartości wcześniej (przez procedurę konstruującą łańcuch kinematyczny) przepisane do tablicy `ad->wdg.artp` z tablicy `palmartp0` zadeklarowanej z procedurami obsługi łańcucha kinematycznego dłoni. Elementy tej tablicy są liczbami z przedziału $[0, 1]$, podobnie jak wartości nadawane przez suwaki w menu.

W liniach 42–44, dla wszystkich czterech węzłów, w tablicy `qparams` jest zapamiętywana jedynka kwaternionowa, która reprezentuje przekształcenie tożsamościowe (tj. obrót obserwatora do położenia początkowego). W liniach 45 i 46 są konstruowane obie krzywe interpolacyjne, które dla początkowych warunków interpolacyjnych będą funkcjami stałymi.

Procedura `CleanupKeyFrames` zwalnia pamięć zarezerwowaną przez procedurę `InitKeyFrames` na tablice i należy ją wywołać podczas sprzątania przed zatrzymaniem aplikacji.

Procedura `FindKeyFrame` (linie 62–88) jest wywoływana w odpowiedzi na pstryknięcie guzika ze strzałką `<-` lub `->`. Procedura ta znajduje pierwszy węzeł na lewo albo na prawo od liczby x pamiętanej w polu `xc` wihajstra osi czasu. Węzeł ten zostaje przypisany wihajstrowi jako bieżący, a następnie jest dokonywana artykulacja łańcucha kinematycznego, przy czym parametry artykulacji są brane z warunków interpolacyjnych dla bieżącego węzła. Również macierz przejścia do układu obserwatora jest (w linii 86) obliczana na podstawie kwaternionu będącego warunkiem interpolacyjnym obrotu obserwatora dla tego węzła.

Procedura `SetKeyFrame` (linie 90–107) jest wywoływana po pstryknięciu guzika `Set` w menu. Jej zadaniem jest przypisanie nowych warunków interpolacyjnych związanych z bieżącym węzłem — wektora parametrów artykulacji (w liniach 100–101) i kwaternionu określającego bieżące położenie obserwatora (w liniach 103–104). Obie krzywe interpolacyjne są niezwłocznie rekonstruowane.

Procedura `InsertKeyFrame` (linie 118–152) jest wywoływana po dodaniu nowego węzła za pomocą wihajstra osi czasu. Jeśli nowy węzeł nie jest większy niż dotychczasowy ostatni, to wykonywane są instrukcje w liniach 128–140. Dla nowego węzła, na podstawie dotychczasowych krzywych interpolacyjnych, jest obliczany wektor parametrów artykulacji, które następnie są ograniczane (przez procedurę `ClampArtParams`) do przedziału $[0,1]$ i (w liniach 134–135) wpisywane do tablicy warunków interpolacyjnych. W liniach 149–150 jest obliczany odpowiedni punkt na krzywej kwaternionowej, a jego współrzędne są zapamiętywane jako warunek interpolacyjny nowej krzywej, która zostanie skonstruowana dla wydłużonego ciągu węzłów.

Procedura `ChangeKeyFrame` jest wywoływana po przesunięciu węzła. Jeśli nastąpiła zmiana kolejności węzłów, to odpowiadające tym węzłom warunki interpolacyjne są przestawiane, czym zajmuje się procedura `SwapKeyFrames`. Przestawia ona odpowiednie wektory parametrów artykulacji łańcucha i kwaterniony.

Procedura `DeleteKeyFrame`, wywoływana po usunięciu węzła, dokonuje odpowiednich przemieszczeń warunków interpolacyjnych w tablicach, po czym konstruuje krzywe interpolacyjne na podstawie danych, które zostały po tej operacji.

Listing 36.8 przedstawia procedury, które realizują artykulację łańcucha i przemieszczanie obserwatora na potrzeby animacji. Drugim parametrem procedury `ArticulateKLAtX` jest czas, tj. argument krzywej sklejanej, której punkty są wektorami zmiennych artykulacji. Po obliczeniu punktu (przez procedurę `BSCdeBoorf`) jego współrzędne są obcinane do przedziału $[0,1]$, po czym procedura `ArticulatePalmLinkage` odpowiednio zgina stawy poszczególnych palców. Podobnie procedura `ArticulateVPosAtX` oblicza punkt krzywej kwaternionowej, zamienia go na obrót (tj. oblicza wektor osi i kąt obrotu) i wywołuje procedurę `SetupViewMatrix` w celu skonstruowania (i zapamiętania w pamięci GPU) macierzy przejścia do układu obserwatora.

Listing 36.8. Procedury pomocnicze animacji

C

```
1: void ArticulateKLAtX ( AppData *ad, float x )
2: {
3:   AppWidgets *aw;
4:
5:   aw = &ad->wdg;
6:   if ( x <= ad->keyknots[0] || !ad->bs_ok )
7:     memcpy ( aw->artp, ad->keyparams, NKLARTPARAMS*sizeof(float) );
8:   else if ( x >= ad->keyknots[ad->lastkeyframe] )
9:     memcpy ( aw->artp, &ad->keyparams[(ad->lastkeyframe)*NKLARTPARAMS],
10:             NKLARTPARAMS*sizeof(float) );
11:  else {
12:    BSCdeBoorf ( 3, ad->lastbsknot, ad->bsknots, NKLARTPARAMS,
13:                 ad->bsparams, x, aw->artp );
14:    ClampArtParams ( aw->artp );
15:  }
16:  ArticulatePalmLinkage ( ad );
17: } /*ArticulateKLAtX*/
```

```
18:
19: void ArticulateVPosAtX ( AppData *ad, float x )
20: {
21:   float      q[4], v[3];
22:   double     a;
23:
24:   if ( x <= ad->keyknots[0] || !ad->qs_ok )
25:     memcpy ( q, ad->qparams, 4*sizeof(float) );
26:   else if ( x >= ad->keyknots[ad->lastkeyframe] )
27:     memcpy ( q, &ad->qparams[4*ad->lastkeyframe], 4*sizeof(float) );
28:   else
29:     QuatSlerpdeBoorf ( 3, ad->lastbsknot, ad->bsknots,
30:                        ad->qbsparams, x, q );
31:   RotVQuatf ( v, &a, q );
32:   SetupViewMatrix ( ad, v, a );
33: } /*ArticulateVPosAtX*/
34:
35: void KnotWidgetPoint ( AppData *ad, int xi )
36: {
37:   AppWidgets *aw;
38:   float      x;
39:
40:   aw = &ad->wdg;
41:   x = KnotsWidgetXitoX ( &aw->kw, xi );
42:   ArticulateKLAtX ( ad, x );
43:   if ( aw->animate_vp )
44:     ArticulateVPosAtX ( ad, x );
45: } /*KnotWidgetPoint*/
```

Procedura `KnotWidgetPoint` jest wywoływana, gdy wihajster osi czasu jest w stanie przesuwania chwili (`XGESTATE_KW_MOVING_MOUSE`) i użytkownik wskazał nowy punkt na osi czasu. Dla tego punktu jest obliczany argument funkcji sklejanych i wywoływane są opisane wcześniej procedury artykulacji.

36.4. Menu trzeciego podokna

Nowe wihajstry w menu trzeciego podokna mają kolejne numery, opatrzone nazwami (zob. listing 36.9) widocznymi w obu częściach aplikacji, okienkowej i graficznej, w której są identyfikatorami obsługiwanych poleceń.

Listing 36.10 przedstawia procedury związane z menu trzeciego podokna. Procedura `SetupApp3dWin3Menu` (linie 87–117) tworzy menu i jego wihajstry. W liniach 96–98 powstaje wihajster osi czasu; wcześniej była wywołana procedura `InitKeyFrames`, która zarezerwowała m.in. tablicę węzłów, obecnie „przyczepianą” do tworzonego wihajstra. Guzik tworzony przez instrukcję w linii 104 służy do włączania i wyłączania animacji. W linii 105 jest mu przypisywana nowa procedura rysowania. Przełączniki tworzone w liniach 106–115 będą

Listing 36.9. Nazwy i numery nowych wihajstrów

C

```
#define KNOTWD_ID          29  /* SL_ID_ARTP0 + NKLARTPARAMS */
#define BTN_ID_PLAY        30
#define BTN_ID_LEFT        31
#define BTN_ID_SET         32
#define BTN_ID_RIGHT       33
#define SW_ID_EDIT_KNOTS   34
#define SW_ID_KNW_PANZOOM  35
#define SW_ID_ANIMATE_KL   36
#define SW_ID_ANIMATE_MM   37
#define SW_ID_ANIMATE_VP   38
```

nadawać wartości polom `editswitch` i `panswitch` określającym tryb działania wihajstra osi czasu, dlatego muszą być utworzone *po* utworzeniu tego wihajstra.

Tło i ramka guzika włączania i wyłączania animacji są rysowane tak samo jak dla zwykłego guzika, z kolei obrazek wyświetlany zamiast napisu na tym guziku wykonują instrukcje w liniach 69–84. Guzik jest w stanie podstawowym (`WDGSTATE_DEFAULT`), gdy animacja jest wyłączona, i w stanie innym niż podstawowy, gdy jest włączona; stan przypisuje temu guzikowi procedura `ToggleAnimation`.

Listing 36.10. Procedury tworzenia i obsługi menu

C

```
char str_SET[]     = "Set";
char str_LEFT[]    = "<-";
char str_RIGHT[]   = "->";
char str_PANZOOM[] = "pan/zoom";
char str_EDIT[]    = "edit";

void Win3Reshape ( xwinmenu *wm, short w, short h )
{
  myknotwdg->wdg.input ( &myknotwdg->wdg, WDGMSG_RECONFIGURE, 0,
                         w-MENU1_WIDTH, MENU3_HEIGHT-1 );
  PostExposeEvent ( wm->window, w, h );
} /*Win3Reshape*/

void Win3Callback ( struct xwidget *wdg, int msg, int key, int x, int y )
{
  switch ( msg ) {
case WDGMSG_RECONFIGURE:
    Win3Reshape ( wdg->wm, x, y );
    break;

case WDGMSG_BUTTON_PRESS:
    wm3->changed |= ProcessButtonCommand ( wdg->id );
    break;
```

```
24:
25: case WDGMSG_SWITCH_CHANGE:
26:     ProcessSwitchCommand ( wdg->id );
27:     break;
28:
29: case WDGMSG_KNOT_MCLICK:
30:     if ( key == 0 || key == 2 ) {
31:       ProcessKnotWidgetCommand ( wdg->id, msg, x );
32:       goto let_redraw_it;
33:     }
34:     break;
35:
36: case WDGMSG_KNOT_MMOVE:
37: case WDGMSG_KNOT_CHANGE:
38: case WDGMSG_KNOT_INSERT:
39: case WDGMSG_KNOT_DELETE:
40:     ProcessKnotWidgetCommand ( wdg->id, msg, x );
41: let_redraw_it:
42:     wm1->changed = wm2->changed = wm3->changed = true;
43:     PostMenuExposeEvent ( wm1 );
44:     PostMenuExposeEvent ( wm2 );
45:     break;
46:
47: case XWMSG_KEY_PRESS:
48:     mywdg->input ( mywdg, msg, key, x, y );
49:     if ( wm2->changed )
50:       PostMenuExposeEvent ( wm2 );
51:     break;
52:
53: default:
54:     break;
55:   }
56: } /*Win3Callback*/
57:
58: void MyButtonRedraw ( struct xwidget *wdg )
59: {
60:   XRectangle rect[2];
61:   XPoint tr[3];
62:
63:   XSetForeground ( xdisplay, xgc, XWP_BUTTON_COLOUR );
64:   XFillRectangle ( xdisplay, wdg->wm->pixmap, xgc,
65:                    wdg->r.x, wdg->r.y, wdg->r.width-1, wdg->r.height-1 );
66:   XSetForeground ( xdisplay, xgc, XWP_TEXT_COLOUR );
67:   XDrawRectangle ( xdisplay, wdg->wm->pixmap, xgc,
68:                    wdg->r.x, wdg->r.y, wdg->r.width-1, wdg->r.height-1 );
69:   if ( wdg->state == WDGSTATE_DEFAULT ) {
70:     tr[0].x = tr[1].x = wdg->r.x+wdg->r.width/2-4;
```

```
71:     tr[2].x = wdg->r.x+wdg->r.width/2+4;
72:     tr[0].y = wdg->r.y+4;  tr[1].y = wdg->r.y+wdg->r.height-4;
73:     tr[2].y = wdg->r.y+wdg->r.height/2;
74:     XFillPolygon ( xdisplay, wdg->wm->pixmap, xgc, tr, 3,
75:                    Convex, CoordModeOrigin );
76:   }
77:   else {
78:     rect[0].width = rect[1].width = 3;
79:     rect[0].height = rect[1].height = wdg->r.height-8;
80:     rect[0].y = rect[1].y = wdg->r.y+4;
81:     rect[0].x = wdg->r.x+wdg->r.width/2-4;
82:     rect[1].x = wdg->r.x+wdg->r.width/2+3;
83:     XFillRectangles ( xdisplay, wdg->wm->pixmap, xgc, rect, 2 );
84:   }
85: } /*MyButtonRedraw*/
86:
87: xwinmenu *SetupApp3dWin3Menu ( void )
88: {
89:   KnotsWidgetf *kw;
90:   xwinmenu     *wm;
91:
92:   if ( !(wm = NewWinMenu ( window[3], WIN0_WIDTH, MENU3_HEIGHT,
93:                  0, WIN0_HEIGHT-MENU3_HEIGHT, NULL, NULL, Win3Callback)) )
94:     ExitOnError ( "SetupApp3dWin3Menu" );
95:   kw = &appwdg->kw;
96:   myknotwdg = NewKnotsWidget ( wm, kw, KNOTWD_ID, WIN0_WIDTH-MENU1_WIDTH,
97:                MENU3_HEIGHT-4, MENU1_WIDTH, 4, 4, MAXKEYFRAMES, kw->nknots,
98:                kw->knots, kw->knots[0], kw->knots[kw->nknots-1] );
99:   kw->editswitch = true;
100:  kw->panswitch = kw->motion_off = false;
101:  NewButton ( wm, BTN_ID_LEFT, 16, 18, 0, 0, str_LEFT );
102:  NewButton ( wm, BTN_ID_SET, 22, 18, 19, 0, str_SET );
103:  NewButton ( wm, BTN_ID_RIGHT, 16, 18, 44, 0, str_RIGHT );
104:  playbtn = NewButton ( wm, BTN_ID_PLAY, 60, 18, 0, 20, NULL );
105:  playbtn->redraw = MyButtonRedraw;
106:  NewSwitch ( wm, SW_ID_ANIMATE_KL, 16, 16, 63, 20, NULL,
107:              &appwdg->animate_kl );
108:  NewSwitch ( wm, SW_ID_ANIMATE_MM, 16, 16, 82, 20, NULL,
109:             &appwdg->animate_mm );
110:  NewSwitch ( wm, SW_ID_ANIMATE_VP, 16, 16, 101, 20, NULL,
111:             &appwdg->animate_vp );
112:  NewSwitch ( wm, SW_ID_EDIT_KNOTS, 16, 16, 0, 40, str_EDIT,
113:              &kw->editswitch );
114:  NewSwitch ( wm, SW_ID_KNW_PANZOOM, 16, 16, 44, 40, str_PANZOOM,
115:              &kw->panswitch );
116:  return wm;
117: } /*SetupApp3dWin3Menu*/
```

Wihajstry w menu trzeciego podokna wywołują procedurę `Win3Callback` (linie 14–56), której działanie chyba nie wymaga wyjaśnień.

36.5. Część graficzna aplikacji

Procedury pokazane na listingach 36.11–36.14 należą do części graficznej i zajmują się wykonywaniem poleceń otrzymywanych od części okienkowej, w tym edycją warunków interpolacyjnych i realizacją animacji. Włączaniem i wyłączaniem animacji zajmuje się procedura `ToggleAnimation` (listing 32.17), wywoływana po naciśnięciu klawisza spacji i po pstryknięciu odpowiedniego guzika w menu. Wywoływana przez nią procedura `ProcessWorldRequest` w części okienkowej zajmuje się zapewnieniem, że będzie wywoływana procedura `MoveOn`, ale też zmianą wyglądu guzika włączającego i wyłączającego animację.

Procedura `ProcessSwitchCommand` w dodatku do przełączników wybierających rysowane siatki obsługuje przełączniki wybierające tryb działania wihajstra osi czasu (zależnie od niego, po naciśnięciu przycisku wihajster umożliwia przesuwanie węzłów albo zmienianie przedziału czasowego) oraz przełączniki wyboru elementów animacji. Z trzech elementów co najmniej jeden powinien być włączony, aby aplikacja reagowała na polecenie uruchamiania animacji, dlatego wyłączenie ostatniego przełącznika powoduje włączenie jednego z pozostałych dwóch. Ale, takie reakcje aplikacji mogą być dla użytkownika niezrozumiałe, więc zawsze trzeba się zastanowić, czy warto je programować.

Listing 36.11. Procedura `ProcessSwitchCommand`

C

```
1: char ProcessSwitchCommand ( int wdg_id )
2: {
3:   switch ( wdg_id ) {
4: case SW_ID_MESH0:
5:     return true;
6: case SW_ID_MESH1:  case SW_ID_MESH2:  case SW_ID_MESH3:  case SW_ID_MESH4:
7:     if ( appdata.wdg.sw[wdg_id-SW_ID_MESH0] ) {
8:       memset ( &appdata.wdg.sw[1], false, NPALMMESHES );
9:       appdata.wdg.sw[(int)(appdata.lod = wdg_id-SW_ID_MESH0)] = true;
10:      kl_Articulate ( appdata.linkage );
11:    }
12:    else
13:      appdata.lod = -1;
14:    return true;
15: case SW_ID_ANIMATE_KL:
16:    if ( !appdata.wdg.animate_kl && !appdata.wdg.animate_vp )
17:      appdata.wdg.animate_mm = true;
18:    return true;
19: case SW_ID_ANIMATE_MM:
20:    if ( !appdata.wdg.animate_mm && !appdata.wdg.animate_vp )
21:      appdata.wdg.animate_kl = true;
```

```
22:     return true;
23: case SW_ID_ANIMATE_VP:
24:     if ( !appdata.wdg.animate_vp && !appdata.wdg.animate_kl )
25:       appdata.wdg.animate_mm = true;
26:     return true;
27: case SW_ID_EDIT_KNOTS:
28:     if ( appdata.wdg.kw.editswitch )
29:       appdata.wdg.kw.panswitch = false;
30:     return true;
31: case SW_ID_KNW_PANZOOM:
32:     if ( appdata.wdg.kw.panswitch )
33:       appdata.wdg.kw.editswitch = false;
34:     return true;
35: default:
36:     return false;
37:   }
38: } /*ProcessSwitchCommand*/
```

Procedura `ProcessSlidebarCommand`, wywoływana po przesunięciu suwaka, jest identyczna jak w aplikacji 3B (listing 34.6). Listing 36.12 przedstawia procedurę wywoływaną po pstryknięciu guzika — zależnie od jego identyfikatora (czyli identyfikatora polecenia wydawanego przez ten guzik) procedura włącza lub wyłącza animację, odnajduje węzeł, który ma stać się bieżącym dla wihajstra osi czasu albo zapamiętuje położenia suwaków wyznaczające parametry artykulacji dla bieżącego węzła.

Listing 36.12. Procedura `ProcessButtonCommand`

C

```
1: char ProcessButtonCommand ( int wdg_id )
2: {
3:   switch ( wdg_id ) {
4: case BTN_ID_PLAY:
5:     ToggleAnimation ( &appdata );
6:     return appdata.wdg.animation;
7: case BTN_ID_LEFT:
8:     if ( appdata.wdg.animation )
9:       ToggleAnimation ( &appdata );
10:    FindKeyFrame ( &appdata, false );
11:    return true;
12: case BTN_ID_RIGHT:
13:    if ( appdata.wdg.animation )
14:      ToggleAnimation ( &appdata );
15:    FindKeyFrame ( &appdata, true );
16:    return true;
17: case BTN_ID_SET:
18:    SetKeyFrame ( &appdata );
19:    return false;
20: default:
```

```
21:     return false;
22:   }
23: } /*ProcessButtonCommand*/
```

Procedura ProcessCharCommand, wykonująca polecenia wydawane za pomocą klawiatury, gdy kursor jest w obszarze obrazu, jest identyczna jak w aplikacji 3C. Jej listing pomijam, natomiast na listingu 36.13 przedstawiam procedurę wykonującą polecenia wydawane przez wihajster osi czasu. Jej zadaniem jest wywołanie odpowiedniej procedury z listingu 36.7.

Listing 36.13. Procedura ProcessKnotWidgetCommand

C

```
1: char ProcessKnotWidgetCommand ( int wdg_id, int msg, int x )
2: {
3:   switch ( msg ) {
4: case WDGMSG_KNOT_MCLICK:
5:     KnotWidgetPoint ( &appdata, x );
6:     return true;
7: case WDGMSG_KNOT_MMOVE:
8:     if ( !appdata.wdg.animation )
9:       KnotWidgetPoint ( &appdata, x );
10:    return true;
11: case WDGMSG_KNOT_CHANGE:
12:    ChangeKeyFrame ( &appdata );
13:    return true;
14: case WDGMSG_KNOT_INSERT:
15:    InsertKeyFrame ( &appdata );
16:    return true;
17: case WDGMSG_KNOT_DELETE:
18:    DeleteKeyFrame ( &appdata );
19:    return true;
20: default:
21:    return false;
22:   }
23: } /*ProcessKnotWidgetCommand*/
```

Procedura MoveOn (listing 36.14) „posuwa do przodu” wybrane elementy animacji: w liniach 8–10 oblicza nową macierz przekształcenia modelu wykonującego ruch obrotowy ze stałą prędkością. W liniach 12–14 procedura dodaje do czasu pamiętanego w polu xc wihajstra osi czasu przyrost zmierzony przez stoper i jeśli ten czas przekracza ostatni węzeł, to „wraca na początek”, tj. do pierwszego węzła, dzięki czemu ruch jest okresowy. Zależnie od stanu przełączników procedura dokonuje artykulacji łańcucha kinematycznego i oblicza nowe położenie obserwatora w ruchu obrotowym na podstawie krzywej kwaternionowej.

Listing 36.14. Procedura `MoveOn`

C

```
1: char MoveOn ( void )
2: {
3:   double dt;
4:
5:   if ( appdata.wdg.animation ) {
6:     dt = TimerTocTic ();
7:     if ( appdata.wdg.animate_mm ) {
8:       if ( (appdata.model_rot_angle += appdata.speed * dt) >= PI )
9:         appdata.model_rot_angle -= 2.0*PI;
10:      SetupModelMatrix ( &appdata );
11:    }
12:    appdata.wdg.kw.xc += dt;
13:    if ( appdata.wdg.kw.xc > appdata.keyknots[appdata.wdg.kw.nknots-1] )
14:      appdata.wdg.kw.xc = appdata.keyknots[0];
15:    if ( appdata.wdg.animate_kl )
16:      ArticulateKLAtX ( &appdata, appdata.wdg.kw.xc );
17:    if ( appdata.wdg.animate_vp )
18:      ArticulateVPosAtX ( &appdata, appdata.wdg.kw.xc );
19:  }
20:  return appdata.wdg.animation;
21: } /*MoveOn*/
```

Na listingu 36.15 jest pokazana zmiana w procedurze inicjalizacji danych części graficznej aplikacji; dodane zostało wywołanie procedury `InitKeyFrames` po skonstruowaniu łańcucha kinematycznego. Menu i wihajstry w oknach są tworzone po powrocie z tej procedury, która polu `appdata.wdg.kw.knots` przypisała adres tablicy węzłów (listing 36.7, linia 38) — w ten sposób tablica ta jest udostępniana części okienkowej, co umożliwia utworzenie wihajstra osi czasu, który będzie manipulował węzłami w tej tablicy.

Listing 36.15. Zmiany w procedurze `InitMyWorld`

C

```
1: AppWidgets *InitMyWorld ( int argc, char *argv[], int width, int height )
2: {
3:   .... /* instrukcje z linii 3-22, listing 34.5 */
4:   if ( ConstructPalmLinkage ( &appdata ) ) {
5:     ArticulatePalmLinkage ( &appdata );
6:     if ( InitKeyFrames ( &appdata ) )
7:       return &appdata.wdg;
8:   }
9:   ExitOnError ( "InitMyWorld" );
10:  return NULL;
11: } /*InitMyWorld*/
```

36.6. Pozostałe zmiany w aplikacji

Listing 36.16 przedstawia procedurę wywoływaną po otrzymaniu przez okno główne komunikatu o zmianie wymiarów (wywołanie tej procedury zastępuje instrukcje w liniach 7–10 na listingu 32.24); obszar okna głównego jest teraz podzielony między trzy podokna, których wymiary trzeba obliczyć i wysłać do nich komunikat o zmianie wymiarów. Procedury obsługi tych komunikatów w oknach zajmą się odpowiednim nadawaniem wymiarów i rozmieszczaniem wihajstrów.

Listing 36.16. Procedura `Win0ConfigureNotify`

C

```
1: #define WIN0_WIDTH   560  /* wymiary początkowe okna głównego */
2: #define WIN0_HEIGHT  420
3: #define MENU1_WIDTH  120
4: #define MENU3_HEIGHT  60
5:
6: Window window[4];
7:
8: void Win0ConfigureNotify ( int width, int height )
9: {
10:   window0_width  = width;
11:   window0_height = height;
12:   XMoveResizeWindow ( xdisplay, window[1], 0, 0,
13:                       MENU1_WIDTH, height-MENU3_HEIGHT );
14:   XMoveResizeWindow ( xdisplay, window[2], MENU1_WIDTH, 0,
15:                       width-MENU1_WIDTH, height-MENU3_HEIGHT );
16:   XMoveResizeWindow ( xdisplay, window[3], 0, height-MENU3_HEIGHT,
17:                       width, MENU3_HEIGHT );
18: } /*Win0ConfigureNotify*/
```

Pozostałe zmiany to dodanie instrukcji, które na początku działania aplikacji tworzą trzecie podokno i wywołują opisaną wcześniej procedurę `SetupApp3dWin3Menu`, dodanie do procedury `MessageLoop` przesyłania komunikatów do tego podokna oraz dodanie instrukcji wykonujących dodatkowe sprzątanie podczas zatrzymania aplikacji.

36.7. *Uzupełnienia — użycie macierzy zagęszczania siatek

Zastąpienie procedury zagęszczania, która wykonuje pełne obliczenie niezmieniającej się topologii siatki i współrzędnych wierzchołków, przez procedurę obliczającą tylko współrzędne przy użyciu macierzy zagęszczania (zobacz podrozdz. 31.11) wymaga niewielu zmian w kodzie aplikacji. Do struktury typu `KLMesh` trzeba dodać opakowania macierzy zagęszczania — tablicę o długości `NPALMMESHES` struktur typu `MeshRefineMatrix` pokazanych na listingu 31.36 (tablica ta otrzymała nazwę `refm`). Zamiast procedur z listingu 31.10, kompilujących i sprzątających pełny program zagęszczania siatek, trzeba wywołać procedury

`LoadMeshRefinementMatrixProgram` i `DeleteMeshRefinementMatrixProgram`, kompilujące i sprzątające program znajdujący macierze zagęszczania. Dodatkowo trzeba dołączyć do aplikacji i wywołać procedury, które na początku przygotowują do pracy, a na końcu likwidują programy mnożenia na GPU macierzy rzadkich i macierzy rzadkiej przez wektor. Programy te zawierają szadery opisane w p. G.4.3 i G.4.1.

Listing 36.17 przedstawia zmienione instrukcje metod obiektu dłoni. Konstruktor, tj. procedura `KLInitPalmMesh`, w pętli wywołuje przedstawioną na listingu 31.41 procedurę `GPUmeshRefinementMatrix`, która znajduje topologie kolejnych zagęszczonych siatek i odpowiednie macierze zagęszczania. Dodatkowym zadaniem destruktora obiektu, tj. procedury `KLDeletePalmMesh`, jest zwolnienie pamięci GPU zajmowanej przez te macierze. Procedura `KLPostprocessMesh` dokonuje artykulacji siatki niezagęszczonej tak samo jak dotąd. Następnie *zamiast* procedury `GPUmeshRefinement` wywołuje szybszą procedurę `GPUMatrixRefineMesh`, tyle razy, ilu kolejnych zagęszczań wynik ma być narysowany. Przed wywołaniem tej procedury, w linii 31, zmiennej opisującej liczbę skalarnych atrybutów wierzchołka siatki jest przypisywana wartość 3, bo tyle ich mają wierzchołki siatki oryginalnej i tyle samo będą początkowo mieć wierzchołki siatek zagęszczonych (ponieważ liczba atrybutów w tym miejscu może tylko się zmniejszyć, nie trzeba rezerwować nowego bufora na współrzędne wierzchołków). Ta, która ma być narysowana, zostaje potem przetworzona przez procedurę `ComputeMeshNormalVectors`, która rozszerza zestaw atrybutów o współrzędne wektora normalnego.

Listing 36.17. Zmiany w konstruktorze, destruktorze i metodzie postprocesingu obiektu dłoni

C

```
1: static char KLInitPalmMesh ( kl_linkage *lkg, kl_object *obj )
2: {
3:   .... /* linie 3-45 z listingu 34.3 bez zmian */
4:     for ( i = 1; i <= NPALMMESHES; i++ ) {
5:       if ( !GPUmeshRefinementMatrix ( 3, palms[i], palms[i+1],
6:                                       &md->refm[i-1] ) )
7:         ExitOnError ( "KLInitPalmMesh 1" );
8:     }
9:    md->mtn = SetupMaterial ( &ad->mat, -1, diffr, specr, shn, wa, we );
10:  .... /* dalsze instrukcje bez zmian */
11: } /*KLInitPalmMesh*/
12:
13: static void KLDeletePalmMesh ( kl_linkage *lkg, kl_object *obj )
14: {
15:   KLMesh  *md;
16:   int     i;
17:
18:   md = (KLMesh*)obj->usrdata;
19:   for ( i = 0; i <= NPALMMESHES+1; i++ )
20:     DeleteGPUmesh ( md->mesh[i] );
21:   for ( i = 0; i < NPALMMESHES; i++ )
22:     GPUDeleteMeshRefinementMatrix ( &md->refm[i] );
```

```
23:   glDeleteBuffers ( 1, &md->tribuf );
24: } /*KLDeletePalmMesh*/
25:
26: static void KLPostprocessMesh ( kl_linkage *lkg, kl_object *obj )
27: {
28:   .... /* linie 79-98 na listingu 34.3 bez zmian */
29:   if ( ad->lod >= 1 ) {
30:     for ( i = 1; i <= ad->lod; i++ ) {
31:       mesh[i+1]->nsattr = mesh[i+1]->pdim = 3;
32:       GPUMatrixRefineMesh ( &md->refm[i-1] );
33:     }
34:     ComputeMeshNormalVectors ( mesh[ad->lod+1], 6, 3 );
35:   }
36:   ExitIfGLError ( "KLPostprocessMesh" );
37: } /*KLPostprocessMesh*/
```

36.8. *Ćwiczenia

W tym miejscu kurs, choć nie taki krótki, zakończył się i ufam, że Czytelnik, który go przeszedł, jest dobrze przygotowany do prawdziwej nauki programowania grafiki. Polega ona na *samodzielnym* formułowaniu problemów, studiowaniu źródeł, wymyślaniu ćwiczeń i rozwiązywaniu ich. Wszystkim Czytelnikom życzę udanych projektów, podziwu innych osób i końcowej satysfakcji.

A

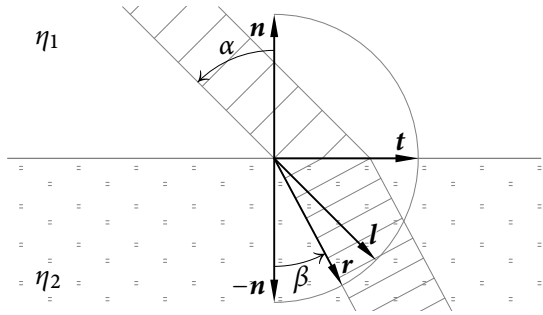
Jeszcze trochę algebry z geometrią

Idę na hamak — z nienawistną
Książką, bodaj ją dunder świsnął,
Mistyczną, apokaliptyczną,
Z abrakadabrą liter greckich,
Szatańskich szyfrów, cięć zdradzieckich:
Z kabałą trygonometryczną.

Julian Tuwim: *Kwiaty polskie*

A.1. Załamanie światła

Wyprowadzimy wzór (9.1), na podstawie którego dostępna w GLSL-u funkcja `refract` oblicza wektor kierunku załamania światła na granicy przezroczystych ośrodków. **Współczynnik załamania światła** jest ilorazem prędkości światła w próżni i w danym ośrodku — jest to zawsze liczba większa lub równa 1.



Rysunek A.1. Geometria załamania światła

Przyjmiemy oznaczenia z rysunku A.1; symbol $\boldsymbol{n}$ oznacza jednostkowy wektor normalny powierzchni rozgraniczającej ośrodki (np. powietrze i szkło lub wodę), a $\boldsymbol{l}$ oznacza jednost-

kowy wektor kierunku padania światła na tę powierzchnię[1]. Symbolem $\boldsymbol{r}$ oznaczymy wektor kierunku, w jakim światło porusza się po przejściu granicy ośrodków, a literami α i β odpowiednio kąty między wektorami $\boldsymbol{l}$ i $\boldsymbol{r}$ a wektorem $-\boldsymbol{n}$.

Literą η oznaczymy iloraz współczynników załamania światła ośrodka, z którego światło pada na granicę ośrodków i ośrodka po drugiej stronie (jeśli zatem światło wpada do ośrodka gęstszego, to $\eta < 1$). Znane ze szkoły **prawo załamania światła** głosi, że

$$\frac{\sin\beta}{\sin\alpha} = \frac{\eta_1}{\eta_2} = \eta.$$

Możemy na tej podstawie obliczyć

$$\sin\beta = \eta\sin\alpha \quad \text{oraz} \quad \cos\beta = \sqrt{1-\sin^2\beta} = \sqrt{1-\eta^2\sin^2\alpha} = \sqrt{1-\eta^2(1-\cos^2\alpha)}.$$

Mamy też $\cos\alpha = \langle -\boldsymbol{n}, \boldsymbol{l}\rangle$. Niech

$$\boldsymbol{t} = \frac{1}{\sin\alpha}(\boldsymbol{l} - \langle \boldsymbol{n}, \boldsymbol{l}\rangle \boldsymbol{n});$$

jest to jednostkowy wektor styczny do granicy ośrodków, który umożliwia obliczenie wektora jednostkowego

$$\begin{aligned}\boldsymbol{r} &= -\boldsymbol{n}\cos\beta + \boldsymbol{t}\sin\beta = -\boldsymbol{n}\cos\beta + \boldsymbol{t}\eta\sin\alpha = -\boldsymbol{n}\sqrt{1-\eta^2(1-\langle \boldsymbol{n}, \boldsymbol{l}\rangle^2)} + \eta(\boldsymbol{l} - \langle \boldsymbol{n}, \boldsymbol{l}\rangle \boldsymbol{n}) \\ &= \eta\boldsymbol{l} - \Big(\sqrt{1-\eta^2(1-\langle \boldsymbol{n}, \boldsymbol{l}\rangle^2)} + \eta\langle \boldsymbol{n}, \boldsymbol{l}\rangle\Big)\boldsymbol{n}.\end{aligned}$$

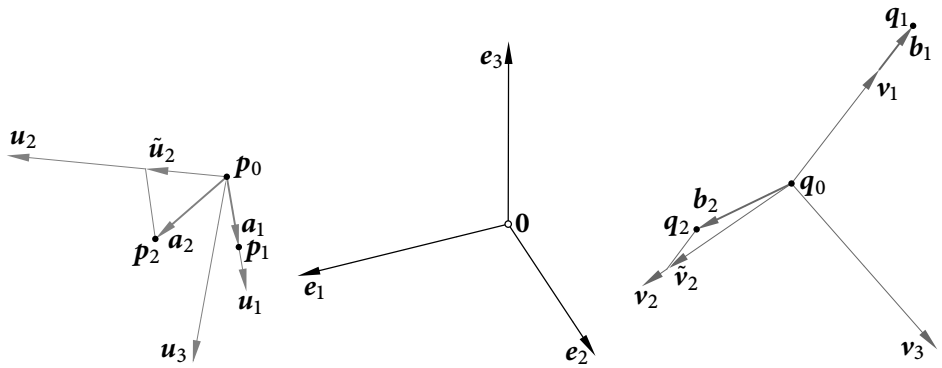
Symbol k we wzorze (9.1) oznacza $\cos^2\beta$ — jest to wyrażenie pod pierwiastkiem w wyprowadzeniu powyżej. Jeśli światło pada na granicę z wnętrza ośrodka gęstszego (o większym współczynniku załamania światła), to może zdarzyć się, że $k < 0$. W takim przypadku światło nie przechodzi przez granicę ośrodków — następuje tzw. **całkowite odbicie wewnętrzne** światła. Wartością funkcji `refract` jest wtedy wektor zerowy.

Należy pamiętać, że współczynniki załamania światła (i ich ilorazy) zależą od długości fali świetlnej i podczas tworzenia obrazów o bardzo wysokiej jakości scen, w których występuje pryzmat, może być potrzebne wykonanie osobnego obrazu dla kilku (więcej niż trzech) długości fali, a potem połączenie tych obrazów w jeden.

A.2. Konstrukcje obrotów do ustalonego położenia

Rozwiążemy następujące zadanie: mając dane dwie trójki punktów, $\boldsymbol{p}_0$, $\boldsymbol{p}_1$ i $\boldsymbol{p}_2$ oraz $\boldsymbol{q}_0$, $\boldsymbol{q}_1$ i $\boldsymbol{q}_2$, z których żadna nie leży na jednej prostej, należy skonstruować przekształcenie afiniczne — obrót z przesunięciem — które punkt $\boldsymbol{p}_0$ przekształci na $\boldsymbol{q}_0$, półprostą $\boldsymbol{p}_0\boldsymbol{p}_1$ (o początku $\boldsymbol{p}_0$, przechodzącą przez $\boldsymbol{p}_1$) na półprostą $\boldsymbol{q}_0\boldsymbol{q}_1$ i wreszcie zawierającą punkt $\boldsymbol{p}_2$ półpłaszczyznę,

[1]Tu wektor $\boldsymbol{l}$ jest zorientowany przeciwnie do wektorów obliczanych przez szadery używane przez opisane w książce aplikacje i do wektorów rozważanych w podrozdziale 28.1.



Rysunek A.2. Konstrukcja obrotu z przesunięciem do zadanego położenia

której brzegiem jest prosta $\boldsymbol{p}_0\boldsymbol{p}_1$, na zawierającą punkt $\boldsymbol{q}_2$ półpłaszczyznę, której brzegiem jest prosta $\boldsymbol{q}_0\boldsymbol{q}_1$. Wszystkie punkty reprezentujemy za pomocą współrzędnych kartezjańskich w ustalonym układzie (np. świata). Rozwiązanie tego zadania może się przydać, gdy trzeba rozmieścić obiekty w zadany sposób, na przykład ustawić samochód na nierównym terenie.

Zaczniemy od skonstruowania macierzy R opisującej część liniową tego przekształcenia; jest to macierz poszukiwanego obrotu. Niech

$$\boldsymbol{a}_1 = \boldsymbol{p}_1 - \boldsymbol{p}_0, \quad \boldsymbol{a}_2 = \boldsymbol{p}_2 - \boldsymbol{p}_0 \quad \text{oraz} \quad \boldsymbol{b}_1 = \boldsymbol{q}_1 - \boldsymbol{q}_0, \quad \boldsymbol{b}_2 = \boldsymbol{q}_2 - \boldsymbol{q}_0.$$

Na podstawie tych wektorów skonstruujemy kolejne (rys. A.2):

$$\boldsymbol{u}_1 = \frac{1}{\|\boldsymbol{a}_1\|}\boldsymbol{a}_1, \quad \tilde{\boldsymbol{u}}_2 = \boldsymbol{a}_2 - \boldsymbol{u}_1\langle \boldsymbol{u}_1, \boldsymbol{a}_2\rangle, \quad \boldsymbol{u}_2 = \frac{1}{\|\tilde{\boldsymbol{u}}_2\|}\tilde{\boldsymbol{u}}_2, \quad \boldsymbol{u}_3 = \boldsymbol{u}_1 \wedge \boldsymbol{u}_2,$$
$$\boldsymbol{v}_1 = \frac{1}{\|\boldsymbol{b}_1\|}\boldsymbol{b}_1, \quad \tilde{\boldsymbol{v}}_2 = \boldsymbol{b}_2 - \boldsymbol{v}_1\langle \boldsymbol{v}_1, \boldsymbol{b}_2\rangle, \quad \boldsymbol{v}_2 = \frac{1}{\|\tilde{\boldsymbol{v}}_2\|}\tilde{\boldsymbol{v}}_2, \quad \boldsymbol{v}_3 = \boldsymbol{v}_1 \wedge \boldsymbol{v}_2.$$

Macierze $U = [\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3]$ i $V = [\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3]$ są ortogonalne i reprezentują pewne obroty w przestrzeni $\mathbb{R}^3$; obrót reprezentowany przez macierz U przeprowadza wektory $\boldsymbol{e}_1 = (1,0,0)$, $\boldsymbol{e}_2 = (0,1,0)$ i $\boldsymbol{e}_3 = (0,0,1)$ na wektory $\boldsymbol{u}_1$, $\boldsymbol{u}_2$ i $\boldsymbol{u}_3$, a zatem obrót reprezentowany przez macierz $U^{-1} = U^T$ przeprowadza wektory $\boldsymbol{u}_1$, $\boldsymbol{u}_2$ i $\boldsymbol{u}_3$ na $\boldsymbol{e}_1$, $\boldsymbol{e}_2$ i $\boldsymbol{e}_3$. Z kolei macierz V reprezentuje obrót przeprowadzający wektory $\boldsymbol{e}_1$, $\boldsymbol{e}_2$ i $\boldsymbol{e}_3$ na $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ i $\boldsymbol{v}_3$. Obrót przeprowadzający wektory $\boldsymbol{u}_1$, $\boldsymbol{u}_2$ i $\boldsymbol{u}_3$ na $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ i $\boldsymbol{v}_3$ jest więc reprezentowany przez macierz $R = VU^T$. Znalezienie wektora przesunięcia $\boldsymbol{t}$ konstruowanego przekształcenia afinicznego jest już łatwe: ma być $R\boldsymbol{p}_0 + \boldsymbol{t} = \boldsymbol{q}_0$, a zatem $\boldsymbol{t} = \boldsymbol{q}_0 - R\boldsymbol{p}_0$.

Procedura `M4x4RotationFromPointsf` na listingu A.1 wykonuje według powyższych wzorów obliczenie, którego wynikiem jest macierz 4×4 będąca jednorodną reprezentacją poszukiwanego przekształcenia. Pomocnicza procedura `M4x4AuxOrtf` na podstawie danych trójek punktów konstruuje macierze U i V. W linii 23 jest znajdowana transpozycja macierzy U; w tym celu wystarczy przestawić trzy pary współczynników. W liniach 26–28 obliczane

są kolumny macierzy R — iloczyny macierzy V i kolejnych kolumn macierzy U^T. Instrukcje w liniach 29–30 obliczają wektor przesunięcia $\boldsymbol{t}$.

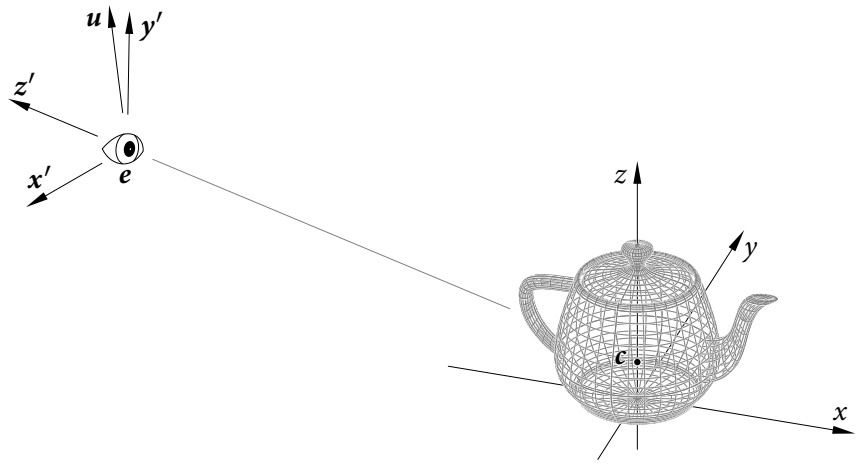
Listing A.1. Procedura wyznaczania macierzy obrotu z przesunięciem

C

```
1: void M4x4AuxOrtf ( GLfloat u[16], float p0[3], float p1[3], float p2[3] )
2: {
3:   float sp;
4:
5:   memset ( u, 0, 16*sizeof(GLfloat) );
6:   u[0] = p1[0]-p0[0];  u[1] = p1[1]-p0[1];  u[2] = p1[2]-p0[2];
7:   V3Normalisef ( &u[0] );
8:   u[4] = p2[0]-p0[0];  u[5] = p2[1]-p0[1];  u[6] = p2[2]-p0[2];
9:   sp = V3DotProductf ( &u[0], &u[4] );
10:  u[4] -= sp*u[0];     u[5] -= sp*u[1];     u[6] -= sp*u[2];
11:  V3Normalisef ( &u[4] );
12:  V3CrossProductf ( &u[8], &u[0], &u[4] );
13: } /*M4x4AuxOrtf*/
14:
15: void M4x4RotationFromPointsf ( GLfloat a[16],
16:                               float p0[3], float p1[3], float p2[3],
17:                               float q0[3], float q1[3], float q2[3] )
18: {
19: #define SWAP(x,y) { s = x;  x = y;  y = s; }
20:   GLfloat u[16], v[16], s;
21:
22:   M4x4AuxOrtf ( u, p0, p1, p2 );
23:   SWAP ( u[1], u[4] )  SWAP ( u[2], u[8] )  SWAP ( u[6], u[9] )
24:   M4x4AuxOrtf ( v, q0, q1, q2 );
25:   memset ( a, 0, 16*sizeof(GLfloat) );
26:   M4x4MultMV3f ( &a[0], v, &u[0] );
27:   M4x4MultMV3f ( &a[4], v, &u[4] );
28:   M4x4MultMV3f ( &a[8], v, &u[8] );
29:   M4x4MultMV3f ( &u[0], a, p0 );
30:   a[12] = q0[0]-u[0];  a[13] = q0[1]-u[1];  a[14] = q0[2]-u[2];
31:   a[15] = 1.0;
32: #undef SWAP
33: } /*M4x4RotationFromPointsf*/
```

Zobaczmy jeszcze jedną konstrukcję obrotu z przesunięciem, która może się przydać w aplikacjach OpenGL-a: przekształcenie to określa przejście od układu współrzędnych świata do układu umieszczonego w punkcie $\boldsymbol{e}$ obserwatora, który „patrzy” na punkt $\boldsymbol{c}$ znajdujący się na ujemnej półosi z (układu obserwatora, rys. A.3). Do jednoznacznego określenia tego układu potrzebny jest jeszcze jeden punkt lub wektor; podamy wektor $\boldsymbol{u}$ określający kierunek „do góry”. Wektory $\boldsymbol{e} - \boldsymbol{c}$ i $\boldsymbol{u}$ muszą mieć różne kierunki. Konstrukcja[2] zaczyna się od

[2] W bibliotece GLU jest procedura `gluLookAt` realizująca równoważną konstrukcję, ale ta procedura jest przeznaczona dla aplikacji starego OpenGL-a.



Rysunek A.3. Określenie układu obserwatora patrzącego na dany punkt

znalezienia wersorów osi układu obserwatora: wektor $\boldsymbol{z}'$ (wersor osi z) jest unormowaną (tj. podzieloną przez długość) różnicą $\boldsymbol{e} - \boldsymbol{c}$. Wersor $\boldsymbol{y}'$ osi y otrzymamy, normując rzut wektora $\boldsymbol{u}$ na płaszczyznę prostopadłą do wektora $\boldsymbol{z}'$, czyli biorąc $\boldsymbol{u}' = \boldsymbol{u} - \langle \boldsymbol{z}', \boldsymbol{u} \rangle \boldsymbol{z}'$, $\boldsymbol{y}' = \boldsymbol{u}'/\|\boldsymbol{u}'\|$, a wersor osi x jest równy $\boldsymbol{x}' = \boldsymbol{y}' \wedge \boldsymbol{z}'$. Macierz ortogonalna $[\boldsymbol{x}', \boldsymbol{y}', \boldsymbol{z}']$ opisuje część liniową przejścia od układu obserwatora do układu świata, zatem do przejścia w drugą stronę jest potrzebna transpozycja tej macierzy. Wektor przesunięcia trzeba dobrać tak, aby punkt $\boldsymbol{e}$ był początkiem układu obserwatora. Procedura realizująca tę konstrukcję jest pokazana na listingu A.2.

Listing A.2. Procedura `M4x4LookAtf`

C

```
1: void M4x4LookAtf ( GLfloat a[16], float eye[3], float c[3], float up[3] )
2: {
3:   float x[3], y[3], z[3], d;
4:   int   i;
5:
6:   V3Subtractf ( z, eye, c );
7:   V3Normalisef ( z );
8:   d = V3DotProductf ( z, up );
9:   for ( i = 0; i < 3; i++ )
10:    y[i] = up[i] - d*z[i];
11:  V3Normalisef ( y );
12:  V3CrossProductf ( x, y, z );
13:  a[0] = x[0];  a[1] = y[0];  a[2] = z[0];   a[3] = 0.0;
14:  a[4] = x[1];  a[5] = y[1];  a[6] = z[1];   a[7] = 0.0;
15:  a[8] = x[2];  a[9] = y[2];  a[10] = z[2];  a[11] = 0.0;
16:  a[12] = -V3DotProductf ( x, eye );
17:  a[13] = -V3DotProductf ( y, eye );
```

```
18:   a[14] = -V3DotProductf ( z, eye );
19:   a[15] = 1.0;
20: } /*M4x4LookAtf*/
```

A.3. Rozkładanie przekształceń afinicznych

W wielu książkach można przeczytać, że dowolne przekształcenie afiniczne przestrzeni trójwymiarowej[3] jest złożeniem opisanych w rozdziale 5 przekształceń elementarnych: przesunięć, obrotów i skalowań. Znacznie rzadziej można znaleźć informację, *jak* znaleźć takie przekształcenia elementarne, aby ich złożenie było przekształceniem danym, czyli jak znaleźć macierze przesunięcia, obrotów i skalowania, których iloczyn jest daną macierzą 4×4, będącą jednorodną reprezentacją tego przekształcenia. Znalezienie odpowiedniego przesunięcia jest łatwe, skupimy się zatem na przedstawieniu opisującej część liniową przekształcenia macierzy A jako iloczynu macierzy obrotów i skalowania.

Wartość własna macierzy A jest to liczba λ spełniająca równość $A\boldsymbol{x} = \lambda\boldsymbol{x}$ razem z pewnym niezerowym wektorem $\boldsymbol{x}$, zwanym **wektorem własnym**. Pomnożenie wektora własnego $\boldsymbol{x}$ przez macierz A daje zatem ten sam wynik, co pomnożenie go przez liczbę λ.

Macierz $n \times n$ ma n wartości własnych, które mogą się nakładać[4]. Wartości własne mogą być liczbami rzeczywistymi lub zespolonymi, ale jeśli macierz jest rzeczywista, to jej zespolone wartości własne występują w parach sprzężonych, a związane z nimi wektory własne mają przynajmniej niektóre współrzędne zespolone (a więc w przestrzeni $\mathbb{R}^n$ ich nie znajdziemy). Iloczyn wszystkich wartości własnych macierzy jest jej wyznacznikiem.

Jeśli macierz A jest **symetryczna**, tj. $A = A^T$, to istnieje ortogonalna macierz X i diagonalna macierz Λ, takie że $A = X\Lambda X^{-1}$; współczynniki na diagonali macierzy Λ są wartościami własnymi macierzy A. Kolumny macierzy X są jednostkowymi wektorami własnymi macierzy A, przy czym możemy przyjąć, że wyznacznik macierzy X jest dodatni (równy +1). Dla $n = 3$ taka macierz X reprezentuje pewien obrót przestrzeni $\mathbb{R}^3$ (o czym niżej) i jednocześnie reprezentuje zmianę układu współrzędnych. Przekształcenie reprezentowane przez symetryczną macierz A jest zatem złożeniem trzech przekształceń: reprezentowanego przez macierz X^{-1} obrotu, skalowania osi x, y i z (wartości własne macierzy A są współczynnikami tego skalowania) i przekształcenia odwrotnego do wcześniej wykonanego obrotu. Macierz symetryczna jest więc macierzą skalowania (w ogólności nierównomiernego) wzdłuż wzajemnie prostopadłych osi *pewnego* układu współrzędnych[5].

Rozważmy teraz macierze ortogonalne Q o wymiarach 3×3. Wszystkie ich wartości własne mają wartość bezwzględną 1, przy czym są dwie możliwości: albo wszystkie trzy wartości własne są rzeczywiste (równe 1 albo -1), albo jedna wartość własna jest rzeczywista,

[3] a właściwie, przestrzeni n-wymiarowej dla dowolnego $n \geqslant 2$

[4] Mówi się o tzw. **krotnościach algebraicznych** wartości własnych; suma tych krotności jest równa n, jeśli zatem pewna wartość własna ma krotność większą niż 1, to macierz ma mniej niż n *różnych* wartości własnych.

[5] Jeśli macierz A nie jest symetryczna, to macierz X spełniająca równość $A = X\Lambda X^{-1}$ z macierzą diagonalną Λ nie istnieje albo nie jest ortogonalna. Nie rozwijam tego tematu, ale zachęcam Czytelników do zajrzenia do notatek z wykładu lub do podręcznika algebry liniowej.

± 1, a pozostałe dwie są sprzężonymi ze sobą liczbami zespolonymi, (c, s) i $(c, -s)$, których iloczyn $c^2 + s^2 = 1$. W pierwszym przypadku macierz jest symetryczna. Opisane przez nią przekształcenie jest skalowaniem trzech wzajemnie prostopadłych osi (o kierunkach wektorów własnych) o czynniki ± 1. Jeśli macierz Q ma wartość własną 1 o krotności 3, to jest to macierz jednostkowa. Jeśli wartość własna -1 ma krotność 1, to macierz Q opisuje odbicie symetryczne względem płaszczyzny, jeśli 2, to jest to macierz odbicia symetrycznego względem prostej (jest to także obrót o kąt π wokół tej prostej), a jeśli 3, to $Q = -I$, a zatem dla dowolnego wektora $\boldsymbol{w}$ jest $Q\boldsymbol{w} = -\boldsymbol{w}$. Reprezentowane przez macierz $-I$ przekształcenie jest odbiciem symetrycznym względem punktu (wektora $\mathbf{0}$).

Macierz ortogonalna Q, która ma zespolone wartości własne, reprezentuje obrót albo złożenie obrotu z odbiciem. Jeśli jej rzeczywista wartość własna jest równa 1, to macierz Q reprezentuje obrót, którego oś ma kierunek wektora własnego związanego z tą wartością własną[6]. Części rzeczywista c i urojona s zespolonej wartości własnej to kosinus i sinus kąta φ tego obrotu. Jeśli macierz Q ma wartość własną -1 i dwie zespolone wartości własne, to przekształcenie reprezentowane przez tę macierz jest złożeniem dwóch przekształceń: obrotu o kąt φ wokół osi o kierunku wektora własnego związanego z wartością własną -1 i odbicia symetrycznego względem płaszczyzny prostopadłej do tej osi.

Znając dowolną rzeczywistą wartość własną λ macierzy A, możemy znaleźć związane z nią wektory własne; są nimi wszystkie (oprócz zerowego) wektory prostopadłe do wierszy macierzy $A - \lambda I$. Ich znalezienie jest szczególnie łatwe w przypadku macierzy ortogonalnej 3×3, bo jej wartością własną jest zawsze liczba rzeczywista 1 lub -1.

Dowolna macierz 3×3 ma pewną wartość własną rzeczywistą, której znalezienie wymaga rozwiązania równania trzeciego stopnia $\det(A - \lambda I) = 0$. Znane od XVI wieku wzory Cardana są niezbyt praktyczną metodą rozwiązywania takich równań, dlatego lepiej jest użyć którejś z uniwersalnych metod numerycznych rozwiązywania równań nieliniowych. Rzeczywiste wartości własne macierzy A o współczynnikach a_{ij} należą do przedziału $[a, b]$, który można znaleźć w taki sposób: niech $\{i, j, k\} = \{1, 2, 3\}$ i niech $r_i = |a_{ij}| + |a_{ik}|$. Można przyjąć $a = \min_{i\in\{1,2,3\}}(a_{ii} - r_i)$, $b = \max_{i\in\{1,2,3\}}(a_{ii} + r_i)$, i jeśli funkcja $f(\lambda) = \det(A - \lambda I)$ w punktach a i b ma wartości różne od zera, to mają one przeciwne znaki, co umożliwia użycie metody bisekcji[7]. Wartości własne macierzy A o wymiarach 3×3 spełniają równości $\lambda_1 + \lambda_2 + \lambda_3 = a_{11} + a_{22} + a_{33}$ oraz $\lambda_1\lambda_2\lambda_3 = \det A$. Na tej podstawie, znając jedną wartość własną, λ_1, można (jeśli $\lambda_1 \neq 0$) otrzymać równanie kwadratowe

$$\lambda^2 + (\lambda_1 - a_{11} - a_{22} - a_{33})\lambda + \det A/\lambda_1 = 0,$$

którego rozwiązaniami są pozostałe dwie wartości własne macierzy A. Opracowanie szczegółów i implementację algorytmu znajdowania wartości i wektorów własnych macierzy 3×3 pozostawiam jako ćwiczenie.

Na podstawie znanego twierdzenia algebry liniowej, dla dowolnej (także prostokątnej) macierzy A o współczynnikach rzeczywistych istnieje jej **rozkład względem wartości szczególnych** (*singular value decomposition*, *SVD*), tj. macierze ortogonalne U i V oraz diagonalna

[6] Wektor $\boldsymbol{v}$ jest związanym z wartością własną 1 wektorem własnym macierzy $R_{\boldsymbol{v},\varphi}$ określonej wzorem (5.18).

[7] Inne metody, na przykład metoda siecznych, mogą działać szybciej (i warto je wypróbować), ale nie gwarantują znalezienia rozwiązania dla każdej macierzy.

macierz Σ o nieujemnych współczynnikach[8], takie że $A = U\Sigma V^T$. Zobaczmy związek tych macierzy z postawionym problemem.

Wyznacznik każdej macierzy ortogonalnej jest równy +1 albo −1. Macierz ortogonalna 3×3 o dodatnim wyznaczniku jest macierzą obrotu w przestrzeni $\mathbb{R}^3$. Zatem rozkład SVD, w którym macierze U i V mają wyznacznik +1, jest potrzebnym rozkładem macierzy A, bo macierz diagonalna Σ jest macierzą skalowania. Jeśli obie macierze U i V mają wyznaczniki ujemne, to możemy je zastąpić przez $-U$ i $-V$. Jeśli zaś tylko jedna z nich, na przykład U ma wyznacznik −1, to zamiast U i Σ możemy przyjąć macierze UD i $D\Sigma$, otrzymane przy użyciu macierzy diagonalnej D, która ma na diagonali współczynniki 1, 1 i −1. Zmienimy w ten sposób zwrot ostatniej kolumny macierzy U i znak ostatniego współczynnika (skalowania osi z) na diagonali macierzy Σ.

Łatwiej niż rozkład SVD można znaleźć tzw. **rozkład biegunowy** macierzy kwadratowej, tj. macierz ortogonalną Q i symetryczną S, takie że $A = QS$. Przekształcając rozkład SVD, możemy napisać $A = U\Sigma V^T = UV^TV\Sigma V^T$. Iloczyn UV^T macierzy ortogonalnych jest macierzą ortogonalną, a macierz $V\Sigma V^T$ jest symetryczna. Możemy zatem przyjąć $Q = UV^T$ oraz $S = V\Sigma V^T$.

Jeśli macierz A jest nieosobliwa, to jej rozkład biegunowy można znaleźć przy użyciu następującego **algorytmu Highama**: po przyjęciu macierzy $A_0 = A$ obliczamy w kolejnych iteracjach macierze

$$A_k = \frac{1}{2}(A_{k-1} + A_{k-1}^{-T}), \quad k = 1, 2, \ldots$$

W każdej iteracji trzeba obliczyć macierz A_{k-1}^{-T}, tj. transpozycję odwrotności macierzy A_{k-1}.[9] Otrzymany ciąg macierzy zbiega do macierzy ortogonalnej Q, przy czym zbieżność jest dość szybka i w wielu przypadkach wystarczy wykonać tylko kilka iteracji. Możemy następnie znaleźć macierz symetryczną $S = Q^TA$.

Wartości i wektory własne macierzy S można znaleźć metodą wspomnianą wcześniej. Wektory własne są kolumnami macierzy V; znając ją, można obliczyć macierz $U = QV$. W typowych zastosowaniach w grafice komputerowej dokładność algorytmu opartego na tym opisie powinna być wystarczająca[10].

Jednym z etapów rejestrowania ruchu (*motion capture*, zobacz [34]) jest określenie położenia i zorientowania w przestrzeni używanych w tej technice kamer. W tym celu na pod-

[8] Współczynniki na diagonali macierzy Σ są nazywane **wartościami szczególnymi** macierzy A.

[9] Aby rozwiązać układ równań liniowych $A\boldsymbol{x} = \boldsymbol{b}$, nie trzeba znać macierzy A^{-1}; wystarczy znać rozkład macierzy A na czynniki trójkątne. Co więcej, algorytm polegający na znalezieniu macierzy A^{-1} i pomnożeniu przez nią wektora $\boldsymbol{b}$ jest bardziej kosztowny i mniej dokładny niż rozwiązanie układów z macierzami trójkątnymi (np. znalezionymi przez procedurę `M4x4LUDecompf`, zobacz podrozdz. 5.7). Algorytm Highama jest jednym z nielicznych algorytmów numerycznych, w których jawne wyznaczanie odwrotności macierzy jest konieczne.

[10] Przypomnę, że w obliczeniach przy użyciu arytmetyki zmiennopozycyjnej występują błędy zaokrągleń, których skutkiem są niedokładne wyniki. Opisane tu rozkłady macierzy o wymiarach większych niż 3×3, potrzebne w różnych zastosowaniach, trzeba wyznaczać bardziej wyrafinowanymi metodami.

stawie zarejestrowanych przez te kamery obrazów, na których są widoczne rozmieszczone w przestrzeni znaczniki, układane są równania, których rozwiązanie prowadzi do znalezienia macierzy opisujących rozmieszczenie (przesunięcia i obroty) poszczególnych kamer. Skutkiem ograniczonej dokładności obrazów i popełnionych w obliczeniach błędów zaokrągleń jest otrzymanie przekształcenia, którego część liniowa jest opisana przez nieortogonalną macierz A. Właśnie w tym zastosowaniu algorytm Highama może pomóc: będąca czynnikiem rozkładu biegunowego macierz Q „najlepiej ze wszystkich macierzy ortogonalnych" przybliża macierz A.[11] Z istnienia rozkładu biegunowego wynika, że macierz A opisuje złożenie skalowania z izometrią (obrotem). Zastąpienie jej przez czynnik Q eliminuje niepożądane skalowanie, kompensując wspomniane błędy.

Rozkładanie macierzy na czynniki reprezentujące przekształcenia elementarne może się przydać w animacji. Przypuśćmy, że ruch pewnego obiektu jest otrzymany przez animowanie macierzy przekształcenia modelu; mamy dane macierze M_i, z których każda opisuje przekształcenie nadające obiektowi położenie w chwili t_i. Znalezienie przekształceń nadających położenia pośrednie wymaga interpolacji położeń danych, ale dokonanie interpolacji poszczególnych *współczynników* macierzy w przypadku, gdy zadane przekształcenia opisują obroty z przesunięciami, prowadzi do otrzymania przekształceń nieizometrycznych. Obiekt sztywny w tak otrzymanym ruchu zmieniałby kształt, więc trzeba postępować inaczej.

Dla uproszczenia rozważmy dane macierze M_0 i M_1 o wymiarach 4×4, opisujące położenia obiektu w chwilach 0 i 1. Macierz M_t nadającą obiektowi położenie odpowiednie w chwili t możemy otrzymać po rozłożeniu macierzy danych: niech macierz T_i opisuje przesunięcie, Q_i obrót, a S_i skalowanie, takie że $M_i = T_i Q_i S_i$. Obie macierze Q_i są ortogonalne, a S_i symetryczne. Możemy przyjąć macierze $T_t = (1-t)T_0 + tT_1$ oraz $S_t = (1-t)S_0 + tS_1$, ponieważ pierwsza z nich opisuje (interpolowane) przesunięcie, a druga, symetryczna dla każdego t, opisuje skalowanie obiektu wzdłuż pewnych wzajemnie prostopadłych osi (zależnych od t); jeśli przekształcenie ma być izometrią (w animacji bryły sztywnej), to macierze S_0, S_1 i S_t są macierzą jednostkową.

Aby dokonać interpolacji obrotów, możemy znaleźć wektory osi i kąty obrotów reprezentowanych przez macierze ortogonalne Q_0 i Q_1, utworzyć reprezentujące te obroty kwaterniony, dokonać interpolacji łukowej kwaternionów i skonstruować macierz Q_t na podstawie reprezentacji kwaternionowej obrotu w chwili t. Podrozdział A.4 zawiera dokładny opis tej reprezentacji. Mając macierze T_t, Q_t i S_t, możemy za macierz M_t przyjąć ich iloczyn.

A.4. Kwaterniony i obroty

Kwaterniony są wektorami w przestrzeni $\mathbb{R}^4$ z określonymi działaniami dodawania (zwykłego) i mnożenia, którego definicja jest niżej. Kwaternion $q = (a, x, y, z)$ możemy przedstawić w postaci $q = (a, \boldsymbol{b})$, w której wyróżniamy **część skalarną** $a \in \mathbb{R}$ (pierwszą współrzędną) i **część wektorową** $\boldsymbol{b} = (x, y, z) \in \mathbb{R}^3$. Korzystając z tego zapisu, można definicję mnożenia

[11] Miarą błędu przybliżenia jest w tym przypadku tzw. **norma druga indukowana macierzy**, $\|\cdot\|_2$. Jeśli W oznacza macierz ortogonalną 3×3, to liczba $\|A - W\|_2$ jest najmniejsza, gdy $W = Q$.

kwaternionów przedstawić wzorem

$$(a_1, \boldsymbol{b}_1) \cdot (a_2, \boldsymbol{b}_2) = (a_1 a_2 - \langle \boldsymbol{b}_1, \boldsymbol{b}_2 \rangle, a_1 \boldsymbol{b}_2 + \boldsymbol{b}_1 a_2 + \boldsymbol{b}_1 \wedge \boldsymbol{b}_2). \tag{A.1}$$

Wzór ten przypomina definicję mnożenia liczb zespolonych (F.1); najbardziej widoczna różnica to składnik $\boldsymbol{b}_1 \wedge \boldsymbol{b}_2$ (iloczyn wektorowy wektorów $\boldsymbol{b}_1$ i $\boldsymbol{b}_2$) w części wektorowej iloczynu. Z powodu tego składnika, który zmienia zwrot po przestawieniu argumentów, mnożenie kwaternionów jest nieprzemienne.

Aby ułatwić zbadanie własności mnożenia, kwaternionowi $q = (a, x, y, z) = (a, \boldsymbol{b})$ przyporządkujemy macierz

$$Q = \left[\begin{array}{c|ccc} a & -x & -y & -z \\ \hline x & a & -z & y \\ y & z & a & -x \\ z & -y & x & a \end{array}\right] = \left[\begin{array}{c|c} a & -\boldsymbol{b}^T \\ \hline \boldsymbol{b} & aI_3 + \boldsymbol{b} \wedge I_3 \end{array}\right].$$

Oczywiście, każdej macierzy utworzonej z czterech liczb zgodnie z tym schematem odpowiada pewien kwaternion, który jest jej pierwszą kolumną. Wykonując stosowne rachunki, możemy sprawdzić, że sumie kwaternionów q_1 i q_2 odpowiada suma przyporządkowanych im macierzy, $Q_1 + Q_2$, a ponadto $q_1 \cdot q_2 = Q_1 q_2$, skąd dalej wynika, że iloczyn macierzy $Q_1 Q_2$ odpowiada iloczynowi kwaternionów $q_1 \cdot q_2$. Mnożenie macierzy jest łączne i rozdzielne względem dodawania, zatem także mnożenie kwaternionów ma te własności[12].

Z punktu widzenia algebry zbiór kwaternionów z opisanymi wyżej działaniami jest **ciałem nieprzemiennym**. Tradycyjnie oznacza się je symbolem $\mathbb{H}$, dla uczczenia sir Williama R. Hamiltona, który 16 października 1843 r. odkrył je w Dublinie [43][13]. Hamilton wymyślił wtedy opisany tu sposób mnożenia czwórek liczb rzeczywistych. Razem ze zwykłym dodawaniem wektorów w $\mathbb{R}^4$ spełnia on wszystkie warunki potrzebne do otrzymania ciała, z wyjątkiem przemienności. Kwintesencją tego mnożenia są cztery, a właściwie dziesięć równości

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{i} \cdot \mathbf{j} \cdot \mathbf{k} = -1,$$

zapisanych przy użyciu symboli $\mathbf{i} = (0,1,0,0)$, $\mathbf{j} = (0,0,1,0)$, $\mathbf{k} = (0,0,0,1)$ oraz $-1 = (-1,0,0,0)$.

Kwaternion sprzężony z $q = (a, \boldsymbol{b})$ to kwaternion $(a, -\boldsymbol{b})$, który oznaczamy symbolem $\overline{q}$; **wartość bezwzględna** kwaternionu $q = (a, \boldsymbol{b}) = (a, x, y, z)$ jest liczbą rzeczywistą

$$|q| = \sqrt{a^2 + \langle \boldsymbol{b}, \boldsymbol{b} \rangle} = \sqrt{a^2 + x^2 + y^2 + z^2}.$$

Wartość bezwzględna kwaternionu jest więc euklidesową długością wektora (a, x, y, z) i jest spełniona równość $|q| = |\overline{q}|$. **Kwaternion zerowy**, $(0, \mathbf{0})$, jest jedynym kwaternionem, którego wartość bezwzględna jest równa 0.

[12] Zwróćmy uwagę, że łączność mnożenia kwaternionów nie jest sprzeczna z faktem, że iloczyn wektorowy w $\mathbb{R}^3$, użyty w definicji tego mnożenia, nie jest działaniem łącznym.

[13] Byłem w tamtym miejscu; kwaterniony cały czas tam są, ale można je też dostrzec wszędzie indziej.

Zobaczmy, jak to wygląda w notacji macierzowej. Jeśli kwaternion q jest związany z macierzą Q, to sprzężonemu z nim kwaternionowi $\overline{q}$ odpowiada macierz transponowana Q^T. Możemy sprawdzić, że iloczynowi $q \cdot \overline{q}$ odpowiada macierz $QQ^T = |q|^2 I_4$. Macierz Q jest więc iloczynem pewnej macierzy ortogonalnej i liczby $|q|$. Wyznacznik macierzy Q jest nieujemny: $\det Q = (a^2+x^2+y^2+z^2)^2 = |q|^4$, a stąd (i z twierdzenia Cauchy'ego, s. 107) wynika, że dla dowolnych kwaternionów q_1, q_2 zachodzi równość

$$|q_1 \cdot q_2| = |q_1||q_2|.$$

Ponadto z równości $(Q_1Q_2)^T = Q_2^T Q_1^T$ dla *dowolnych* macierzy Q_1, Q_2, których iloczyn istnieje (zobacz podrozdz. 5.1), wynika równość $\overline{q_1 \cdot q_2} = \overline{q}_2 \cdot \overline{q}_1$ dla dowolnych kwaternionów q_1, q_2.

Kwaternion niemy ma część wektorową równą $\mathbf{0}$ (odpowiada mu macierz diagonalna aI_4). Zauważmy, że mnożenie kwaternionów niemych daje w wyniku kwaternion niemy, o części skalarnej równej iloczynowi części skalarnych czynników; można więc utożsamić kwaterniony nieme z liczbami rzeczywistymi i wtedy dodawanie oraz mnożenie kwaternionów i liczb dają takie same wyniki[14]. Zauważmy jeszcze dwie rzeczy: jeśli dowolny argument mnożenia jest kwaternionem niemym, to kolejność tych argumentów można zmienić, a ponadto wzór opisujący wartość bezwzględną dowolnego kwaternionu możemy teraz zapisać w postaci $|q| = \sqrt{q \cdot \overline{q}}$ (bo pod pierwiastkiem jest kwaternion niemy utożsamiony z liczbą rzeczywistą nieujemną).

Jedynka kwaternionowa to kwaternion $(1, \mathbf{0})$. Odpowiada jej macierz jednostkowa 4×4. Jedynka jest elementem neutralnym mnożenia, tj. $(1, \mathbf{0}) \cdot q = q \cdot (1, \mathbf{0}) = q$ dla każdego q i jest to jedyny kwaternion o tej własności. Dla skrótu kwaternion zerowy i jedynkę można zapisywać symbolami 0 i 1, pamiętając, że to kwaterniony.

Kwaternion odwrotny do q to q^{-1}, taki że $q \cdot q^{-1} = q^{-1} \cdot q = (1, \mathbf{0})$. Dla każdego niezerowego kwaternionu istnieje (jeden) kwaternion odwrotny, opisany wzorem

$$q^{-1} = \frac{1}{|q|^2}\overline{q},$$

który przypomina wzór na odwrotność liczby zespolonej. W notacji macierzowej kwaternionowi q^{-1} odpowiada macierz $Q^{-1} = \frac{1}{|q|^2} Q^T$. Z łączności mnożenia kwaternionów wynika, że jeśli kwaterniony q_1 i q_2 nie są zerowe, to

$$(q_1 \cdot q_2)^{-1} = q_2^{-1} \cdot q_1^{-1}.$$

Mając pojęcie odwrotności, można określić dzielenie kwaternionów, a właściwie **dwa dzielenia**, określone wzorami

$$q_1/q_2 = q_1 \cdot q_2^{-1} \quad \text{i} \quad q_2 \backslash q_1 = q_2^{-1} \cdot q_1.$$

[14] Odnotujmy jako ciekawostkę, że dodawanie i mnożenie kwaternionów, których części wektorowe mają ten sam kierunek, jest zgodne z działaniami na liczbach zespolonych. W szczególności jeśli $\boldsymbol{v}$ jest dowolnym wektorem jednostkowym w $\mathbb{R}^3$ oraz $(a_1, b_1)(a_2, b_2) = (a, b) \in \mathbb{C}$, to $(a_1, b_1\boldsymbol{v}) \cdot (a_2, b_2\boldsymbol{v}) = (a, b\boldsymbol{v}) \in \mathbb{H}$.

Rzecz w tym, że na ogół $q_1 \cdot q_2^{-1} \neq q_2^{-1} \cdot q_1$ (równość zachodzi wtedy, gdy części wektorowe obu kwaternionów są liniowo zależne). Dlatego nie będziemy pisać kwaternionowych wyrażeń z poziomą kreską ułamkową, chyba że mianownik (lub licznik) jest liczbą rzeczywistą (albo kwaternionem niemym).

Kwaternion, którego część skalarna jest równa 0, nazywamy **kwaternionem czystym**, a **kwaternion jednostkowy** to taki, którego wartość bezwzględna jest równa 1. Ponieważ wartość bezwzględna iloczynu kwaternionów jest iloczynem ich wartości bezwzględnych, iloczyn kwaternionów jednostkowych jest kwaternionem jednostkowym. Co więcej, odwrotnością kwaternionu jednostkowego jest jego kwaternion sprzężony. Kwaternionom jednostkowym odpowiadają macierze ortogonalne 4×4.

Dowolny kwaternion można przedstawić w **postaci trygonometrycznej**: dla każdego kwaternionu q istnieje wektor jednostkowy $\boldsymbol{v}$ i liczba α, takie że

$$q = |q|(\cos\alpha, \boldsymbol{v}\sin\alpha). \tag{A.2}$$

Czynnik $(\cos\alpha, \boldsymbol{v}\sin\alpha)$ jest kwaternionem jednostkowym. Dla kwaternionów niemych liczba α jest całkowitą wielokrotnością liczby π, a kierunek wektora $\boldsymbol{v}$ jest nieokreślony. Dla każdego kwaternionu q i każdej liczby naturalnej n zachodzi równość

$$q^n = |q|^n(\cos n\alpha, \boldsymbol{v}\sin n\alpha).$$

Dzięki temu możemy określić **potęgowanie kwaternionów** wzorem

$$q^t = |q|^t(\cos t\alpha, \boldsymbol{v}\sin t\alpha), \tag{A.3}$$

w którym może wystąpić dowolny wykładnik rzeczywisty t.[15]

Obroty w przestrzeni $\mathbb{R}^3$ są reprezentowane przez kwaterniony jednostkowe. Weźmy dowolny wektor jednostkowy $\boldsymbol{v} \in \mathbb{R}^3$ i liczbę φ. Obrotowi o kąt φ wokół prostej o kierunku wektora $\boldsymbol{v}$ przyporządkujemy kwaternion $q = (\cos\frac{\varphi}{2}, \boldsymbol{v}\sin\frac{\varphi}{2})$. Jest on oczywiście jednostkowy. Wektorowi $\boldsymbol{w} \in \mathbb{R}^3$, który zamierzamy obrócić, przyporządkujemy kwaternion czysty $w = (0, \boldsymbol{w})$. Udowodnimy, że kwaternion

$$u = q \cdot w \cdot q^{-1} \tag{A.4}$$

jest czysty, tj. $u = (0, \boldsymbol{u})$, a jego część wektorowa jest obrazem wektora $\boldsymbol{w}$ w rozpatrywanym obrocie.

Oznaczmy $s = \sin\frac{\varphi}{2}$ i $c = \cos\frac{\varphi}{2}$. Liczymy

$$\begin{aligned} q \cdot w \cdot q^{-1} &= (c, s\boldsymbol{v}) \cdot (0, \boldsymbol{w}) \cdot (c, -s\boldsymbol{v}) = \big(-s\langle\boldsymbol{v}, \boldsymbol{w}\rangle, c\boldsymbol{w} + s\boldsymbol{v} \wedge \boldsymbol{w}\big) \cdot (c, -s\boldsymbol{v}) \\ &= \big(-cs\langle\boldsymbol{v}, \boldsymbol{w}\rangle + cs\langle\boldsymbol{w}, \boldsymbol{v}\rangle + s\langle\boldsymbol{v} \wedge \boldsymbol{w}, \boldsymbol{v}\rangle, \\ &\qquad s^2\langle\boldsymbol{v}, \boldsymbol{w}\rangle\boldsymbol{v} + c^2\boldsymbol{w} + cs\boldsymbol{v} \wedge \boldsymbol{w} - cs\boldsymbol{w} \wedge \boldsymbol{v} - s^2(\boldsymbol{v} \wedge \boldsymbol{w}) \wedge \boldsymbol{v}\big). \end{aligned}$$

[15] Ale aby to działanie było dobrze określone (tj. miało jednoznaczny wynik) dla każdego $t \in \mathbb{R}$, liczbę α należy wybierać z przedziału $(-\pi, \pi)$. Wtedy jeśli $t_1\alpha, t_2\alpha \in (-\pi, \pi)$, to zachodzą równości $q^{t_1} \cdot q^{t_2} = q^{t_1+t_2}$ i $(q^{t_1})^{t_2} = q^{t_1 t_2}$. Jeśli $q = (a, \boldsymbol{0})$ i $a < 0$ (czyli $\alpha = \pi$), to q^t jest określone tylko dla całkowitych wykładników t.

Zgodnie z zapowiedzią, część skalarna powyższego iloczynu jest równa 0: dwa pierwsze składniki opisującego ją wyrażenia mają przeciwne znaki, a w trzecim składniku mamy iloczyn skalarny wektorów $\boldsymbol{v} \wedge \boldsymbol{w}$ i $\boldsymbol{v}$, które są wzajemnie prostopadłe. Obliczmy zatem część wektorową,

$$\boldsymbol{u} = s^2\langle \boldsymbol{v}, \boldsymbol{w}\rangle \boldsymbol{v} + c^2 \boldsymbol{w} + cs\boldsymbol{v} \wedge \boldsymbol{w} - cs\boldsymbol{w} \wedge \boldsymbol{v} - s^2(\boldsymbol{v} \wedge \boldsymbol{w}) \wedge \boldsymbol{v}.$$

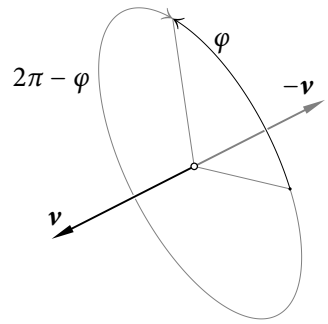
Zauważmy, że $s^2 = 1 - c^2$; stąd mamy

$$s^2\langle \boldsymbol{v}, \boldsymbol{w}\rangle \boldsymbol{v} + c^2 \boldsymbol{w} = \langle \boldsymbol{v}, \boldsymbol{w}\rangle \boldsymbol{v} + c^2(\boldsymbol{w} - \langle \boldsymbol{v}, \boldsymbol{w}\rangle \boldsymbol{v}).$$

W otrzymanych wzorach występują wektory $\langle \boldsymbol{v}, \boldsymbol{w}\rangle \boldsymbol{v} = \boldsymbol{a}$, $\boldsymbol{v} \wedge \boldsymbol{w} = \boldsymbol{c}$ i $\boldsymbol{c} \wedge \boldsymbol{v} = \boldsymbol{w} - \langle \boldsymbol{v}, \boldsymbol{w}\rangle \boldsymbol{v} = \boldsymbol{b}$ (zobacz rys. 5.6[16]). Stąd na podstawie znanych tożsamości trygonometrycznych $2cs = \sin\varphi$ i $c^2 - s^2 = \cos\varphi$ otrzymujemy

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

czyli wzór (5.17), co kończy dowód. □



Rysunek A.4. Dwa obroty będące tym samym obrotem

Zauważmy, że reprezentacja kwaternionowa obrotu nie jest jednoznaczna: kwaternion $-q$ reprezentuje ten sam obrót co q. Mamy bowiem

$$-q = \left(-\cos\frac{\varphi}{2}, -\boldsymbol{v}\sin\frac{\varphi}{2}\right) = \left(\cos\frac{2\pi-\varphi}{2}, -\boldsymbol{v}\sin\frac{2\pi-\varphi}{2}\right),$$

[16] *Rachunek* dowodzący, że $(\boldsymbol{v} \wedge \boldsymbol{w}) \wedge \boldsymbol{v} = \boldsymbol{w} - \langle \boldsymbol{v}, \boldsymbol{w}\rangle \boldsymbol{v}$ jest taki: przypominamy sobie, że $\boldsymbol{v} \wedge \boldsymbol{w} = -\boldsymbol{w} \wedge \boldsymbol{v}$ oraz $\langle \boldsymbol{v}, \boldsymbol{v}\rangle = 1$ i obliczamy iloczyny kwaternionów

$$\big((0,\boldsymbol{v})\cdot(0,\boldsymbol{v})\big)\cdot(0,-\boldsymbol{w}) = (-\langle \boldsymbol{v},\boldsymbol{v}\rangle, \boldsymbol{0})\cdot(0,-\boldsymbol{w}) = (0,\langle \boldsymbol{v},\boldsymbol{v}\rangle \boldsymbol{w}) = (0,\boldsymbol{w}),$$
$$(0,\boldsymbol{v})\cdot\big((0,\boldsymbol{v})\cdot(0,-\boldsymbol{w})\big) = (0,\boldsymbol{v})\cdot\big(\langle \boldsymbol{v},\boldsymbol{w}\rangle, -\boldsymbol{v}\wedge\boldsymbol{w}\big) = \big(0,\boldsymbol{v}\langle \boldsymbol{v},\boldsymbol{w}\rangle - \boldsymbol{v}\wedge(\boldsymbol{v}\wedge\boldsymbol{w})\big).$$

Mnożenie kwaternionów jest łączne, więc w obu przypadkach wynik jest ten sam. Pozostaje zbadać wyrażenia opisujące jego część wektorową.

czyli reprezentację obrotu o kąt $2\pi - \varphi$ w drugą stronę, tj. wokół osi zorientowanej przeciwnie (rys. A.4). Ponadto jedynka kwaternionowa (a także kwaternion $(-1, \mathbf{0})$) reprezentuje przekształcenie tożsamościowe, czyli obrót o kąt $\varphi = 0$ wokół osi, której kierunek nie jest (i nie musi być) określony.

Powołując się na łączność mnożenia kwaternionów, możemy napisać

$$\begin{aligned} q_2 \cdot \left(q_1 \cdot (0, \boldsymbol{w}) \cdot q_1^{-1}\right) \cdot q_2^{-1} &= (q_2 \cdot q_1) \cdot (0, \boldsymbol{w}) \cdot (q_1^{-1} \cdot q_2^{-1}) \\ &= (q_2 \cdot q_1) \cdot (0, \boldsymbol{w}) \cdot (q_2 \cdot q_1)^{-1}. \end{aligned}$$

Stąd złożenie kolejno wykonanych obrotów reprezentowanych przez kwaterniony q_1 i q_2 jest obrotem reprezentowanym przez iloczyn tych kwaternionów. Powyższy rachunek odpowiada sytuacji, gdy oba obroty są określone w układzie nieruchomym (np. świata) i wtedy ich złożenie jest reprezentowane przez iloczyn $q_2 \cdot q_1$. Obliczając część skalarną i wektorową tego iloczynu, otrzymamy wzory podane na początku rozdziału 8. Jeśli natomiast oba obroty są określone w układzie współrzędnych związanym z obiektem (tj. obracającym się razem z nim), to należy ustawić czynniki w odwrotnej kolejności: $q_1 \cdot q_2$ (zobacz s. 113).

Bezpośrednie stosowanie wzoru (A.4) nie jest zbyt tanie; trzeba wykonać przy tym 24 mnożenia liczb rzeczywistych, podczas gdy pomnożenie wektora współrzędnych jednorodnych punktu przez macierz 4×4, reprezentującą obrót lub dowolne inne przekształcenie afiniczne lub rzutowe, wymaga wykonania tylko 16 mnożeń. Mając kwaternion jednostkowy $q = (a, x, y, z)$, możemy łatwo skonstruować macierz reprezentowanego przezeń obrotu. Niech $w = (0, r, s, t)$. Na podstawie wzoru (A.4) możemy obliczyć

$$\begin{aligned} q \cdot w \cdot q^{-1} &= \begin{bmatrix} a & -x & -y & -z \\ x & a & -z & y \\ y & z & a & -x \\ z & -y & x & a \end{bmatrix} \begin{bmatrix} 0 & -r & -s & -t \\ r & 0 & -t & s \\ s & t & 0 & -r \\ t & -s & r & 0 \end{bmatrix} \begin{bmatrix} a \\ -x \\ -y \\ -z \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ (a^2 + x^2 - y^2 - z^2)r + 2(xy - az)s + 2(xz + ay)t \\ 2(xy + az)r + (a^2 + y^2 - x^2 - z^2)s + 2(yz - ax)t \\ 2(xz - ay)r + 2(yz + ax)s + (a^2 + z^2 - x^2 - y^2)t \end{bmatrix}. \end{aligned}$$

Stąd otrzymamy macierz obrotu reprezentowanego przez kwaternion q:

$$R = \begin{bmatrix} 1 - 2(y^2 + z^2) & 2(xy - az) & 2(xz + ay) \\ 2(xy + az) & 1 - 2(x^2 + z^2) & 2(yz - ax) \\ 2(xz - ay) & 2(yz + ax) & 1 - 2(x^2 + y^2) \end{bmatrix}. \tag{A.5}$$

Przejście od kwaternionowej do macierzowej reprezentacji obrotu jest więc wykonalne bez obliczania wartości jakichkolwiek funkcji przestępnych.

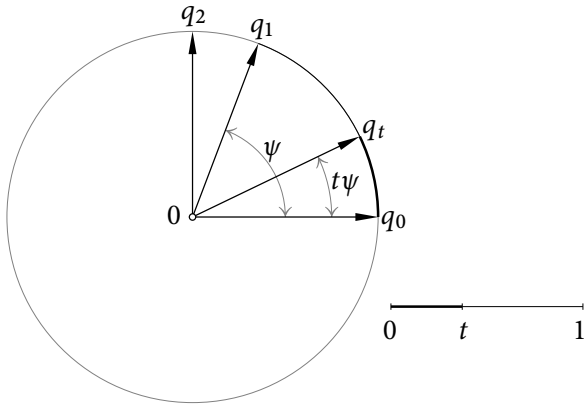
Dysponując kwaternionami, mamy możliwość stosunkowo łatwego dokonania interpolacji położeń kątowych bryły w ruchu kulistym[17], zadanych w wybranych chwilach. W tym celu

[17] Czyli w ruchu obrotowym wokół osi o zmieniającym się kierunku, w każdej chwili przechodzącej przez pewien ustalony punkt.

trzeba skonstruować krzywą, której punktami są kwaterniony jednostkowe, tj. krzywą położoną na sferze jednostkowej w $\mathbb{R}^4$. Krzywa ta ma przechodzić przez podane punkty (kwaterniony odpowiadające kolejno zadanym położeniom kątowym bryły w ruchu), określając jednoznacznie położenia kątowe bryły w innych chwilach. Elementarnym krokiem konstrukcji takich krzywych przechodzących przez wiele zadanych punktów (w tym przykładowej konstrukcji przedstawionej w podrozdz. B.4) jest konstrukcja najkrótszej krzywej o zadanych końcach położonej na sferze jednostkowej.

Przypuśćmy, że dwa kwaterniony, q_0 i q_1, reprezentują pewne obroty, które wyznaczają położenia kątowe dowolnego obiektu w chwilach 0 i 1. Chcielibyśmy interpolować te obroty, tj. dla dowolnego $t \in [0,1]$ znaleźć obrót (czyli odpowiedni kwaternion jednostkowy q_t), który wyznacza położenie „pośrednie" obiektu.

Obrót odpowiadający chwili t moglibyśmy określić przy użyciu jednego z kwaternionów określonych wzorami $\tilde{q}_t = q_0^{1-t} \cdot q_1^t$ albo $\hat{q}_t = q_1^t \cdot q_0^{1-t}$. Podstawiając $t = 0$ do każdego z tych wzorów, otrzymalibyśmy kwaternion q_0, a podstawiając $t = 1$, dostalibyśmy kwaternion q_1. Niestety, brak przemienności mnożenia kwaternionów powoduje, że jeśli części wektorowe kwaternionów q_0 i q_1 mają różne kierunki, to dla $0 < t < 1$ jest $\tilde{q}_t \neq \hat{q}_t$, a więc każdy z tych wzorów opisuje parametryzację innej krzywej na sferze jednostkowej (i żadna z nich nie jest najkrótsza). Aby odwrócić ruch w czasie, należałoby zamienić kwaterniony q_0 i q_1, co doprowadziłoby do otrzymania innych położeń pośrednich[18]. Ponadto, dokonując interpolacji w obróconym układzie współrzędnych, otrzymalibyśmy inny ruch obrotowy. Dlatego metoda interpolacji położeń kątowych oparta na każdym z podanych wyżej wzorów *nie jest poprawna*.



Rysunek A.5. Interpolacja łukowa

Poprawna metoda polega na dokonaniu **interpolacji łukowej kwaternionów**. Na rysunku A.5 jest pokazany przekrój przez sferę jednostkową w $\mathbb{R}^4$ płaszczyzną zawierającą kwaterniony jednostkowe q_0 i q_1, takie że $q_0 \neq q_1$ i $q_0 \neq -q_1$; przekrój ten jest oczywiś-

[18] Zamieniając kwaterniony q_0 i q_1, należałoby zatem zastąpić użyty do interpolacji wzór tym drugim wzorem, ale jak tu dokonać pierwszego wyboru?

cie okręgiem jednostkowym. Kwaterniony q_0 i q_1 jednoznacznie określają najkrótszy łuk na sferze jednostkowej, którego są końcami. Dla chwili $t \in [0,1]$ chcemy skonstruować obrót reprezentowany przez kwaternion q_t, który dzieli ten łuk w proporcji $t : 1-t$. Funkcja, której argumentami są kwaterniony q_0, q_1 i liczba t i której wartością jest ten kwaternion q_t, jest znana pod nazwą Slerp (*Spherical linear interpolation*)[19].

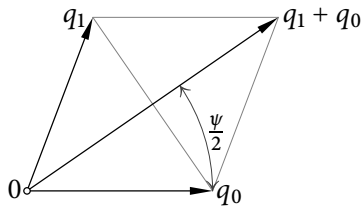
Wzór opisujacy interpolację łukową przy użyciu potęgowania otrzymamy, rozpatrując przyporządkowane kwaternionom macierze. Macierz odpowiadająca dowolnemu kwaternionowi jednostkowemu jest ortogonalna, a zatem reprezentuje pewną izometrię przestrzeni $\mathbb{R}^4$. Łuk, którego końcami są punkty q_0 i q_1, przekształcimy przy użyciu takiej izometrii, aby obrazem q_0 była jedynka kwaternionowa, do czego użyjemy odwrotności macierzy Q_0 przyporządkowanej kwaternionowi q_0. Dla każdego t obrazem punktu q_t na tym łuku jest punkt $\tilde{q}_t = q_0^{-1} \cdot q_t = Q_0^{-1} q_t$, a w szczególności koniec łuku, tj. punkt q_1, przejdzie na punkt $\tilde{q}_1 = q_0^{-1} \cdot q_1 = Q_0^{-1} q_1$. Na podstawie wzoru (A.3) kwaternion $\tilde{q}_t$, dzielący łuk o końcach 1 i $\tilde{q}_1$ w proporcji $t : 1-t$, jest równy $\tilde{q}_1^t$. Stąd dostajemy wzór

$$q_t = Q_0 \tilde{q}_t = q_0 \cdot (q_0^{-1} \cdot q_1)^t. \tag{A.6}$$

W podobny sposób możemy otrzymać wzór

$$q_t = q_1 \cdot (q_1^{-1} \cdot q_0)^{1-t}; \tag{A.7}$$

łatwo możemy też sprawdzić, że $\left(q_0 \cdot (q_0^{-1} \cdot q_1)^t\right) \cdot \left(q_1 \cdot (q_1^{-1} \cdot q_0)^{1-t}\right)^{-1} = (1, \mathbf{0})$, a zatem oba powyższe wzory są równoważne i każdy z nich opisuje funkcję Slerp. Rozpatrując ruch obrotowy w układzie współrzędnych, w którym zorientowanie początkowe jest opisane przez jedynkę, a końcowe przez kwaternion $\tilde{q}_1$, możemy zauważyć, że jeśli podczas animowania obiektu zmieniamy parametr t ze stałą szybkością, dokonujemy interpolacji łukowej kwaternionów i określamy na tej podstawie chwilowe przekształcenia obiektu, to obiekt ten obraca się wokół pewnej ustalonej osi ze stałą prędkością kątową.



Rysunek A.6. Znajdowanie kąta między kwaternionami jednostkowymi

Zobaczmy inny, algebraicznie równoważny sposób dokonywania interpolacji łukowej. Miarę kąta ψ między kwaternionami q_0 a q_1 możemy znaleźć, traktując je jak wektory w $\mathbb{R}^4$ i obliczając ich iloczyn skalarny, równy $\cos\psi$. Jeśli jednak kąt ψ jest bliski zera, to obliczenie

[19] Nazwę tę wprowadził Ken Shoemake w 1985 r.

$\sin\psi = \sqrt{1-\cos^2\psi}$ może wskutek błędów zaokrągleń dać bardzo niedokładny wynik. Metoda dokładniejsza opiera się na równościach $|q_1 - q_0| = 2\sin\frac{\psi}{2}$, $|q_1 + q_0| = 2\cos\frac{\psi}{2}$ (rys. A.6), z których wynikają wzory[20]

$$\sin\psi = \frac{1}{2}|q_1 - q_0||q_1 + q_0|, \quad \cos\psi = \frac{1}{4}\big(|q_1 + q_0|^2 - |q_1 - q_0|^2\big), \quad \psi = 2\operatorname{arc\,tg}\frac{|q_1 - q_0|}{|q_1 + q_0|}.$$

Dalsze rachunki są takie: niech q_2 oznacza leżący na rozważanym okręgu kwaternion, który jest wektorem prostopadłym do q_0 (rys. A.5). Wtedy

$$q_1 = q_0\cos\psi + q_2\sin\psi,$$
$$q_t = q_0\cos t\psi + q_2\sin t\psi.$$

Wyznaczając q_2 na podstawie pierwszego równania i wstawiając do drugiego, po uporządkowaniu otrzymamy wzór

$$\operatorname{Slerp}(q_0, q_1; t) = q_t = \frac{q_0\sin(1-t)\psi + q_1\sin t\psi}{\sin\psi}. \tag{A.8}$$

Stosowanie wzoru (A.8) zamiast (A.6) lub (A.7) i (A.3) też wymaga użycia funkcji trygonometrycznych. Można tego uniknąć w szczególnym przypadku, gdy chcemy znaleźć punkt w połowie łuku. Możemy wtedy użyć wzoru

$$\operatorname{Slerp}(q_0, q_1; 1/2) = q_{1/2} = \frac{q_0 + q_1}{|q_0 + q_1|}.$$

Łuk łączący kwaterniony q_0 i q_1 możemy dzielić rekurencyjnie na połowy, ćwiartki itd., wykonując tylko dodawania, mnożenia i dzielenia liczb rzeczywistych oraz obliczając pierwiastek kwadratowy.

Uwaga: Chcąc interpolować położenia kątowe obiektu reprezentowane przez kwaterniony q_0 i q_1, możemy dzielić w odpowiednich proporcjach łuk o końcach q_0 i q_1 lub łuk o końcach q_0 i $-q_1$. Zwykle wybieramy łuk krótszy, tj. jeśli kosinus kąta ψ między q_0 a q_1 jest ujemny, to wybieramy drugi z tych dwóch łuków. Ruchy określone przez oba łuki są obracaniem w przeciwne strony. Wybór dłuższego łuku oznacza, że obiekt obróci się o kąt większy niż π.

Listing A.3 przedstawia garść procedur, które mogą być użyte do opisanych wyżej obliczeń z kwaternionami. Procedura `QuatMultf` mnoży dwa kwaterniony. Funkcje `QuatAbsf` i `QuatArgf` obliczają wartość bezwzględną i argument (czyli liczbę α występującą w postaci trygonometrycznej kwaternionu (A.2)). Procedury `QuatLDivf` i `QuatRDivf` wykonują oba dzielenia kwaternionów, lewostronne i prawostronne.

Procedura `M4x4QuatRotationf` na podstawie wzoru (A.5) konstruuje macierz 4×4, której blok 3×3 reprezentuje obrót określony przez dany kwaternion *jednostkowy*. Procedury `QuatRotVf` i `RotVQuatf` dokonują konwersji reprezentacji obrotów; pierwsza znajduje

[20] Z uwagi na błędy zaokrągleń lepiej jest obliczać kosinus ψ jako iloczyn skalarny w $\mathbb{R}^4$: $\cos\psi = \langle q_0, q_1\rangle$.

Listing A.3. Procedury obliczeń z kwaternionami

C

```
void QuatMultf ( float q[4], const float q1[4], const float q2[4] )
{
  q[0] = q1[0]*q2[0] - q1[1]*q2[1] - q1[2]*q2[2] - q1[3]*q2[3];
  q[1] = q1[1]*q2[0] + q1[0]*q2[1] - q1[3]*q2[2] + q1[2]*q2[3];
  q[2] = q1[2]*q2[0] + q1[3]*q2[1] + q1[0]*q2[2] - q1[1]*q2[3];
  q[3] = q1[3]*q2[0] - q1[2]*q2[1] + q1[1]*q2[2] + q1[0]*q2[3];
} /*QuatMultf*/

float QuatAbsf ( float q[4] )
{
  return sqrt ( V4DotProductf ( q, q ) );
} /*QuatAbsf*/

double QuatArgf ( float q[4] )
{
  return atan2 ( sqrt ( V3DotProductf ( &q[1], &q[1]) ), q[0] );
} /*QuatArgf*/

void QuatLDivf ( float q[4], const float q2[4], const float q1[4] )
{
  float q2i[4], s;

  s = V4DotProductf ( q2, q2 );
  if ( s > 0.0 ) {
    q2i[0] = q2[0]/s;  q2i[1] = -q2[1]/s;
    q2i[2] = -q2[2]/s;  q2i[3] = -q2[3]/s;
    QuatMultf ( q, q2i, q1 );
  }
} /*QuatLDivf*/

void QuatRDivf ( float q[4], const float q1[4], const float q2[4] )
{
  float q2i[4], s;

  s = V4DotProductf ( q2, q2 );
  if ( s > 0.0 ) {
    q2i[0] = q2[0]/s;  q2i[1] = -q2[1]/s;
    q2i[2] = -q2[2]/s;  q2i[3] = -q2[3]/s;
    QuatMultf ( q, q1, q2i );
  }
} /*QuatRDivf*/

void M4x4QuatRotationf ( GLfloat a[16], float q[4] )
{
  double xx, yy, zz, xa, xy, xz, ya, yz, za;
```

```
46:
47:   xx = 2.0*q[1]*q[1];  yy = 2.0*q[2]*q[2];  zz = 2.0*q[3]*q[3];
48:   xa = 2.0*q[1]*q[0];  xy = 2.0*q[1]*q[2];  xz = 2.0*q[1]*q[3];
49:   ya = 2.0*q[2]*q[0];  yz = 2.0*q[2]*q[3];  za = 2.0*q[3]*q[0];
50:   memset ( a, 0, 16*sizeof(GLfloat) );
51:   a[0] = 1.0-(yy+zz);  a[1] = xy-za;  a[2] = xz-ya;
52:   a[4] = xy-za;  a[5] = 1.0-(xx+zz);  a[6] = yz+xa;
53:   a[8] = xa+ya;  a[9] = yz-xa;  a[10] = 1.0-(xx+yy);
54:   a[15] = 1.0;
55: } /*M4x4QuatRotationf*/
56:
57: void QuatRotVf ( float q[4], const float v[3], double phi )
58: {
59: #define TOL 1.0e-6
60:   float d;
61:
62:   d = V3DotProductf ( v, v );
63:   if ( d > TOL*TOL ) {
64:     d = sin ( 0.5*phi )/sqrt ( d );
65:     q[0] = cos ( 0.5*phi );
66:     q[1] = d*v[0];  q[2] = d*v[1];  q[3] = d*v[2];
67:   }
68:   else
69:     { q[0] = 1.0, q[1] = q[2] = q[3] = 0.0; }
70: } /*QuatRotVf*/
71:
72: void RotVQuatf ( float v[3], double *phi, const float q[4] )
73: {
74:   float s, d;
75:
76:   if ( (s = V3DotProductf ( &q[1], &q[1] )) < TOL*TOL ||
77:        (d = q[0]*q[0] + s) < TOL*TOL )
78:     { v[0] = 1.0, v[1] = v[2] = 0.0; *phi = 0.0; }
79:   else {
80:     s = sqrt ( s );
81:     *phi = 2.0*atan2 ( s, q[0] );
82:     d = 1.0/(sqrt ( d )*s);
83:     v[0] = d*q[1];  v[1] = d*q[2];  v[2] = d*q[3];
84:   }
85: #undef TOL
86: } /*RotVQuatf*/
87:
88: void QuatAnglef ( double *psi, float *spsi, float *cpsi,
89:                   float q0[4], float q1[4] )
90: {
91:   float a[4], p, r;
92:
```

```
  V4Addf ( a, q0, q1 );
  p = sqrt ( V4DotProductf ( a, a ) );
  V4Subtractf ( a, q0, q1 );
  r = sqrt ( V4DotProductf ( a, a ) );
  if ( psi )  *psi = 2.0*atan2 ( r, p );
  if ( spsi ) *spsi = 0.5*p*r;
  if ( cpsi ) *cpsi = V4DotProductf ( q0, q1 );
} /*QuatAnglef*/

void QuatArcInterpf ( float qt[4], float q0[4], float q1[4],
                      double psi, float spsi, float t )
{
  float stp, s1tp;

  stp = sin ( t*psi );  s1tp = sin ( (1.0-t)*psi );
  qt[0] = (s1tp*q0[0] + stp*q1[0]) / spsi;
  qt[1] = (s1tp*q0[1] + stp*q1[1]) / spsi;
  qt[2] = (s1tp*q0[2] + stp*q1[2]) / spsi;
  qt[3] = (s1tp*q0[3] + stp*q1[3]) / spsi;
} /*QuatArcInterp*/

void QuatSlerpf ( float qt[4], float q0[4], float q1[4], float t )
{
  float psi, spsi;

  QuatAnglef ( &psi, &spsi, NULL, q0, q1 );
  if ( spsi > 0.0 )
    QuatArcInterpf ( qt, q0, q1, psi, spsi, t );
  else
    memcpy ( qt, q0, 4*sizeof(float) );
} /*QuatSlerpf*/
```

kwaternion jednostkowy q reprezentujący obrót o kąt φ wokół osi o kierunku wektora $\boldsymbol{v}$, a druga, mając dany kwaternion q (niekoniecznie jednostkowy), znajduje odpowiedni wektor $\boldsymbol{v}$ i kąt φ.

Procedura `QuatAnglef` oblicza kąt ψ między danymi dwoma kwaternionami jednostkowymi oraz jego sinus i kosinus. Procedura `QuatInterpf` dokonuje interpolacji łukowej kwaternionów jednostkowych, przy czym jej parametry zawierają informację redundantną: kąt ψ i jego sinus. Przed wielokrotnym wywoływaniem procedury interpolacji dla różnych argumentów t można je obliczyć (przy użyciu `QuatAnglef`) tylko raz. Procedura `QuatSlerpf` realizuje funkcję Slerp. Jeśli jednak $\sin\psi = 0$, to albo $q_0 = q_1$ (i łuk jest zdegenerowany do punktu), albo $q_1 = -q_0$ (i wtedy podane końce nie wyznaczają łuku jednoznacznie, każdy półokrąg, którego końce to q_0 i $-q_0$, jest najkrótszym łukiem między nimi). W każdym z tych przypadków procedura podaje wynik $q_t = q_0$.

Opowieść o kwaternionach i obrotach ma ciąg dalszy w podrozdziale B.4.

B

Krzywe i powierzchnie B-sklejane

W wielu zastosowaniach wygodniejsze od krzywych i płatów Béziera są będące ich uogólnieniem krzywe i powierzchnie B-sklejane. Nie ma tu miejsca na szczegółowy opis ich własności ani na szerszy przegląd algorytmów ich przetwarzania. Przedstawiając tylko minimum informacji umożliwiających konstruowanie krzywych interpolacyjnych i napisanie szaderów służących do rysowania powierzchni, zachęcam Czytelników do eksperymentowania i zbierania doświadczeń. Mając je, można sięgnąć do literatury (polecam moją książkę [41], oczywiście są też inne), aby krzywe i powierzchnie B-sklejane lepiej poznać i tym piękniej rysować.

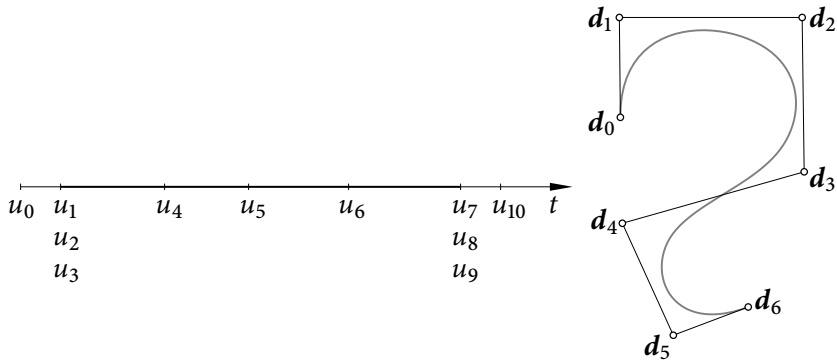
B.1. Określenie funkcji, krzywych i płatów B-sklejanych

Podstawą reprezentacji krzywych i płatów B-sklejanych są tak zwane **unormowane funkcje B-sklejane**, określone przez niemalejący ciąg **węzłów** $u_0, \ldots, u_N$. Funkcje te można zdefiniować za pomocą **wzoru Mansfielda-de Boora-Coxa**:

$$N_i^0(t) = \begin{cases} 1 & \text{dla } t \in [u_i, u_{i+1}), \\ 0 & \text{w przeciwnym razie,} \end{cases} \qquad i = 0, \ldots, N-1, \tag{B.1}$$

$$N_i^n(t) = \frac{t - u_i}{u_{i+n} - u_i} N_i^{n-1}(t) + \frac{u_{i+n+1} - t}{u_{i+n+1} - u_{i+1}} N_{i+1}^{n-1}(t), \quad n > 0,\ i = 0, \ldots, N-n-1. \tag{B.2}$$

Funkcja N_i^n jest jednoznacznie określona przez węzły $u_i, \ldots, u_{i+n+1}$ i przyjmuje niezerowe (dodatnie) wartości tylko w przedziale $[u_i, u_{i+n+1})$, co oznacza, że jeśli $u_i = u_{i+n+1}$, to jest to funkcja zerowa. Dlatego ciągi węzłów będące podstawą określenia funkcji B-sklejanych używanych do reprezentowania krzywych lub powierzchni stopnia n powinny spełniać warunek $u_i < u_{i+n+1}$ dla $i = 0, \ldots, N-n-1$. Jeśli $u_i = u_{i+n}$ lub $u_{i+1} = u_{i+n+1}$, to we wzorze (B.2) mamy ułamek z mianownikiem równym 0, ale jest on pomnożony przez funkcję zerową N_i^{n-1} albo N_{i+1}^{n-1}, wskutek czego odpowiedni składnik w tym wzorze jest równy 0. Funkcja N_i^n jest w każdym z przedziałów $[u_i, u_{i+1}), \ldots, [u_{i+n}, u_{i+n+1})$ wielomianem stopnia n. Jeśli $u_i \leqslant u_k \leqslant u_{i+n+1}$ oraz $u_{k-1} < u_k = \cdots = u_{k+r-1} < u_{k+r}$, to w węźle u_k funkcja ta jest ciągła razem z pochodnymi rzędu $1, \ldots, n-r$. Jeśli $u_{i+1} = \cdots = u_{i+n} < u_{i+n+1}$, to $N_i^n(u_{i+1}) = 1$.



Rysunek B.1. Krzywa B-sklejana stopnia 3, jej węzły i punkty kontrolne

Krzywa B-sklejana stopnia n jest określona przez ciąg węzłów[1] $u_0, \ldots, u_N$ oraz **punkty kontrolne** $\boldsymbol{d}_0, \ldots, \boldsymbol{d}_{N-n-1}$, wzorem

$$\boldsymbol{s}(t) = \sum_{i=0}^{N-n-1} \boldsymbol{d}_i N_i^n(t), \quad t \in [u_n, u_{N-n}). \tag{B.3}$$

Ciąg węzłów musi być dostatecznie długi, tj. musi być $N > 2n$ (i $u_n < u_{N-n}$), aby dziedzina parametryzacji $\boldsymbol{s}$ była przedziałem o długości większej niż 0.

Jeśli $u_1 = \cdots = u_n = 0$ i $u_{n+1} = \cdots = u_{2n} = 1$, to dla $i = 0, \ldots, n$ funkcja N_i^n w przedziale $[0,1)$ jest wielomianem Bernsteina B_i^n stopnia n (zobacz wzór (15.1)) i krzywa B-sklejana określona z takimi węzłami jest krzywą Béziera.

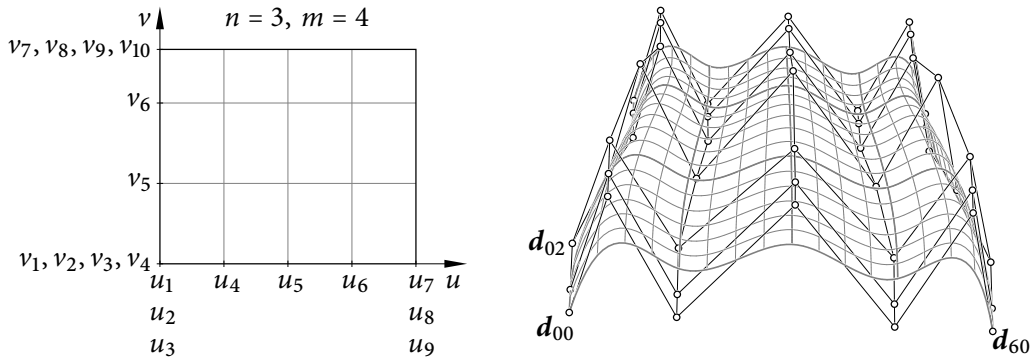
Tensorowy płat powierzchni B-sklejanej stopnia (n, m) jest określony wzorem

$$\boldsymbol{s}(u,v) = \sum_{i=0}^{N-n-1} \sum_{j=0}^{M-m-1} \boldsymbol{d}_{ij} N_i^n(u) N_j^m(v), \quad u \in [u_n, u_{N-n}),\ v \in [v_m, v_{M-m}). \tag{B.4}$$

We wzorze tym występują **punkty kontrolne płata** $\boldsymbol{d}_{ij}$ i dwie rodziny funkcji B-sklejanych, stopni n i m; stopnie te mogą być różne, ale niezależnie od stopni użyte do określenia funkcji ciągi węzłów, $u_0, \ldots, u_N$ oraz $v_0, \ldots, v_M$, mogą być różne (a wtedy rodziny funkcji N_i^n i N_j^m są różne, nawet jeśli $n = m$).

Suma funkcji B-sklejanych $N_0^n, \ldots, N_{N-n-1}^n$ w każdym punkcie przedziału $[u_n, u_{N-n})$ jest równa 1, skąd wynika, że aby poddać krzywą lub płat B-sklejany dowolnemu przekształceniu afinicznemu, wystarczy zastosować to przekształcenie do wszystkich punktów kontrolnych, podobnie jak w przypadku krzywych i płatów Béziera. Wartości funkcji B-sklejanych są nieujemne, dzięki czemu reprezentacja B-sklejana (tak jak reprezentacja Béziera) ma **własność otoczki wypukłej** — krzywa lub powierzchnia jest w całości położona w otoczce wypukłej zbioru swoich punktów kontrolnych. Zaletą krzywych i płatów B-sklejanych jest znacznie większa łatwość ich kształtowania: nawet wielka liczba punktów kontrolnych nie wymusza wysokiego stopnia wielomianów, a ponadto zmiana każdego punktu ma lokalny wpływ na

[1] Węzły u_0 i u_N, potrzebne do określenia funkcji N_0^n i N_{N-n-1}^n, nie wpływają na wartości tych funkcji w przedziale $[u_n, u_{N-n})$, a zatem nie mają wpływu na krzywą $\boldsymbol{s}$ i można je wybrać dowolnie.



Rysunek B.2. Płat powierzchni B-sklejanej stopnia $(3,4)$, jego węzły i punkty kontrolne

kształt krzywej lub powierzchni — zmieni się tylko jej fragment. Bardzo ważną własnością jest możliwość **wstawiania węzłów**; można dzięki niej ukształtować zgrubnie krzywą lub powierzchnię z niewieloma punktami kontrolnymi, a następnie wstawić dodatkowe węzły, aby otrzymać reprezentację tej samej krzywej lub powierzchni o większej liczbie punktów kontrolnych, co umożliwi cyzelowanie detali. Ale po szczegóły tych i innych własności odsyłam do literatury.

B.2. Algorytmy de Boora

Przedstawiona na listingu B.1 procedura oblicza wartości funkcji B-sklejanych stopnia n, określonych przez dany ciąg węzłów $u_0, \ldots, u_N$. Ściślej biorąc, na podstawie wzorów (B.1) i (B.2) procedura ta oblicza wartości wielomianów stopnia n opisujących funkcje B-sklejane $N^n_{k-n}, \ldots, N^n_k$, które w przedziale $[u_k, u_{k+1})$ przyjmują niezerowe wartości.

Parametr `bfv` procedury `EvaluateBSplinesf` jest tablicą o długości $n+1$, w której ma się znaleźć wynik, czyli wartości tych funkcji w punkcie t, podanym w parametrze `t`. Parametr `n` określa stopień n funkcji, parametr `knots` jest tablicą z węzłami, a parametr `k` określa przedział $[u_k, u_{k+1})$ zawierający punkt t, przy czym powinno być $t \in [u_n, u_{N-n}]$; należy zatem zapewnić, że $k \in \{n, \ldots, N-n-1\}$. Dla ustalonego ciągu węzłów i danej liczby t odpowiednią liczbę k można znaleźć różnymi sposobami. Jeśli węzły są równoodległe, tj. $u_i = u_0 + ih$ dla ustalonego $h > 0$ oraz $i = 1, \ldots, N$, to $k = \lfloor (t - u_0)/h \rfloor$. Jeśli węzły nie są równoodległe, to właściwy przedział można wyszukać metodą bisekcji. W pewnych sytuacjach przedział jest „znany z góry", na przykład w konstrukcji krzywych interpolacyjnych, gdy potrzebne są wartości funkcji w określających je węzłach (zobacz podrozdz. B.3).

Przekształcając wzory (B.1), (B.2), można otrzymać algorytm znajdowania punktu $\boldsymbol{s}(t)$ krzywej określonej wzorem (B.3) dla ustalonego t. Algorytm ten umożliwia rysowanie krzywych i płatów B-sklejanych, ale w tym celu potrzebna jest jego implementacja w języku GLSL. Razem z punktem $\boldsymbol{s}(t)$ trzeba obliczać wektor styczny (o kierunku pochodnej parametryzacji $\boldsymbol{s}$ w punkcie t), co umożliwi obliczanie wektora normalnego płata.

Listing B.1. Procedura obliczania wartości funkcji B-sklejanych

```C
void EvaluateBSplinesf ( float *bfv, int n, int k, const float *knots,
                         float t )
{
  int   i, j, l;
  float alpha, beta;

  l = k-n;
  bfv[n] = 1.0;
  for ( j = 1; j <= n; j++ ) {
    beta = (knots[k+1]-t)/(knots[k+1]-knots[k-j+1]);
    bfv[n-j] = beta*bfv[n-j+1];
    for ( i = k-j+1; i < k; i++ ) {
      alpha = 1.0-beta;
      beta = (knots[i+j+1]-t)/(knots[i+j+1]-knots[i+1]);
      bfv[i-l] = alpha*bfv[i-l] + beta*bfv[i+1-l];
    }
    bfv[n] *= 1.0-beta;
  }
} /*EvaluateBSplinesf*/
```

Niech $t \in [u_k, u_{k+1}) \subset [u_n, u_{N-n})$ i niech $\boldsymbol{d}_{k-n}^{(0)} = \boldsymbol{d}_{k-n}, \ldots, \boldsymbol{d}_k^{(0)} = \boldsymbol{d}_k$. **Algorytm de Boora** znajdowania punktu krzywej B-sklejanej polega na skonstruowaniu punktów

$$\begin{cases} \boldsymbol{d}_i^{(j)} = \left(1 - \alpha_i^{(j)}\right)\boldsymbol{d}_{i-1}^{(j-1)} + \alpha_i^{(j)}\boldsymbol{d}_i^{(j-1)}, \\ \alpha_i^{(j)} = \dfrac{t - u_i}{u_{i+n+1-j} - u_i}, \end{cases} \qquad i = k-n+j, \ldots, k, \quad j = 1, \ldots, n. \tag{B.5}$$

Ostatni obliczony punkt, $\boldsymbol{d}_k^{(n)}$, jest punktem $\boldsymbol{s}(t)$.

Można wykazać, że jeśli $n > 0$ i punkt $t \in [u_k, u_{k+1})$ nie jest węzłem lub jest węzłem o krotności mniejszej iż n (w takim węźle pochodna parametryzacji $\boldsymbol{s}$ może być nieciągła), to

$$\boldsymbol{s}'(t) = \frac{n}{u_{k+1} - u_k}\left(\boldsymbol{d}_k^{(n-1)} - \boldsymbol{d}_{k-1}^{(n-1)}\right).$$

Listing B.2 przedstawia deklaracje bloków magazynowych zawierających reprezentację płata B-sklejanego i procedury `BSCdeBoor3f` i `BSPdeBoor3f`. Są one częścią szadera rozdrabniania podobnego do szadera przedstawionego na listingu 15.4.

W bloku `BSPatch` znajdują się podstawowe dane opisujące płat. Pole `dim` zawiera liczbę d współrzędnych punktu, 2, 3 lub 4. W polach `n` i `m` są podane stopnie płata, a w polach `N` i `M` numery ostatnich węzłów w ciągach $u_0, \ldots, u_N$ oraz $v_0, \ldots, v_M$. Wartością pola `stride` jest odstęp między początkami kolejnych kolumn siatki kontrolnej płata w tablicy punktów kontrolnych. W polu `Colour` *można* podać kolor do użycia na obrazie płata[2]. Wartość pola

[2] Szader fragmentów może użyć tego koloru lub skorzystać z opisu materiału podanego w innym miejscu.

`BSPNormals` określa wybór wektora normalnego do użycia w modelu oświetlenia, a pole `tesslevel` zawiera parametr — stopień rozdrobnienia płata, który szader sterowania rozdrabnianiem powinien wpisać do tablic `gl_TessLevelOuter` i `gl_TessLevelInner`[3].

Tablica `uv` w bloku `BSKnots` zawiera węzły, tj. $N + M + 2$ liczby $u_0, \ldots, u_N, v_0, \ldots, v_M$ (w tej kolejności), a współrzędne punktów kontrolnych są podane w tablicy `cp` w bloku `BSCPoints` — ma w niej być $N - n$ kolumn, z których każda składa się z $M - m$ punktów, Jeśli między kolumnami nie ma przerw, to wartością pola `stride` ma być liczba $d(M - m)$.

Procedura `BSCdeBoor3f` oblicza punkty $\boldsymbol{d}_{k-1}^{(n-1)}$ i $\boldsymbol{d}_{k}^{(n-1)}$ dla krzywej B-sklejanej w przestrzeni trójwymiarowej. Parametr `n` określa stopień krzywej. Wartością parametru `t` jest liczba t. Parametr `k` jest indeksem do tablicy węzłów — jeśli $k < N$, to $t \in [u_k, u_{k+1})$, a w przeciwnym razie $t \in [v_l, v_{l+1})$, gdzie l jest wartością parametru `k` pomniejszoną o $N + 1$. W tablicy `d` są podane punkty $\boldsymbol{d}_{k-n}^{(0)}, \ldots, \boldsymbol{d}_{k}^{(n)}$; określają one pewien łuk wielomianowy krzywej $\boldsymbol{s}$. Obliczone punkty są przypisywane parametrom wyjściowym `p0` i `p1`, a parametr `a` służy do przekazania liczby $\alpha_k^{(n)}$. Na ich podstawie procedura `BSPdeBoor3f`, wykonując ostatni krok algorytmu de Boora, oblicza punkt $\boldsymbol{s}(t) = (1 - \alpha_k^{(n)})\boldsymbol{d}_{k-1}^{(n-1)} + \alpha_k^{(n)}\boldsymbol{d}_{k}^{(n-1)}$, oraz wektor $\boldsymbol{d}_{k}^{(n-1)} - \boldsymbol{d}_{k-1}^{(n-1)}$, mający ten sam kierunek i zwrot co wektor $\boldsymbol{s}'(t)$, ale na ogół inną długość, nieistotną dla obliczenia wektora normalnego płata tensorowego[4].

Procedura `BSPdeBoor3f` znajduje punkt $\boldsymbol{s}(u, v)$ płata B-sklejanego stopnia (n, m) i jego wektor normalny w tym punkcie. Parametry wejściowe tej procedury to: parametr u płata, numer węzła u_k, takiego że $u \in [u_k, u_{k+1})$, parametr v płata i powiększony o $N + 1$ numer węzła v_l, takiego że $v \in [v_l, v_{l+1})$. Parametry wyjściowe `pos` i `nv` wyprowadzają współrzędne jednorodne obliczonego punktu płata i wektora normalnego.

W pętli w liniach 32–37 procedura wybiera odpowiednie fragmenty kolumn siatki kontrolnej, po czym wywołuje procedurę `BSCdeBoor3f`. Punkty obliczone przez tę procedurę są wpisywane do tablic `q0` i `q1`. Otrzymane w ten sposób punkty w każdej z tych tablic są dalej przetwarzane jak punkty kontrolne krzywych B-sklejanych stopnia n; wynikiem obliczeń wykonywanych przez procedurę wywołaną w liniach 38 i 39 są punkty $\boldsymbol{p}_{00}$, $\boldsymbol{p}_{01}$, $\boldsymbol{p}_{10}$ i $\boldsymbol{p}_{11}$ (zobacz rys. B.3). Punkty te i liczby α_u i α_v (które w ostatnim, pominiętym kroku algorytmu de Boora są parametrami interpolacji punktów otrzymanych w kroku przedostatnim) umożliwiają obliczenie wektorów

$$\boldsymbol{r}_u = (1 - \alpha_v)(\boldsymbol{p}_{10} - \boldsymbol{p}_{00}) + \alpha_v(\boldsymbol{p}_{11} - \boldsymbol{p}_{01}), \quad \boldsymbol{r}_v = (1 - \alpha_u)(\boldsymbol{p}_{01} - \boldsymbol{p}_{00}) + \alpha_u(\boldsymbol{p}_{11} - \boldsymbol{p}_{10}),$$
$$\boldsymbol{n} = \boldsymbol{r}_u \wedge \boldsymbol{r}_v$$

i punktu

$$\boldsymbol{s}(u, v) = (1 - \alpha_u)\big((1 - \alpha_v)\boldsymbol{p}_{00} + \alpha_v\boldsymbol{p}_{10}\big) + \alpha_u\big((1 - \alpha_v)\boldsymbol{p}_{01} + \alpha_v\boldsymbol{p}_{11}\big).$$

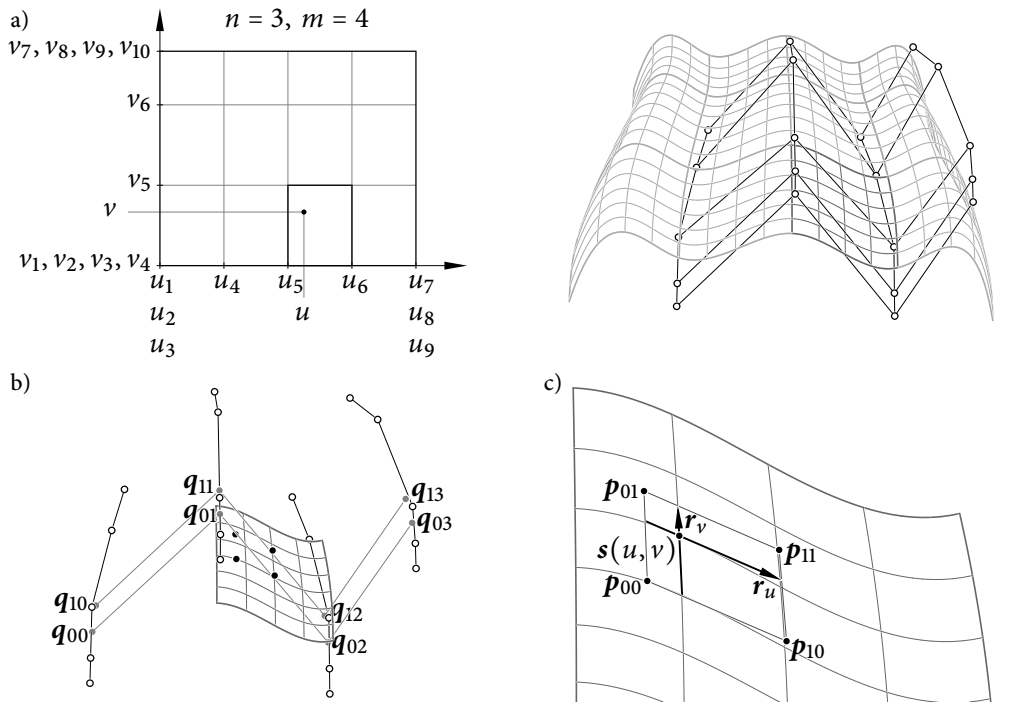
[3] Dla płatów B-sklejanych zasadne wydaje się wprowadzenie co najmniej dwóch parametrów rozdrabniania dziedziny, osobno wzdłuż każdej osi, a jeszcze lepszym rozwiązaniem byłoby adaptacyjne obliczanie parametrów rozdrabniania przez szader na podstawie kształtu siatki kontrolnej i wielkości obrazu płata.

[4] Długość wektora $\boldsymbol{s}'(t)$ jest istotna, jeśli na powierzchnię B-sklejaną ma być nałożona tekstura odkształceń określona w sposób opisany w rozdziale 21.

Listing B.2. Procedury BSCdeBoor3f i BSPdeBoor3f

GLSL

```
#define MAX_DEG 6

layout(std430,binding=0) buffer BSKnots { float uv[]; };
layout(std430,binding=1) buffer BSCPoints { float cp[]; };
layout(std430,binding=2) buffer BSPatch {
    int  dim, n, N, m, M, stride;
    vec3 Colour;
    bool BSPNormals;
    int  tesslevel;
  } bsp;

void BSCdeBoor3f ( int n, int k, vec3 d[MAX_DEG+1], float t,
                   out vec3 p0, out vec3 p1, out float a )
{
  int i, j;

  for ( j = 1; j < n; j++ )
    for ( i = 0; i <= n-j; i++ )
      d[i] = mix ( d[i], d[i+1], (t-uv[k-n+i+j])/(uv[k+1+i]-uv[k-n+i+j]) );
  a = (t-uv[k])/(uv[k+1]-uv[k]);
  p0 = d[0];  p1 = d[1];
} /*BSCdeBoor3f*/

void BSPdeBoor3f ( float u, int kk, float v, int ll,
                   out vec4 pos, out vec4 nv )
{
  vec3  p[MAX_DEG+1], q0[MAX_DEG+1], q1[MAX_DEG+1],
        p00, p01, p10, p11, a, b, ru, rv;
  float au, av;
  int   i, j, k, l;

  for ( i = 0, k = (kk-bsp.n)*bsp.stride;  i <= bsp.n;
        i++, k += bsp.stride ) {
    for ( j = 0, l = k+3*(ll-bsp.N-1-bsp.m);  j <= bsp.m;  j++, l += 3 )
      p[j] = vec3 ( cp[l], cp[l+1], cp[l+2] );
    BSCdeBoor3f ( bsp.m, ll, p, v, q0[i], q1[i], av );
  }
  BSCdeBoor3f ( bsp.n, kk, q0, u, p00, p10, au );
  BSCdeBoor3f ( bsp.n, kk, q1, u, p01, p11, au );
  ru = (b = mix ( p10, p11, av )) - (a = mix ( p00, p01, av ));
  rv = mix ( p01-p00, p11-p10, au );
  pos = vec4 ( mix ( a, b, au ), 1.0 );
  nv = vec4 ( cross ( ru, rv ), 0.0 );
} /*BSPdeBoor3f*/
```



Rysunek B.3. Punkty do końcowego obliczenia punktu płata i wektora normalnego

Listing B.3 przedstawia procedurę `main` szadera rozdrabniania dla programu rysowania płatów B-sklejanych. Zależnie od wartości pola `dim` w bloku `BSPatch` ma ona wywołać procedurę `BSPdeBoor3f` z listingu B.2 lub jedną z procedur `BSPdeBoor2f` albo `BSPdeBoor4f`, których napisanie pozostawiłem jako ćwiczenie. Przedtem jednak potrzebne są pewne przygotowania, niezależne od liczby współrzędnych punktów.

Szader rozdrabniania otrzymuje współrzędne punktu w dziedzinie płata w zmiennej wbudowanej `gl_TessCoord`. Ale dziedzina ta (dla algorytmu rozdrabniania wbudowanego w potok przetwarzania grafiki) jest kwadratem jednostkowym, podczas gdy płat B-sklejany ma dziedzinę $[u_n, u_{N-n}] \times [v_m, v_{M-m}]$, która może być dowolnym prostokątem (dołączając dwa „brakujące" odcinki brzegu, dostajemy prostokąt domknięty). Dlatego pierwszym krokiem do zrobienia jest odwzorowanie punktu podanego przez etap rozdrabniania w dziedzinę płata B-sklejanego. Otrzymamy w ten sposób parametry płata u i v.

Kolejną czynnością jest wyszukanie w obu ciągach węzłów właściwych miejsc, czyli znalezienie liczb k i l, takich że $u \in [u_k, u_{k+1})$ i $v \in [v_l, v_{l+1})$, przy czym jeśli $u = u_{N-n}$, to trzeba przyjąć $k = N - n - 1$, a jeśli $v = v_{M-m}$, to ma być $l = M - m - 1$. Tym zajmuje się procedura `FindKnotInterval`, której parametry `k0` i `kN` wybierają ciąg węzłów do przeszukania — $u_0, \ldots, u_N$, jeśli `k0` = 0, `kN` = N, albo $v_0, \ldots, v_M$, jeśli `k0` = $N + 1$, `kN` = $N + M + 1$. Parametr `n` zawęża przedział poszukiwań liczby t do $[u_n, u_{N-n}]$ albo $[v_m, v_{M-m}]$. Procedura realizuje algorytm wyszukiwania binarnego.

Listing B.3. Procedura `main` szadera rozdrabniania

GLSL

```
#version 450 core

layout(quads,equal_spacing,cw) in;

out GVertex { .... } Out;

uniform TransBlock { .... } trb;

int FindKnotInterval ( int k0, int kN, int n, float t )
{
  int i, j, k;

  for ( i = k0+n, j = kN-n;  j-i > 1; ) {
    k = i + (j-i)/2;
    if ( t >= uv[k] ) i = k; else j = k;
  }
  return i;
} /*FindKnotInterval*/

void main ( void )
{
  float u, v;
  int   k, l, m0, m1;
  vec4  pos, nv;

  u = uv[bsp.n] + gl_TessCoord.x*(uv[bsp.N-bsp.n] - uv[bsp.n]);
  m0 = bsp.N+1;  m1 = bsp.N+1+bsp.M;
  v = uv[m0+bsp.m] + gl_TessCoord.y*(uv[m1-bsp.m] - uv[m0+bsp.m]);
  k = FindKnotInterval ( 0, bsp.N, bsp.n, u );
  l = FindKnotInterval ( m0, m1, bsp.m, v );
  switch ( bsp.dim ) {
 case 2: BSPdeBoor2f ( u, k, v, l, pos, nv );  break;
 case 3: BSPdeBoor3f ( u, k, v, l, pos, nv );  break;
 case 4: BSPdeBoor4f ( u, k, v, l, pos, nv );  break;
default: pos = nv = vec4 ( 0.0 );
  }
  pos = trb.mm * pos;
  gl_Position = trb.vpm * pos;
  Out.Position = pos.xyz;
  if ( !bsp.BSPNormals || dot ( nv, nv ) < 1.0e-10 )
    Out.Normal = vec3 ( 0.0 );
  else
    Out.Normal = normalize ( (trb.mmti*nv).xyz );
  Out.Colour = bsp.Colour;
} /*main*/
```

Uwaga: Szader opisany w pierwszym wydaniu miał zadeklarowane osobne bloki magazynowe dla obu ciągów węzłów. W konsekwencji do wyszukiwania przedziału w każdym z tych ciągów potrzebna była inna procedura, ponieważ w języku GLSL nie ma wskaźników — tablice będące parametrami muszą mieć znaną długość i są w całości kopiowane (a więc również nie powinny być długie). Umieszczenie obu ciągów w jednej tablicy umożliwiło skrócenie kodu i uniknięcie kopiowania węzłów do tablic przekazywanych następnie jako parametry procedurom `BSPdeBoor3f` i `BSCdeBoor3f`.

Pisząc procedurę `main`, przyjąłem, że blok zmiennych jednolitych z macierzami przekształceń (`TransBlock`) i blok wyjściowy interfejsu (`GVertex`, o lokalnej nazwie `Out`), przez który wyniki rozdrabniania trafią do szadera geometrii, są identyczne jak na listingu 15.4; oczywiście, trzeba to zmienić, jeśli na płaty ma być nałożona tekstura lub na końcowym obrazie mają być cienie. Przedstawiony tu szader rozdrabniania dokonuje przejścia do układu kostki standardowej, ale oblicza i wyprowadza także punkt i wektor normalny w układzie współrzędnych świata. Pole (typu `bool`) `BSPNormals` bloku `BSPatch` pełni rolę analogiczną do pola `BezNormals` bloku `BezPatch` w drugiej aplikacji.

Listing B.4 przedstawia procedury w języku C umożliwiające umieszczenie w pamięci GPU i rysowanie płatów B-sklejanych. Pierwsza procedura musi być wywołana po skompilowaniu i złączeniu dowolnego programu szaderów zawierającego deklaracje bloków magazynowych pokazanych na listingu B.2 (np. dowolngo programu rysującego takie płaty). Procedura odczytuje z programu numery punktów dowiązania bloków i przesunięcia pól w bloku `BSPatch`.

Procedura `EnterBSplinePatch` rezerwuje bufory i przesyła do nich reprezentację płata, nadając wartości domyślne polom `BSPNormals` i `tesslevel`. Nie zamieściłem listingu procedur nadających tym parametrom wartości w trakcie działania aplikacji, uznając, że nie ma w nich niczego wymagającego szczegółowych objaśnień. Obiekt tablicy wierzchołków, tworzony i zapisywany w liniach 79–84, służy do rysowania punktów kontrolnych przez procedurę `DrawBSplineCPoints`,

Przed wywołaniem procedury rysującej płat, `DrawBSplinePatch`, albo punkty kontrolne, `DrawBSplineCPoints`, trzeba wybrać odpowiedni program szaderów za pomocą procedury `glUseProgram`.

Listing B.4. Procedury obsługi płatów B-sklejanych

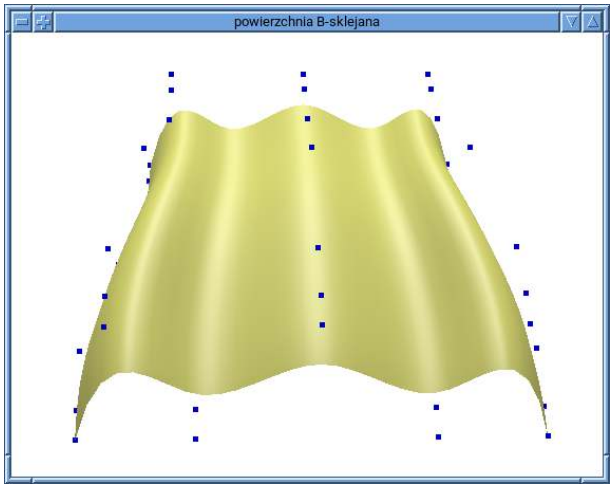
C

```
1: #define MAX_BSPATCH_DEG 6
2:
3: typedef struct BSPatchObjf {
4:     GLint  udeg, lknu, vdeg, lknv, dim, stride;
5:     GLuint buf[3];
6:     GLuint vao;
7:   } BSPatchObjf;
8:
9: #define NBSPLINEPATCHOFFS 9
```

```
10:
11: static GLuint bspbbp = GL_INVALID_INDEX, bsknbbp = GL_INVALID_INDEX,
12:               bscpbbp = GL_INVALID_INDEX;
13: static GLint  bspbsize, bspbofs[NBSPLINEPATCHOFFS];
14:
15: static const GLchar *BSKNNames[] = { "BSKnots" };
16: static const GLchar *BSCPNames[] = { "BSCPoints" };
17: static const GLchar *BSPNames[] =
18:   { "BSPatch", "BSPatch.dim", "BSPatch.n", "BSPatch.N", "BSPatch.m",
19:     "BSPatch.M", "BSPatch.stride", "BSPatch.Colour", "BSPatch.BSPNormals",
20:     "BSPatch.tesslevel" };
21:
22: void GetAccessToBSPatchStorageBlocks ( GLuint program_id )
23: {
24:   GLint size, ofs;
25:
26:   if ( bspbbp == GL_INVALID_INDEX )
27:     GetAccessToStorageBlock ( program_id, NBSPLINEPATCHOFFS, &BSPNames[0],
28:                               &bspbsize, bspbofs, &bspbbp );
29:   if ( bsknbbp == GL_INVALID_INDEX )
30:     GetAccessToStorageBlock ( program_id, 0, &BSKNNames[0],
31:                               &size, &ofs, &bsknbbp );
32:   if ( bscpbbp == GL_INVALID_INDEX )
33:     GetAccessToStorageBlock ( program_id, 0, &BSCPNames[0],
34:                               &size, &ofs, &bscpbbp );
35: } /*GetAccessToBSPatchStorageBlocks*/
36:
37: #define SSB GL_SHADER_STORAGE_BUFFER
38:
39: BSPatchObjf *EnterBSplinePatch (
40:                       GLint udeg, GLint lknu, const GLfloat *knotsu,
41:                       GLint vdeg, GLint lknv, const GLfloat *knotsv,
42:                       GLint dim, GLint stride, const GLfloat *cp,
43:                       const GLfloat *colour )
44: {
45:   BSPatchObjf *bsp;
46:   GLint one = GL_TRUE, ten = 10;
47:
48:   if ( dim < 2 || dim > 4 || udeg < 1 || udeg > MAX_BSPATCH_DEG ||
49:        vdeg < 1 || vdeg > MAX_BSPATCH_DEG || lknu <= 2*udeg ||
50:        lknv <= 2*vdeg )
51:     return NULL;
52:   bsp = malloc ( sizeof(BSPatchObjf) );
53:   if ( bsp ) {
54:     memset ( bsp, 0, sizeof(BSPatchObjf) );
55:     bsp->udeg = udeg;  bsp->lknu = lknu;
56:     bsp->vdeg = vdeg;  bsp->lknv = lknv;
```

```
57:     bsp->dim = dim;    bsp->stride = stride;
58:     glGenBuffers ( 3, bsp->buf );
59:     glBindBuffer ( SSB, bsp->buf[2] );
60:     glBufferData ( SSB, bspbsize, NULL, GL_DYNAMIC_DRAW );
61:     glBufferSubData ( SSB, bspbofs[0], sizeof(GLint), &dim );
62:     glBufferSubData ( SSB, bspbofs[1], sizeof(GLint), &udeg );
63:     glBufferSubData ( SSB, bspbofs[2], sizeof(GLint), &lknu );
64:     glBufferSubData ( SSB, bspbofs[3], sizeof(GLint), &vdeg );
65:     glBufferSubData ( SSB, bspbofs[4], sizeof(GLint), &lknv );
66:     glBufferSubData ( SSB, bspbofs[5], sizeof(GLint), &stride );
67:     glBufferSubData ( SSB, bspbofs[6], 3*sizeof(GLfloat), colour );
68:     glBufferSubData ( SSB, bspbofs[7], sizeof(GLint), &one );
69:     glBufferSubData ( SSB, bspbofs[8], sizeof(GLint), &ten );
70:     glBindBuffer ( SSB, bsp->buf[0] );
71:     glBufferData ( SSB, (lknu+lknv+2)*sizeof(GLfloat), NULL,
72:                    GL_DYNAMIC_DRAW );
73:     glBufferSubData ( SSB, 0, (lknu+1)*sizeof(GLfloat), knotsu );
74:     glBufferSubData ( SSB, (lknu+1)*sizeof(GLfloat),
75:                       (lknv+1)*sizeof(GLfloat), knotsv );
76:     glBindBuffer ( SSB, bsp->buf[1] );
77:     glBufferData ( SSB, stride*(lknu-udeg)*sizeof(GLfloat), cp,
78:                    GL_DYNAMIC_DRAW );
79:     glGenVertexArrays ( 1, &bsp->vao );
80:     glBindVertexArray ( bsp->vao );
81:     glBindBuffer ( GL_ARRAY_BUFFER, bsp->buf[1] );
82:     glEnableVertexAttribArray ( 0 );
83:     glVertexAttribPointer ( 0, dim, GL_FLOAT, GL_FALSE,
84:                             dim*sizeof(GLfloat), (GLvoid*)0 );
85:     glBindVertexArray ( 0 );
86:     ExitIfGLError ( "EnterBSplinePatch" );
87:   }
88:   return bsp;
89: } /*EnterBSplinePatch*/
90:
91: void DrawBSplinePatch ( BSPatchObjf *bsp )
92: {
93:   if ( bsp ) {
94:     glBindBufferBase ( SSB, bsknbbp, bsp->buf[0] );
95:     glBindBufferBase ( SSB, bscpbbp, bsp->buf[1] );
96:     glBindBufferBase ( SSB, bspbbp, bsp->buf[2] );
97:     glBindVertexArray ( empty_vao );
98:     glPatchParameteri ( GL_PATCH_VERTICES, 1 );
99:     glDrawArrays ( GL_PATCHES, 0, 1 );
100:    ExitIfGLError ( "DrawBSplinePatch" );
101:  }
102: } /*DrawBSplinePatch*/
103:
```

```
104: void DrawBSplineCPoints ( BSPatchObjf *bsp )
105: {
106:   if ( bsp ) {
107:     glBindBufferBase ( SSB, bsknbbp, bsp->buf[0] );
108:     glBindBufferBase ( SSB, bscpbbp, bsp->buf[1] );
109:     glBindBufferBase ( SSB, bspbbp, bsp->buf[2] );
110:     glBindVertexArray ( bsp->vao );
111:     glPointSize ( 5.0 );
112:     glDrawArrays ( GL_POINTS, 0,
113:                    (bsp->lknu-bsp->udeg)*(bsp->lknv-bsp->vdeg) );
114:     glBindVertexArray ( 0 );
115:     ExitIfGLError ( "DrawBSplineCPoints" );
116:   }
117: } /*DrawBSplineCPoints*/
```



Rysunek B.4. Obraz płata B-sklejanego w oknie aplikacji OpenGL-a

W pierwszym wydaniu książki zaproponowałem ćwiczenie — napisanie i uruchomienie aplikacji rysującej płat B-sklejany przy użyciu opisanych tu algorytmów. W tym wydaniu proponuję ćwiczenie polegające na oprogramowaniu możliwości reprezentowania w pamięci GPU i jednoczesnego rysowania wielu płatów B-sklejanych, podobnie jak płatów Béziera w drugiej aplikacji. Powinno być przy tym możliwe wprowadzenie dodatkowej tablicy indeksów punktów kontrolnych, aby dany punkt mógł być wspólny dla wielu płatów, co ułatwiłoby m.in. sklejanie brzegów takich płatów. Dodatkową atrakcją może być umożliwienie definiowania poszczególnych płatów z różnymi ciągami węzłów (umieszczonymi w tej samej tablicy jeden za drugim). Blok magazynowy `BSPatch` w tym przypadku powinien zawierać tablicę struktur opisujących poszczególne płaty.

B.3. B-sklejane krzywe interpolacyjne

Oprócz cieniowania są dwa główne zastosowania interpolacji w grafice. Po pierwsze, mając dane punkty na płaszczyźnie lub w przestrzeni, możemy skonstruować gładką krzywą przechodzącą przez te punkty (i ewentualnie użyć tej konstrukcji do otrzymania powierzchni, na której leży siatka danych punktów). Po drugie, mając wartości parametrów artykulacji w pewnych chwilach, możemy znaleźć funkcje, których argumentem jest czas, przyjmujące w tych chwilach zadane wartości. Funkcje te umożliwią takie animowanie łańcucha kinematycznego, aby przywiązane do niego obiekty, poruszając się, przechodziły przez zadane położenia.

W wielu takich zastosowaniach użyteczne okazują się funkcje sklejane trzeciego stopnia (tzw. kubiczne), tj. funkcje opisane w sąsiadujących przedziałach przez wielomiany trzeciego stopnia. Używane w praktyce reprezentacje takich funkcji są różne, ale zawsze można je reprezentować za pomocą odpowiednio dobranych funkcji B-sklejanych, zatem przedstawię przykładową konstrukcję kubicznych B-sklejanych krzywych interpolacyjnych.

Niech $t_0, \ldots, t_M$ oznacza rosnący ciąg liczb zwanych **węzłami interpolacyjnymi**. Mamy też dane liczby $p_0, \ldots, p_M$ (lub punkty $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_M$) i chcemy znaleźć taką funkcję sklejaną s, aby było $s(t_i) = p_i$ (lub taką parametryzację sklejaną $\boldsymbol{s}$, aby było $\boldsymbol{s}(t_i) = \boldsymbol{p}_i$) dla $i = 0, \ldots, M$.

Aby określić kubiczne funkcje B-sklejane, trzeba wybrać ciąg węzłów $u_0, \ldots, u_N$; przyczyną częstej konfuzji jest nazwanie węzłami dwóch *różnych* pojęć. Liczby $u_0, \ldots, u_N$ są **węzłami funkcji sklejanych**, czyli końcami przedziałów, w których funkcje te są wielomianami[5]. Natomiast węzły interpolacyjne są punktami, w których są zadawane wartości funkcji. Aby otrzymać zadanie dobrze określone, przyjmiemy, że $t_0 = u_3$ i $t_M = u_{N-3}$; poszukiwana funkcja $s(t) = \sum_{i=0}^{N-4} d_i N_i^3(t)$ będzie zatem określona w przedziale $[t_0, t_M] = [u_3, u_{N-3}]$. Trzeba znaleźć współczynniki $d_0, \ldots, d_{N-4}$ spełniające równania

$$s(t_i) = p_i, \quad i = 0, \ldots, M.$$

Przyjmiemy $N = M + 6$ i $u_{i+3} = t_i$ dla $i = 0, \ldots, M$, co oznacza, że węzły interpolacyjne będą również węzłami funkcji sklejanych[6]. W węźle $u_{i+3} = t_i$ tylko funkcje N_i^3, N_{i+1}^3 i N_{i+2}^3 przyjmują niezerowe wartości, które możemy obliczyć za pomocą procedury z listingu B.1. Stąd równania interpolacji mają postać

$$N_i^3(t_i)d_i + N_{i+1}^3(t_i)d_{i+1} + N_{i+2}^3(t_i)d_{i+2} = p_i.$$

W każdym z nich występują tylko trzy niewiadome, co ogromnie ułatwia rozwiązywanie. Dla uproszczenia warto też przyjąć $u_1 = u_2 = u_3$ oraz $u_{N-3} = u_{N-2} = u_{N-1}$, ponieważ wtedy $N_0^3(t_0) = 1$, $N_1^3(t_0) = N_2^3(t_0) = 0$ oraz $N_{N-6}^3(t_M) = N_{N-5}^3(t_M) = 0$, $N_{N-4}^3(t_M) = 1$, skąd natychmiast wynika, że $d_0 = p_0$ i $d_{N-4} = p_M$.

Mamy zatem wartości funkcji zadane w $M+1 = N-5$ punktach, podczas gdy kubicznych funkcji B-sklejanych określonych przez ciąg $u_0, \ldots, u_N$ jest $N - 3$, czyli o dwie więcej. To

[5] W tych punktach wielomiany są „sklejone".

[6] Tym większa bywa wspomniana konfuzja. W ogólności węzły interpolacyjne nie muszą być węzłami funkcji sklejanych.

oznacza konieczność podania jeszcze dwóch warunków (dwóch dodatkowych równań), aby powstał układ o jednoznacznym rozwiązaniu. Zazwyczaj warunki te narzucają pewne własności funkcji s w pobliżu końców przedziału $[u_3, u_{N-3}]$ i dlatego są nazywane **warunkami brzegowymi**.

Warunki brzegowe opisane niżej to równości $s''(t_0) = 0$ i $s''(t_M) = 0$; spełniająca je funkcja s jest nazywana **naturalną kubiczną funkcją sklejaną**. Ściślej biorąc, pochodne drugiego rzędu funkcji B-sklejanych N_0^3, N_1^3 i N_2^3 w węźle $u_1 = u_2 = u_3 = t_0$, a także funkcji N_{N-6}^3, N_{N-5}^3 i N_{N-4}^3 w węźle $u_{N-3} = u_{N-2} = u_{N-1} = t_M$ są nieokreślone. Ale wielomiany opisujące w przedziałach $[u_3, u_4)$ i $[u_{N-4}, u_{N-3})$ te funkcje, a także funkcję s, którą chcemy otrzymać, mają wszystkie pochodne. Dlatego warunki brzegowe nałożymy na wielomiany opisujące funkcję s. Jeśli symbolem $P_{i,k}$ oznaczymy wielomian opisujący funkcję N_i^3 w przedziale $[u_k, u_{k+1})$, to rozważane tu warunki brzegowe mają postać

$$P''_{0,3}(t_0)d_0 + P''_{1,3}(t_0)d_1 + P''_{2,3}(t_0)d_2 = 0, \tag{B.6}$$

$$P''_{N-6,N-4}(t_M)d_{N-6} + P''_{N-5,N-4}(t_M)d_{N-5} + P''_{N-4,N-4}(t_M)d_{N-4} = 0. \tag{B.7}$$

Na podstawie wzoru (którego wyprowadzenie można znaleźć w [41])

$$N_i^{n\prime}(t) = \frac{n}{u_{i+n} - u_i} N_i^{n-1}(t) - \frac{n}{u_{i+n+1} - u_{i+1}} N_{i+1}^{n-1}(t) \tag{B.8}$$

i wzorów (B.1) i (B.2), w wyniku dość żmudnych rachunków[7], można obliczyć

$$\begin{aligned}
P''_{0,3}(t_0) &= \frac{6}{(u_4 - u_1)(u_4 - u_2)}, \\
P''_{1,3}(t_0) &= \frac{-6(u_4 - u_1 + u_5 - u_2)}{(u_4 - u_1)(u_4 - u_2)(u_5 - u_2)}, \\
P''_{2,3}(t_0) &= \frac{6}{(u_4 - u_2)(u_5 - u_2)}, \\
P''_{N-6,N-4}(t_M) &= \frac{6}{(u_{N-2} - u_{N-5})(u_{N-2} - u_{N-4})}, \\
P''_{N-5,N-4}(t_M) &= \frac{-6(u_{N-2} - u_{N-5} + u_{N-1} - u_{N-4})}{(u_{N-2} - u_{N-5})(u_{N-2} - u_{N-4})(u_{N-1} - u_{N-4})}, \\
P''_{N-4,N-4}(t_M) &= \frac{6}{(u_{N-2} - u_{N-4})(u_{N-1} - u_{N-4})}.
\end{aligned}$$

Mamy zatem układ $M+3$ równań liniowych z $M+3$ niewiadomymi, przy czym w pierwszym i ostatnim równaniu występuje tylko jedna niewiadoma (mamy $d_0 = p_0$ i $d_{N-4} = p_M$), a jeśli warunki brzegowe przyjmiemy za drugie i przedostatnie równanie, to w każdym równaniu oprócz pierwszego i ostatniego będą tylko trzy kolejne niewiadome, d_i, d_{i+1} i d_{i+2}. Macierz układu równań, która ma niezerowe współczynniki tylko na diagonali i na miejscach

[7] Przyznam się: użyłem pakietu do obliczeń symbolicznych i nie zamierzam mieć wyrzutów.

sąsiadujących z nią, to tak zwana **macierz trójdiagonalna**. Układ równań z taką macierzą $n \times n$ można rozwiązać kosztem rzędu n.[8]

Na listingu B.5 zamieściłem dwie procedury, z których pierwsza znajduje trójkątne czynniki (dolny L i górny U) rozkładu macierzy trójdiagonalnej A, a ściślej, macierzy PA, powstałej z A przez przestawianie wierszy, jeśli jest taka potrzeba[9]. Liczba n wierszy i kolumn macierzy (która musi być większa niż 2) jest wartością parametru `n`. Współczynniki macierzy są dane w trzech tablicach. Tablica `b` zawiera współczynniki na diagonali, a w tablicach `a` i `c` trzeba podać odpowiednio ich sąsiadów z lewej i prawej strony (elementy `a[0]` i `c[n-1]` są nieużywane). Pomocnicza tablica `d` (o długości $n-2$) jest potrzebna dlatego, że wskutek przestawiania wierszy powstaje macierz trójkątna U, która w każdym wierszu oprócz dwóch ostatnich może mieć *dwa* niezerowe współczynniki na prawo od diagonali.

Dane w tablicach współczynniki macierzy A zostają zastąpione przez współczynniki macierzy L i U. W tablicy `a` procedura `M3diagLUDecompf` zapamiętuje współczynniki pod diagonalą macierzy L (której współczynniki na diagonali są równe 1), a w tablicach `b`, `c` i `d` są zapisywane współczynniki macierzy U na jej diagonali i obok. W tablicy `p` jest umieszczana reprezentacja macierzy permutacji P; na miejscu i-tym w tej tablicy zostaje wstawiona jedynka, jeśli i-ty wiersz został przestawiony z i + pierwszym (tylko takie przestawienia są wykonywane), albo zero, jeśli przestawienia nie było.

Procedura `M3diagLUSolvef`, korzystając z czynników rozkładu macierzy A znalezionych przez procedurę `M3diagLUDecompf`, rozwiązuje układ równań liniowych $AX = B$; macierz B ma w ogólności m kolumn i tyle samo kolumn ma rozwiązanie X. Współczynniki tych macierzy są zapisane w jednowymiarowej tablicy `e` (o długości mn), *wierszami*. W miejscu danych współczynników macierzy B procedura pozostawia obliczone rozwiązanie.

Opisanych wyżej procedur użyjemy do konstruowania interpolacyjnych kubicznych funkcji lub krzywych sklejanych. Konstrukcję przeprowadza procedura `ConstructCubicInterpBSplinef` pokazana na listingu B.6. Jej parametr `N` jest wskaźnikiem zmiennej, której ma być przypisany numer N ostatniego węzła w ciągu $u_0, \ldots, u_N$, który zostanie wpisany do tablicy `knots`. W tablicy `cpoints` znajdą się obliczone współczynniki $d_0, \ldots, d_{N-4}$ lub współrzędne punktów kontrolnych $\boldsymbol{d}_0, \ldots, \boldsymbol{d}_{N-4}$; liczba współrzędnych każdego punktu (czyli wymiar przestrzeni, w której znajdują się te punkty i krzywa) jest wartością parametru `dim` (jeśli ma on wartość 1, to konstruujemy funkcję skalarną). Parametr `M` określa liczbę węzłów interpolacyjnych (jest ich $M+1$, przy czym musi być $M > 2$), które należy podać w tablicy `ikn`, a w tablicy `p` mają być podane wartości funkcji s lub współrzędne punktów, przez które ma przechodzić krzywa $\boldsymbol{s}$.

W linii `9` procedura rezerwuje miejsce na współczynniki macierzy układu równań i tablice pomocnicze do pomieszczenia czynników jej rozkładu. W liniach `13–15` powstaje ciąg węzłów funkcji sklejanej, który jest kopią ciągu węzłów interpolacyjnych z dołożonymi na początku i końcu trzema dodatkowymi „egzemplarzami" pierwszego i ostatniego węzła.

[8] I nie należy używać do tego uniwersalnych procedur rozwiązujących układ równań liniowych kosztem rzędu n^3.

[9] Algorytm realizowany przez tę procedurę to oczywiście metoda eliminacji Gaussa z wyborem elementu głównego, zaimplementowana wcześniej (dla macierzy o innej postaci) w procedurze `M4x4LUDecompf`.

Listing B.5. Procedury rozwiązywania układów równań z macierzą trójdiagonalną

C

```
char M3diagLUDecompf ( int n, float *a, float *b, float *c, float *d,
                       char *p )
{
  int   i;
  float l;
#define SWAP(x,y) { l = x;  x = y;  y = l; }

  memset ( d, 0, (n-2)*sizeof(float) );
  for ( i = 0; i < n-2; i++ ) {
    if ( (p[i] = fabs(a[i+1]) > fabs(b[i])) ) {
      SWAP ( a[i+1], b[i] )  SWAP ( b[i+1], c[i] )
      d[i] = c[i+1];  c[i+1] = 0.0;
    }
    if ( b[i] == 0.0 ) return false;  /* przerwij, jeśli macierz osobliwa */
    a[i+1] = l = a[i+1]/b[i];
    b[i+1] -= l*c[i];  c[i+1] -= l*d[i];
  }
  if ( (p[n-2] = fabs(a[n-1]) > fabs(b[n-2])) )
    { SWAP ( a[n-1], b[n-2] )  SWAP ( b[n-1], c[n-2] ) }
  if ( b[n-2] == 0.0 ) return false;
  a[n-1] = l = a[n-1]/b[n-2];
  b[n-1] -= l*c[n-2];
  return b[n-1] != 0.0;  /* jeśli wszystko OK, to przekaż true */
} /*M3diagLUDecompf*/

void M3diagLUSolvef ( int n, float *a, float *b, float *c, float *d,
                      char *p, int m, float *e )
{
  int   i, j, k;
  float l, s, t;

  for ( i = k = 0;  i < n-1;  i++, k += m ) {
    if ( p[i] )  /* przestaw wiersze prawej strony */
      for ( j = 0; j < m; j++ ) SWAP ( e[k+j], e[k+m+j] )
    for ( l = a[i+1], j = 0;  j < m;  j++ ) e[k+m+j] -= l*e[k+j];
  }
  for ( l = b[n-1], j = 0, k = (n-1)*m;  j < m;  j++ )
    e[k+j] /= l;
  for ( l = b[n-2], s = c[n-2], j = 0, k = (n-2)*m;  j < m;  j++ )
    e[k+j] = (e[k+j] - s*e[k+m+j])/l;
  for ( i = n-3, k = (n-3)*m;  i >= 0;  i--, k -= m )
    for ( l = b[i], s = c[i], t = d[i], j = 0;  j < m;  j++ )
      e[k+j] = (e[k+j] - s*e[k+m+j] - t*e[k+m+m+j])/l;
#undef SWAP
} /*M3diagLUSolvef*/
```

Listing B.6. Procedura konstrukcji sklejanych funkcji i krzywych interpolacyjnych

C

```
1: char ConstructCubicInterpBSplinef ( int *N, float *knots, float *cp,
2:                                     int M, float *ikn, int dim, float *p )
3: {
4:   int   i, lkn;
5:   float *a, *b, *c, *d, bfv[4], t0, t1;
6:   char  *permut;
7:
8:   *N = lkn = M+6;
9:   if ( !(a = malloc ( (4*(lkn-3)-3)*sizeof(float)+(lkn-4)*sizeof(char) )) )
10:    return false;
11:  b = &a[lkn-3];  c = &b[lkn-3];  d = &c[lkn-4];  permut = (char*)&d[lkn-5];
12:        /* utwórz ciąg węzłów funkcji sklejanej */
13:  knots[0] = knots[1] = knots[2] = ikn[0];
14:  memcpy ( &knots[3], ikn, (M+1)*sizeof(float) );
15:  knots[lkn-2] = knots[lkn-1] = knots[lkn] = ikn[M];
16:        /* utwórz macierz układu */
17:  b[0] = 1.0;  c[0] = 0.0;
18:  t0 = knots[4]-knots[1];  t1 = knots[5]-knots[2];
19:  a[1] = t1;  b[1] = -(t0+t1);  c[1] = t0;
20:  for ( i = 2; i <= M; i++ ) {
21:    EvaluateBSplinesf ( bfv, 3, i+2, knots, knots[i+2] );
22:    a[i] = bfv[0];  b[i] = bfv[1];  c[i] = bfv[2];
23:  }
24:  t0 = knots[lkn-2]-knots[lkn-5];  t1 = knots[lkn-1]-knots[lkn-4];
25:  a[lkn-3] = t1;  b[lkn-3] = -(t0+t1);  c[lkn-3] = t0;
26:  a[lkn-2] = 0.0;  b[lkn-2] = 1.0;
27:        /* utwórz prawą stronę */
28:  memcpy ( cp, p, dim*sizeof(float) );
29:  memset ( &cp[dim], 0, dim*sizeof(float) );
30:  memcpy ( &cp[dim+dim], &p[dim], (M-1)*dim*sizeof(float) );
31:  memset ( &cp[(M+1)*dim], 0, dim*sizeof(float) );
32:  memcpy ( &cp[(M+2)*dim], &p[dim*M], dim*sizeof(float) );
33:        /* rozwiąż układ równań */
34:  if ( !M3diagLUDecompf ( M+3, a, b, c, d, permut ) ) {
35:    free ( a );
36:    return false;
37:  }
38:  M3diagLUSolvef ( M+3, a, b, c, d, permut, dim, cp );
39:  free ( a );
40:  return true;
41: } /*ConstructCubicInterpBSplinef*/
```

W liniach 17–26 obliczane są współczynniki trójdiagonalnej macierzy układu równań. Współczynniki we wszystkich wierszach oprócz pierwszych i ostatnich dwóch są wartościami funkcji B-sklejanych obliczanymi przez procedurę `EvaluateBSplinesf` z lis-

tingu B.1. W pierwszym i ostatnim równaniu występują współczynniki 0 i 1, natomiast równanie drugie i przedostatnie to warunki brzegowe (B.6) i (B.7). Ich strony pomnożyłem przez $(u_4 - u_1)(u_4 - u_2)(u_5 - u_2)/6$ i $(u_{N-2} - u_{N-5})(u_{N-2} - u_{N-4})(u_{N-1} - u_{N-4})/6$, co uprościło wzory zaprogramowane w liniach 18–19 i 24–25.

W liniach 28–32 powstaje macierz będąca prawą stroną układu równań; jej drugi i przedostatni wiersz zawierają zera (pochodna drugiego rzędu odpowiednich wielomianów ma być zerem), a pozostałe wiersze są wektorami współrzędnych punktów danych. Po rozłożeniu macierzy na czynniki trójkątne jest wywoływana procedura `M3diagLUSolvef`, która do tablicy zawierającej początkowo macierz prawej strony wpisuje rozwiązanie — punkty kontrolne interpolacyjnej krzywej sklejanej.

Do obliczania punktów krzywej, dla danych wartości parametru, najprościej jest używać algorytmu de Boora, którego implementacja w języku GLSL jest pokazana na listingu B.2. Jeśli współrzędne punktów krzywej są parametrami artykulacji potrzebnymi do animowania łańcucha kinematycznego, to obliczanie tych punktów powinna wykonywać CPU. Napisanie odpowiedniej procedury w języku C jest prostym ćwiczeniem, które polecam.

W bardziej zaawansowanej animacji oprócz wartości funkcji może być potrzebne zadawanie wartości pochodnych w węzłach interpolacyjnych. Aby takie zadanie interpolacji było dobrze określone, węzły funkcji sklejanej, w których są podane dwa warunki (wartość i pochodna), muszą być *podwójne*. Kubiczne funkcje B-sklejane w węzłach podwójnych mają ciągłą pochodną, ale pochodna drugiego rzędu jest nieciągła. Godne polecenia (i wykonania) jest nieco trudniejsze ćwiczenie polegające na napisaniu odpowiednich procedur umożliwiających konstruowanie takich funkcji sklejanych i wypróbowaniu ich w aplikacji.

B.4. Sklejane krzywe kwaternionowe

Animację ruchu kulistego można przeprowadzić przez zadanie położeń kątowych obiektu w pewnych chwilach i dokonanie interpolacji między tymi położeniami. Położenia kątowe wygodnie jest reprezentować za pomocą kwaternionów, zgodnie z opisem w podrozdziale A.4. Za pomocą funkcji Slerp łatwo jest otrzymać animację, w której obiekt obraca się wokół ustalonej osi ze stałą prędkością kątową od zadanego położenia początkowego do końcowego, ale większym wyzwaniem jest otrzymanie takiego ruchu, w którym obiekt przejdzie przez wiele zadanych położeń, poruszając się tak, aby oś obrotu i prędkość kątowa zmieniały się płynnie.

Mamy zatem dany rosnący ciąg liczb $t_0, \ldots, t_M$ i ciąg kwaternionów jednostkowych $q_0, \ldots, q_M$ określających obroty obiektu do położeń zadanych w chwilach $t_0, \ldots, t_M$. Zadaniem jest znalezienie funkcji $q\colon [t_0, t_M] \to \mathbb{H}$, takiej że $q(t_i) = q_i$ dla $i = 0, \ldots, M$, przy czym dla każdego $t \in [t_0, t_M]$ kwaternion $q(t)$ ma być jednostkowy, a funkcja q ma mieć co najmniej ciągłą pochodną — prędkość kątowa w chwili t jest równa $2|q'(t)|$.

Zobaczmy dwa z wielu możliwych sposobów rozwiązania tego zadania. Pierwszy sposób polega na znalezieniu w przestrzeni $\mathbb{R}^4$ „zwykłej” sklejanej krzywej interpolacyjnej $\boldsymbol{s}$. Reprezentację B-sklejaną takiej krzywej (klasy C^2) możemy skonstruować za pomocą opi-

sanej w poprzednim punkcie procedury `ConstructCubicInterpBSplinef`. Oczywiście zarówno punkty kontrolne tej krzywej, jak i punkty $\boldsymbol{s}(t)$ inne niż $q_0, \ldots, q_M$ nie muszą być (i na ogół nie są) wektorami o długości 1 (czyli nie są kwaternionami jednostkowymi). Ale po obliczeniu punktu $\boldsymbol{s}(t)$ możemy przyjąć $q(t) = \frac{1}{\|\boldsymbol{s}(t)\|}\boldsymbol{s}(t)$, otrzymując kwaternion jednostkowy reprezentujący odpowiedni obrót dla chwili t.[10]

Opisany wyżej sposób jest stosunkowo prosty, niezawodny[11] i w wielu przypadkach wystarczający; jeśli kąty między kolejnymi danymi kwaternionami są małe (czyli kolejne zadane położenia kątowe nie są zbyt odległe od siebie), to punkty krzywej $\boldsymbol{s}$ mają długości bliskie 1, a to oznacza, że prędkość kątowa ruchu obrotowego jest w każdej chwili t bliska $2\|\boldsymbol{s}'(t)\|$. Jeśli jednak istnieje kwaternion q_A, taki że $q_i = q_0 \cdot q_A^{t_i - t_0}$ dla $i = 1, \ldots, M$, to obiekt może przyjąć kolejno wszystkie zadane położenia, obracając się ze stałą prędkością. Tymczasem interpolacja tych kwaternionów przy użyciu rozpatrywanego tu sposobu spowoduje ruch ze zmieniającą się prędkością kątową — obiekt między zadanymi położeniami będzie zwalniał i przyspieszał[12].

Drugi sposób polega na skonstruowaniu krzywej, której *wszystkie* punkty są wektorami jednostkowymi w $\mathbb{R}^4$ (czyli są kwaternionami jednostkowymi). Rozmaite opisywane w literaturze konstrukcje takich krzywych są oparte na modyfikacjach algorytmów znajdowania punktów krzywych Béziera i B-sklejanych. Zmodyfikujemy algorytm zdefiniowany wzorem (B.5). Mając liczbę $t \in [u_k, u_{k+1})$ oraz punkty $\boldsymbol{d}_{k-n}^{(0)}, \ldots, \boldsymbol{d}_k^{(0)}$ będące wektorami jednostkowymi w $\mathbb{R}^4$, obliczymy kolejne punkty przy użyciu funkcji Slerp:

$$\begin{cases} \boldsymbol{d}_i^{(j)} = \mathrm{Slerp}\left(\boldsymbol{d}_{i-1}^{(j-1)}, \boldsymbol{d}_i^{(j-1)}; \alpha_i^{(j)}\right) \\ \alpha_i^{(j)} = \dfrac{t - u_i}{u_{i+n+1-j} - u_i}, \end{cases} \qquad i = k-n+j, \ldots, k, \quad j = 1, \ldots, n. \tag{B.9}$$

Wszystkie otrzymane w ten sposób punkty, w tym ostatni, są wektorami o długości 1, możemy zatem przyjąć $q(t) = \boldsymbol{d}_k^{(n)}$. Tak określona krzywa umożliwia odtworzenie ruchu ze stałą prędkością kątową wokół ustalonej osi, ale problem polega na obliczeniu punktów kontrolnych krzywej interpolacyjnej — trzeba w tym celu rozwiązać układ równań nieliniowych.

Na listingu B.7 jest przedstawiona pewna propozycja algorytmu konstruowania sklejanej krzywej interpolacyjnej, której punkty są kwaternionami jednostkowymi. Krzywa jest reprezentowana podobnie jak kubiczna krzywa B-sklejana, tj. za pomocą ciągu węzłów $u_0, \ldots, u_N$ i punktów kontrolnych $\boldsymbol{d}_0, \ldots, \boldsymbol{d}_{N-4}$, które są wektorami jednostkowymi. Procedura `QuatSlerpdeBoorf` realizuje zmodyfikowany algorytm de Boora opisany wzorem (B.9). Procedura ta służy do obliczania punktów krzywej w trakcie animacji, ale jest też pomocniczym podprogramem potrzebnym do obliczenia punktów kontrolnych krzywej.

W liniach 7–10 procedura wyszukuje przedział $[u_k, u_{k+1})$, do którego należy parametr t, po czym w liniach 12–17 realizuje właściwe obliczenie. Gdyby nie było błędów zaokrągleń, to wszystkie otrzymane punkty byłyby wektorami jednostkowymi. Wywołanie w linii 19 pro-

[10] W zasadzie dzielenie przez $\|\boldsymbol{s}(t)\|$ nie jest konieczne, bo wzór $(0, \boldsymbol{u}) = q \cdot (0, \boldsymbol{w}) \cdot q^{-1}$ realizuje obrót dla dowolnego kwaternionu $q \neq 0$, ale do wzoru (A.5) trzeba podstawić współrzędne kwaternionu jednostkowego.

[11] Sposób ten może zawieść tylko z powodu błędnych danych lub braku wolnej pamięci RAM.

[12] Ten efekt może być tak słabo zauważalny, że aż nieistotny, ale on jest.

Listing B.7. Procedury `QuatSlerpdeBoorf` i `ConstructQuaternionInterpSplinef`

C

```
void QuatSlerpdeBoorf ( int n, int lkn, float *knots, float *cp, float t,
                        float *p )
{
  int   i, j, k;
  float d[(MAX_DEG+1)*4], alpha;

  for ( k = n, j = lkn-n;  j-k > 1; ) {  /* bisekcja */
    i = k + (j-k)/2;
    if ( t >= knots[i] ) k = i; else j = i;
  }
  memcpy ( d, &cp[(k-n)*4], (n+1)*4*sizeof(float) );
  for ( j = 1; j <= n; j++ )
    for ( i = k-n+j; i <= k; i++ ) {
      alpha = (t-knots[i])/(knots[i-j+n+1]-knots[i]);
      QuatSlerpf ( &d[(i-k-j+n)*4],
                   &d[(i-k-j+n)*4], &d[(i-k-j+n+1)*4], alpha );
    }
  memcpy ( p, d, 4*sizeof(float) );
  V4Normalisef ( p );  /* kompensowanie błędów zaokrągleń */
} /*QuatSlerpdeBoorf*/

static void ModifyQuatCPf ( int N, float *knots, int i, float *qcp,
                            float *qt )
{
  float qf[4], qd[4];

  QuatSlerpdeBoorf ( N, knots, qcp, knots[i+3], qf );
  QuatRDivf ( qd, &qt[4*i], qf );
  QuatMultf ( qf, qd, &qcp[4*(i+1)] );
  memcpy ( &qcp[4*(i+1)], qf, 4*sizeof(float) );
} /*ModifyQuatCPf*/

char ConstructQuaternionInterpSplinef ( int *N, float *knots, float *qcp,
                                        int M, float *ikn, float *qt )
{
#define TOL 1.0e-5
  int   i, lkn;
  float qf[4], qd[4], dist, ldist, d;

  if ( !ConstructCubicInterpBSplinef ( N, knots, qcp, M, ikn, 4, qt ) )
    return false;
  for ( i = 0, lkn = *N;  i < lkn-3;  i++ )
    V4Normalisef ( &qcp[4*i] );
  for ( ldist = 4.0;  ;  ldist = dist ) {
    for ( dist = 0.0, i = 1;  i < M; i++ ) {
```

```
46:       QuatSlerpdeBoorf ( 3, lkn, knots, qcp, ikn[i], qf );
47:       V4Subtractf ( qd, &qt[4*i], qf );
48:       if ( (d = V4DotProductf ( qd, qd )) > dist )
49:         dist = d;
50:     }
51:     if ( dist >= ldist )
52:       return false;
53:     if ( dist <= TOL*TOL )
54:       break;
55:     ModifyQuatCPf ( lkn, knots, 1, qcp, qt );
56:     d = knots[4]-knots[1];  d /= (d+knots[5]-knots[2] );
57:     QuatSlerpf ( &qcp[4], &qcp[0], &qcp[8], d );
58:     for ( i = 2; i < M; i++ )
59:       ModifyQuatCPf ( lkn, knots, i, qcp, qt );
60:     d = knots[lkn-2]-knots[lkn-5];  d /= (d+knots[lkn-1]-knots[lkn-4]);
61:     QuatSlerpf ( &qcp[4*(M+1)], &qcp[4*M], &qcp[4*(M+2)], d );
62:   }
63:   return true;
64: #undef TOL
65: } /*ConstructQuaternionInterpSplinef*/
```

cedury normalizacji[13] ma na celu zmniejszenie skutków tych błędów (w arytmetyce zmiennopozycyjnej nie da się ich całkowicie wyeliminować).

Procedura `ConstructQuaternionInterpSplinef` konstruuje reprezentację krzywej na podstawie danych węzłów interpolacyjnych $t_0, \ldots, t_M$ i odpowiadających im kwaternionów jednostkowych $q_0, \ldots, q_M$. Pierwszym krokiem konstrukcji jest znalezienie „zwykłej" kubicznej interpolacyjnej krzywej B-sklejanej za pomocą opisanej wcześniej procedury `ConstructCubicInterpBSplinef`. W liniach 42–43 punkty kontrolne tej krzywej są zamieniane na wektory (kwaterniony) jednostkowe. W ten sposób powstaje początkowe przybliżenie poszukiwanej krzywej; jej punkty odpowiadające węzłom $t_1, \ldots, t_{M-1}$ są przybliżeniami punktów danych $q_1, \ldots, q_{M-1}$ (ale już jest $q(t_0) = q_0$ i $q(t_M) = q_M$).

Zmiennej `*N` procedura przypisuje numer $N = M + 6$ ostatniego węzła krzywej sklejanej, ciąg węzłów tej krzywej jest wpisywany do tablicy `knots`, a w tablicy `qcp` zostają obliczone punkty kontrolne krzywej — każdy z nich jest kwaternionem, czyli kolejną czwórką liczb w tej tablicy. Liczba punktów kontrolnych jest równa $M + 3$.

Pętla w liniach 44–62 realizuje proces iteracyjnego „poprawiania" punktów kontrolnych. Proces ten zostaje przerwany, gdy maksymalna odległość punktu $q(t_i)$ od q_i nie przekracza progu tolerancji `TOL`; ustalenie go na 10^{-5} wydaje się wystarczające w animacji i nie przekracza możliwości arytmetyki pojedynczej precyzji. Odległości są obliczane w pętli w liniach 45–50, po czym obliczenia są przerywane, jeśli maksymalna odległość mieści się w tolerancji albo jeśli maksymalna odległość wzrosła, co oznacza brak zbieżności procesu poprawiania.

Zasadnicze obliczenie („poprawienie" jednego punktu) wykonuje procedura `ModifyQuatCPf`. W linii 27 oblicza ona punkt $q(t_i)$ krzywej reprezentowanej przez *bieżące* punkty

[13] będące prawdopodobnie przejawem mojej nadmiernej gorliwości

kontrolne, a kolejne instrukcje obliczają kwaternion $q_i \cdot q(t_i)^{-1} \cdot \boldsymbol{d}_{i+1}$, który natychmiast zastąpi w tablicy dotychczasowy punkt kontrolny $\boldsymbol{d}_{i+1}$. Skutkiem jest przemieszczenie punktu $q(t_i)$ w stronę punktu q_i. Powoduje to także zmianę punktów $q(t_{i-1})$ i $q(t_{i+1})$ (z wyjątkiem $q(t_0)$ i $q(t_M)$), ale ich przemieszczenia (zazwyczaj) są mniejsze.

Po poprawieniu punktów $\boldsymbol{d}_2$ i $\boldsymbol{d}_{N-6}$ następuje jeszcze modyfikacja punktów $\boldsymbol{d}_1$ i $\boldsymbol{d}_{N-5}$ (linie `56–57` i `60–61`). Ma ona na celu spełnienie warunków brzegowych. Pochodna drugiego rzędu „zwykłej" naturalnej krzywej sklejanej w węzłach $u_3 = t_0$ i $u_{N-3} = t_M$, będących końcami dziedziny, jest wektorem zerowym[14]. Dla krzywej położonej na sferze $\{\, q \in \mathbb{H} : |q| = 1 \,\}$ analogiczny warunek jest taki, że *wektory* $q''(t_0)$ i $q''(t_M)$ mają odpowiednio kierunki q_0 i q_M, dzięki czemu są prostopadłe do wszystkich wektorów stycznych do sfery w punktach q_0 i q_M (w tym do $q'(t_0)$ i $q'(t_M)$). Takie warunki brzegowe zapewniają odtworzenie ruchu obrotowego wokół ustalonej osi ze stałą prędkością kątową — o ile tylko warunki interpolacyjne umożliwiają taki ruch.

Trzeba pamiętać, że zadanie interpolacji może nie mieć rozwiązania w zbiorze krzywych określonych wzorem (B.9), gdy kolejne punkty w ciągu $q_0, \ldots, q_M$ są od siebie bardzo odległe lub gdy węzły $t_0, \ldots, t_M$ są rozmieszczone zbyt nierównomiernie (tj. gdy długości przedziałów $[t_{i-1}, t_i]$ i $[t_i, t_{i+1}]$ znacznie się różnią). Dlatego *żaden algorytm* nie może dać *gwarancji* znalezienia rozwiązania[15]. Szybkość zbieżności procesu iteracyjnego zaimplementowanego w opisanej tu procedurze nie jest duża — w moich eksperymentach, dla „dobrych" danych, błąd interpolacji malał w każdej iteracji (przebiegu zewnętrznej pętli) w przybliżeniu o połowę i do znalezienia rozwiązania trzeba było kilkunastu iteracji. Istnieją metody szybciej zbieżne i warto je wypróbować, ale moim zamiarem było znalezienie algorytmu wystarczająco skutecznego i jak najprostszego do zaprogramowania, co uczyniłem.

[14] Czyli jest prostopadła do wszystkich wektorów.

[15] Jeśli rozwiązanie nie istnieje, to każdy algorytm numerycznego rozwiązywania równań nieliniowych gwarantuje, że rozwiązania nie znajdzie. Ale jeśli ono istnieje, to żaden algorytm nie daje gwarancji znalezienia go. Wiele zależy od przybliżenia początkowego, a znalezienie takiego, które prowadzi do sukcesu, bywa bardzo trudne.

C

Kolory, barwy i ich współrzędne

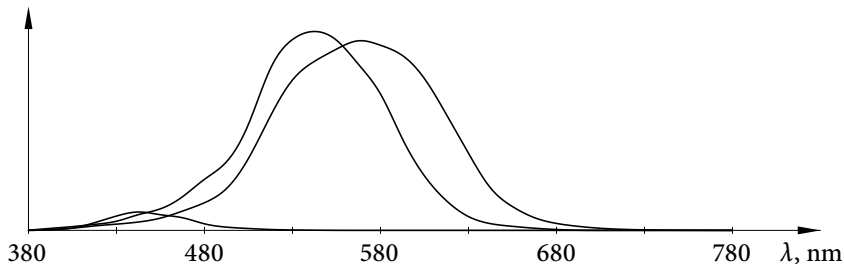
C.1. Widzenie trójbarwne

Podstawy współczesnej kolorymetrii stworzył w 1853 r. Hermann Grassmann, który m.in. wprowadził opis barw za pomocą przedstawionych dalej pojęć intensywności (tj. jasności), odcienia, nasycenia, dominującej długości fali świetlnej i barw dopełniających i sformułował prawa addytywnego mieszania barw. Prawa Grassmanna w szczególności opisują fakt, że wiele różnych bodźców świetlnych (tj. świateł o różnych widmach) pobudza receptory w oku w identyczny sposób, co umożliwia oddanie gamy barw dostatecznie szerokiej dla większości zastosowań praktycznych przez mieszanie trzech barw podstawowych.

W siatkówce ludzkiego oka są dwa rodzaje receptorów: **czopki** i **pręciki**. Pręciki są bardziej czułe (i w słabym oświetleniu przejmują rolę głównego źródła sygnałów dla zmysłu wzroku, jest to tzw. **widzenie nocne**, lub **skotopowe**), ale istnienie tylko jednego rodzaju pręcików nie umożliwia rozróżniania przez nie kolorów. Natomiast czopki trzech rodzajów wykazują różną czułość dla fal o ustalonej długości, wskutek czego odpowiednio jasne światło o określonym widmie (tj. funkcji opisującej gęstość mocy promieniowania w zależności od długości fali) powoduje wysyłanie z oczu do mózgu trzech (skalarnych) sygnałów w odpowiedzi na pobudzenie czopków poszczególnych rodzajów (ma wtedy miejsce tzw. **widzenie dzienne** albo **fotopowe**, jest też stan pośredni, tj. **widzenie o zmierzchu**, gdy „czynne" są i czopki i pręciki). To dlatego do reprezentowania barw i w szczególności wyświetlania obrazów na ekranach komputerów i telewizorów wystarczają ludziom trzy liczby na piksel.

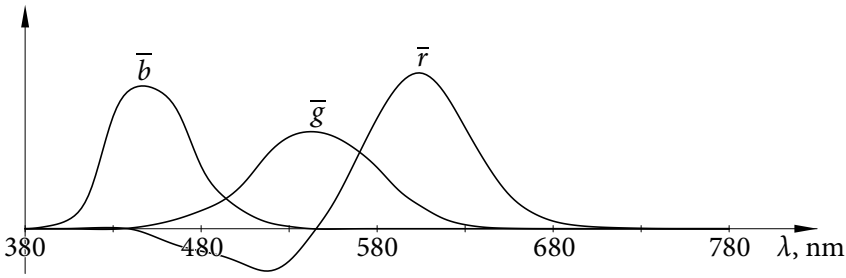
Rysunek C.1 przedstawia wykresy czułości czopków, przy czym trzeba pamiętać, że dokładne pomiary wartości funkcji przedstawionych na tych wykresach są trudne do przeprowadzenia, a ponadto u różnych osób funkcje czułości mogą się trochę różnić. Niemniej, z praw Grassmanna wynika, że poziom bodźca, który pobudza receptor jest całką z iloczynu widma światła padającego na ten receptor i funkcji czułości receptora w przedziale długości fal światła widzialnego[1].

[1]Trzeba pamiętać, że ten model widzenia barwnego jest uproszczony; zakłada się, że funkcja czułości receptora nie zależy od miejsca receptora na siatkówce i nie zmienia się w czasie.



Rysunek C.1. Funkcje czułości czopków na światło o określonej długości fali

Do badania widzenia barwnego[2] służy **kolorymetr klinowy**. Urządzenie takie zawiera komorę, której ściany są czarne i w której jest umieszczony klin o dwóch białych ścianach. Komora ma dwa okienka, przez które wpada światło badane i światło wzorcowe; światło wpadające przez każde z okienek oświetla tylko jedną ścianę klina. Źródłem światła wzorcowego są żarówka z filtrem przepuszczającym długie fale (tj. światło czerwone, o długości fali $\lambda \geqslant 700\,\mathrm{nm}$) oraz lampy rtęciowe emitujące fale krótsze (światło zielone, $\lambda = 546.1\,\mathrm{nm}$, i niebieskie, $\lambda = 435.8\,\mathrm{nm}$). Osoba badana, oglądając przez wziernik ściany klina, ma za zadanie tak ustawić przysłony źródeł światła wzorcowego, aby (zdaniem tej osoby) obie ściany były oświetlone identycznie. Współrzędne światła badanego, określone przez układ odniesienia świateł wzorcowych, odczytuje się z podziałek na przysłonach. Współrzędne te są nieujemne, ale domieszanie światła wzorcowego do światła badanego umożliwia także pomiary współrzędnych ujemnych — są nimi liczby odczytane z podziałek przysłon światła wzorcowego domieszanego do światła badanego, ze znakiem minus.



Rysunek C.2. Składowe trójchromatyczne

Na podstawie eksperymentów zostały znalezione trzy funkcje długości fali, $\overline{r}$, $\overline{g}$ i $\overline{b}$, zwane **składowymi trójchromatycznymi** (rys. C.2). Umożliwiają one obliczenie współrzędnych światła o określonym widmie f w układzie współrzędnych wyznaczonym przez opisane wyżej światła wzorcowe. W tym celu należy scałkować iloczyny widma z każdą z tych

[2]Słowa „kolor" używam tu do określenia własności światła opisanych przez jego widmo, natomiast „barwa" odnosi się do wrażenia wzrokowego spowodowanego przez to światło.

funkcji. Teoretycznie można dalej przejść do układu współrzędnych określonego przez triadę elementów świecących monitora i, jeśli współrzędne w tym układzie są nieujemne, odtworzyć barwę światła o widmie f na ekranie. Jeśli jednak obraz zawiera punkty o barwach niemożliwych do odtworzenia, to trzeba go tak przetworzyć, aby zniekształcenia barw były niezauważalne, o czym będzie mowa dalej.

C.2. Diagram CIE

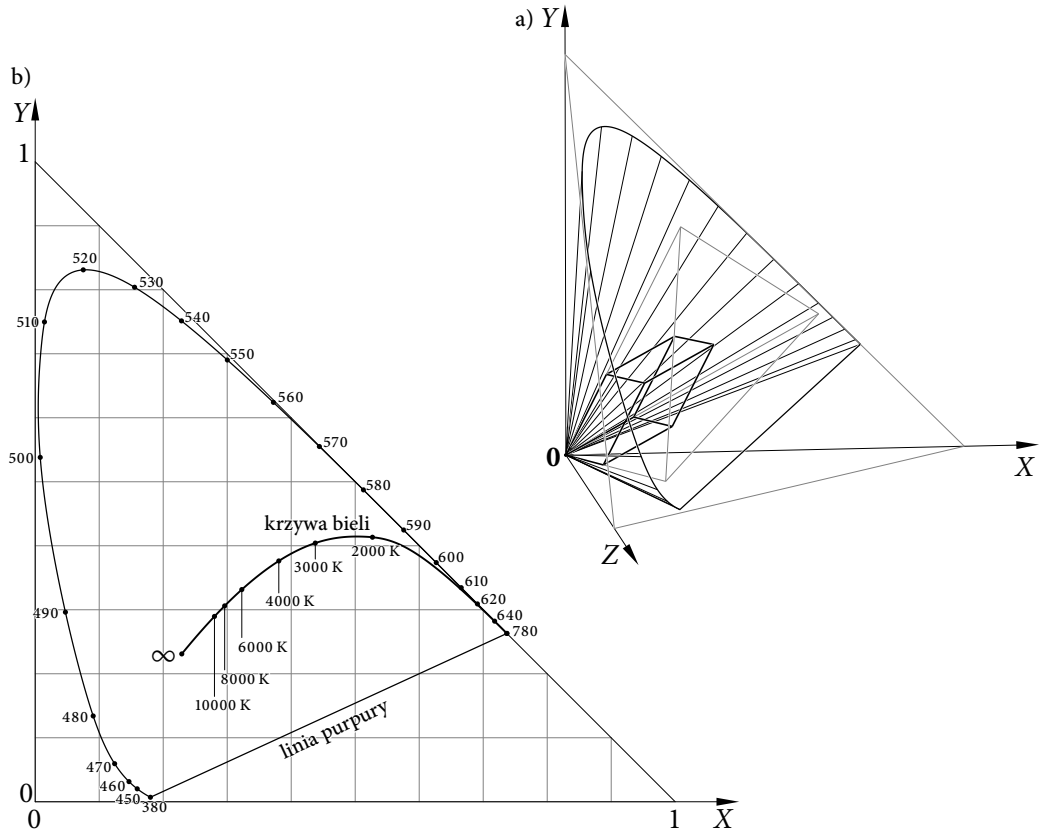
W roku 1931 **Międzynarodowa Komisja Oświetleniowa** (*CIE — Commission Internationale de l'Éclairage*) opracowała pewien układ współrzędnych zwany **CIE XYZ**, obecnie przyjęty jako standard, na podstawie którego są określane wszystkie inne układy współrzędnych w przestrzeni barw. Wszystkie barwy światła realizowalnego fizycznie mają w tym układzie współrzędne nieujemne. Należy podkreślić, że „światło" odpowiadające elementom układu odniesienia nie istnieje, tzn. nie istnieją funkcje nieujemne opisujące widmo światła dla tych elementów.

Punkty odpowiadające barwom światła widzialnego tworzą pewną bryłę wypukłą i stożkową, tzn. taką, że iloczyn wektora współrzędnych każdego punktu tej bryły i dowolnej liczby nieujemnej również reprezentuje barwę światła widzialnego (czyli odpowiadający mu punkt też należy do tej bryły). Ograniczając całkowitą moc światła do pewnej stałej, otrzymamy bryłę pokazaną na rysunku C.3a. Na rysunku C.3b jest pokazana część wspólna tej bryły z płaszczyzną $X + Y + Z = 1$. Można zauważyć, że każdy przekrój bryły barw płaszczyzną o równaniu $X + Y + Z = \text{const} > 0$ jest figurą podobną do pozostałych przekrojów, przy czym stała po prawej stronie tego równania określa moc światła. Dzięki temu możemy (i będziemy dalej) traktować współrzędne XYZ o sumie równej 1 jak współrzędne barycentryczne w układzie określonym przez wierzchołki trójkąta na rysunku, który przedstawia każdy taki przekrój.

Rysunek C.3b przedstawia tzw. **diagram chromatyczności CIE**. Brzeg obszaru światła widzialnego składa się z dwóch części, zwanych **krzywą tęczy** i **linią purpury**. Punkty krzywej tęczy reprezentują światło ściśle monochromatyczne, o jednej długości fali. Na rysunku obok pewnych punktów krzywej tęczy są podane odpowiednie długości fali. Natomiast punkty linii purpury (która jest odcinkiem) reprezentują mieszaniny (w różnych proporcjach) światła o skrajnych długościach fali: najkrótszej ($\lambda = 380\,\text{nm}$) i najdłuższej ($\lambda \geqslant 700\,\text{nm}$).

Widoczna wewnątrz obszaru **krzywa bieli** składa się z punktów reprezentujących barwy światła emitowanego przez **ciało doskonale czarne**[3] rozgrzane do różnych temperatur. W zasadzie każdy punkt tej krzywej reprezentuje barwę, którą można uznać za światło białe, choć włókno żarówki o temperaturze ok. 3000 K daje światło żółtawe, czyli „ciepłe", zaś gwiazdy Syriusz A i B (o temperaturach powierzchni ok. 10000 K i 25000 K) uznajemy za ciała niebieskie. Wiele monitorów umożliwia wybranie barwy światła białego przez ustawienie odpowiadającej mu „temperatury", najczęściej między 6000 K (temperatura powierzchni Słońca) a 7500 K (temperatura powierzchni gwiazdy Procjon B). Polega to na ustaleniu mocy

[3] Fizycy wiedzą, co to takiego.

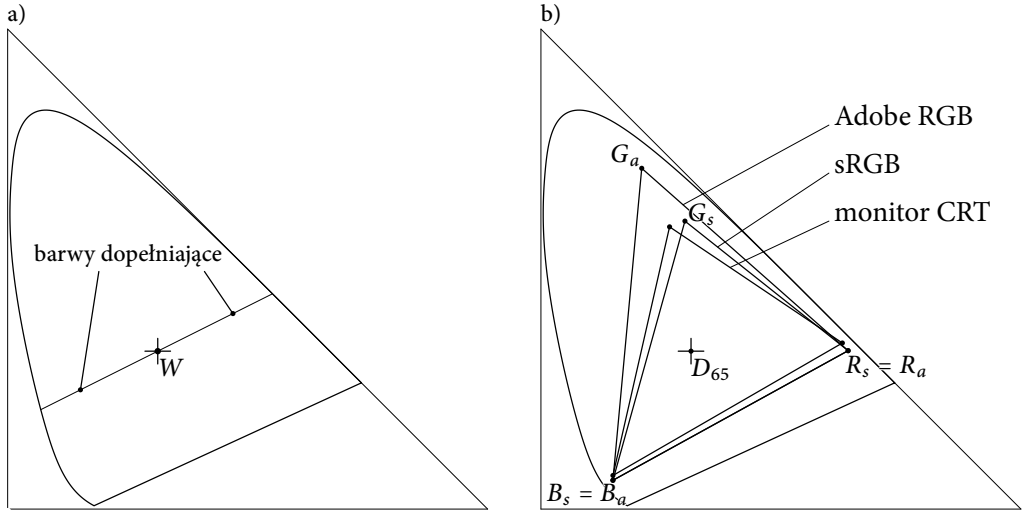


Rysunek C.3. a) bryła barw widzialnych w układzie CIE XYZ, b) diagram CIE

światła emitowanego przez poszczególne elementy (luminofory w lampie kineskopowej lub diody świetlne w nowocześniejszych wyświetlaczach) triad pikseli, które mają przypisane maksymalne wartości składowych r, g, b. Tak wybrane punkty bieli znajdują się w pobliżu środka ciężkości trójkąta, w który wpisany jest obszar barw widzialnych[4].

Nasyceniem barwy (*saturation*) nazywamy względną odległość s punktu reprezentującego tę barwę na diagramie CIE od przyjętego punktu bieli; jest ono równe 1 dla punktów na brzegu obszaru barw (czyli punktów na krzywej tęczy lub na linii purpury), a nasycenie barwy bieli jest równe 0. Nasycenie barwy reprezentowanej przez punkt $\boldsymbol{p}$ możemy obliczyć, dzieląc odległość punktu $\boldsymbol{p}$ od punktu bieli W przez długość przechodzącego przez punkt $\boldsymbol{p}$ odcinka, którego jednym końcem jest punkt bieli, a drugi koniec leży na brzegu obszaru barw widzialnych. Na rysunku C.4a są zaznaczone dwa punkty o tym samym nasyceniu $s = 0.66$. Punkty o tym samym nasyceniu położone po przeciwnych stronach punktu bieli reprezentują **barwy dopełniające**. Mieszając *w pewnych proporcjach* światła o barwach dopełniających, możemy otrzymać światło białe. Zauważmy, że obszar barw widzialnych

[4] Punkt ten nie leży na krzywej bieli, choć jest blisko niej.



Rysunek C.4. a) barwy dopełniające, b) trójkąty barw układów sRGB, Adobe RGB i monitora

jest niesymetryczny, przez co na ogół punkt bieli nie jest środkiem odcinka, którego końce odpowiadają barwom dopełniającym. Nie jest to więc mieszanie „pół na pół".

Dominującą długość fali światła możemy odczytać z diagramu CIE, znajdując taki punkt $\boldsymbol{q}$ na krzywej tęczy, że punkt reprezentujący barwę tego światła leży na odcinku łączącym punkty $\boldsymbol{q}$ i W. Barwę o nasyceniu s reprezentowaną przez punkt $\boldsymbol{p}$ możemy otrzymać, mieszając światło białe ze światłem monochromatycznym o dominującej długości fali w proporcji $1 - s : s$. Jeśli jednak półprosta o początku W przechodząca przez punkt $\boldsymbol{p}$ przecina linię purpury, to dominująca długość fali dla takiego światła nie istnieje.

Triady elementów świecących pikseli wyznaczają trójkąt w obszarze światła widzialnego na diagramie CIE. Wszystkim barwom możliwym do wyświetlenia odpowiadają punkty tego trójkąta[5], ponieważ moc światła emitowanego przez każdy element triady jest nieujemna[6]. Można zatem postawić problem: co zrobić, jeśli pewne punkty na obrazie mają kolory, których nie da się odtworzyć?

Podstawą do opracowania sposobów radzenia sobie z tym problemem, tj. modyfikowania obrazu tak, aby „pozbyć się" kolorów niemożliwych do odtworzenia na ekranie[7], są opisane wyżej pojęcia. Aby modyfikacje były niezauważalne, trzeba wziąć pod uwagę, na co ludzki zmysł wzroku jest najbardziej wyczulony, a na co mniej. Zatem, najmniej zauważalne są

[5]Rozważamy tu światło o ustalonej mocy. W rzeczywistości wszystkie barwy możliwe do wyświetlenia na ekranie są reprezentowane przez punkty równoległościennej kostki zawartej w bryle pokazanej na rysunku C.3a.

[6]W monitorach i telewizorach różnych typów punkty odpowiadające barwom poszczególnych elementów triady są nieco inne, ale w praktyce są one wystarczająco bliskie punktów przyjętych w standardach stosowanych do reprezentowania obrazów, aby zniekształcenia barw były niedostrzegalne — także wtedy, gdy ustawienia monitora (w tym wybór punktu bieli) nie odpowiadają standardowi.

[7]Ten sam problem występuje podczas przygotowywania obrazów do druku, choć tam ma miejsce tzw. **subtraktywne mieszanie barw**, realizowane przez tłumienie pewnych składowych światła białego przez pigmenty, zobacz podrozdział C.5.

niewielkie zmiany jasności całego obrazu i przesunięcia punktu bieli — patrząc na wydrukowany obraz (lub fotografię) w świetle słonecznym i w świetle żarówki, widzimy „to samo". Dość słabo zauważalne są zmiany nasycenia barwy (całego obszaru o stałym kolorze na obrazie), a nieco bardziej zmiany odcienia, wyznaczonego przez dominującą długość fali lub proporcję kolorów czerwonego i niebieskiego dla purpury. Najbardziej widoczne są wszelkie nieciągłości barwy obszarów sąsiadujących na obrazie. Podczas oglądania serii obrazów (w animacji) wyraźnie dostrzegalne są też skokowe zmiany kolorów w czasie.

Jeśli kolory pewnych pikseli są poza obszarem barw odtwarzalnych, to zastąpienie ujemnej składowej r, g lub b przez zero może dać niezadowalający efekt. Znacznie lepszym pomysłem jest **desaturacja**, czyli zmieszanie danego koloru ze światłem białym, co nie zmienia dominującej długości fali ani odcienia purpury. Ale skutek zrobienia tego tylko dla pikseli, których kolory „wystają" poza dozwolony obszar, też może być niezadowalający, w związku z czym lepiej jest skorygować *wszystkie* piksele na obrazie. Stopień desaturacji może być pewną funkcją odcienia, dobraną indywidualnie do obrazu. W animacji podobną korektę trzeba zrobić dla całej sekwencji klatek. Podsumowując, nie ma jednego prostego algorytmu korygowania kolorów, dającego zawsze świetne wyniki. Z drugiej strony, wiele osób (zajmujących się grafiką po amatorsku) się tym nie przejmuje i tworzy piękne obrazy.

C.3. Układy współrzędnych RGB i korekcja gamma

Obecnie w grafice najczęściej są używane dwa standardowe układy współrzędnych RGB, których punkty odniesienia odpowiadają barwom czerwonej, zielonej i niebieskiej. Punkty odniesienia układu **sRGB**, opracowanego w roku 1996 wspólnie przez firmy Hewlett–Packard i Microsoft, mają w układzie CIE XYZ współrzędne $R_s = (0.64, 0.33, 0.03)$, $G_s = (0.3, 0.6, 0.1)$ i $B_s = (0.15, 0.06, 0.79)$. Trójkąt o tych wierzchołkach jest bliższy trójkątów odpowiadających typowym monitorom niż trójkąt o wierzchołkach $R_a = R_s$, $G_a = (0.21, 0.71, 0.18)$, $B_a = B_s$, przy użyciu których jest określony układ **Adobe RGB** firmy Adobe Systems Inc., rok 1998, częściej niż w grafice stosowany w fotografii cyfrowej, obróbce obrazów i poligrafii. Trójkąty dla obu układów są pokazane na rysunku C.4b. W obu tych układach przyjęty jest punkt bieli $W = (0.3127, 0.3290, 0.3583)$. Światło o takiej barwie, zbliżonej do barwy światła dziennego[8], jest emitowane przez ciało doskonale czarne rozgrzane do temperatury ok. 6500 K, stąd punkt ten jest oznaczany symbolem D_{65}.

Przejście od współrzędnych CIE XYZ do sRGB składa się z trzech kroków. Pierwszy krok jest przekształceniem liniowym, tj. zmianą układu współrzędnych kartezjańskich, opisaną wzorem

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{bmatrix} 3.2406 & -1.5372 & -0.4986 \\ -0.9689 & 1.8758 & 0.0415 \\ 0.0557 & -0.2040 & 1.0570 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}.$$

[8] Światło dochodzące w dane miejsce bezpośrednio od Słońca, o temperaturze ok. 6000 K, jest zmieszane z dochodzącym ze wszystkich stron światłem słonecznym rozproszonym w atmosferze. Ponieważ zaś atmosfera najsilniej rozprasza fale krótkie, czyli światło niebieskie, temperatura barwy światła dziennego jest wyższa niż temperatura powierzchni Słońca.

Pozostałe kroki są przekształceniami nieliniowymi. Krok drugi polega na obcięciu każdej składowej do przedziału $[0,1]$.

Ludzki zmysł wzroku jest wyspecjalizowany w rozróżnianiu *względnych* przyrostów jasności światła, tzn. podobnie są postrzegane zmiany na przykład o 3% mocy światła jasnego i ciemniejszego. Gdyby zatem składowe r, g i b były reprezentowane za pomocą liczb ośmiobitowych proporcjonalnie do poziomu składowych[9], to dokładność względna reprezentacji barw ciemniejszych byłaby za mała — szczegóły przedmiotów słabo oświetlonych byłyby widoczne na obrazie za mało dokładnie[10]. Ponadto moc L światła emitowanego przez piksel w lampie kineskopowej zależy w sposób nieliniowy od napięcia V przyłożonego do elektrod modulujących strumień elektronów. W dobrym przybliżeniu zależność ta jest opisana przez funkcję potęgową $L(V) = cV^\gamma$, z wykładnikiem $\gamma \in [1.8, 2.8]$ i pewną stałą c. Podobne odwzorowanie liczb opisujących składowe koloru na moc światła emitowanego przez piksel realizują nowocześniejsze wyświetlacze LED lub LCD. Łatwo się o tym przekonać, rysując obok siebie dwa prostokąty, jeden wypełniony stałym kolorem, na przykład szarym, z pikselami o składowych r, g, b mających stałą wartość, i drugi, w którym białe i czarne piksele są ułożone w szachownicę. Oba prostokąty oglądane z pewnej odległości będą tak samo jasne, gdy wartość przypisana składowym szarych pikseli jest bliska trzem czwartym wartości maksymalnej (przypisanej pikselom białym).

Z tych powodów odwzorowanie poziomu każdej składowej w układzie sRGB jest nieliniowe; zgodnie z opisaną wyżej zależnością, obliczony poziom na przykład składowej $r \in [0,1]$ należy zamienić na liczbę $R = r^{1/\gamma}$. Liczbę tę można następnie pomnożyć przez 255 i zaokrąglić do najbliższej liczby całkowitej, otrzymując liczbę ośmiobitową. Aby w obliczeniach numerycznych uniknąć problemów związanych z tym, że pochodna funkcji $f(x) = x^{1/\gamma}$ dla x bliskiego zera jest nieograniczona, w rzeczywistości stosowany jest nieco inny wzór:

$$R(x) = \begin{cases} 12.92x & \text{dla } x < 0.0031308, \\ 1.055x^{1/2.4} - 0.055 & \text{w przeciwnym razie.} \end{cases}$$

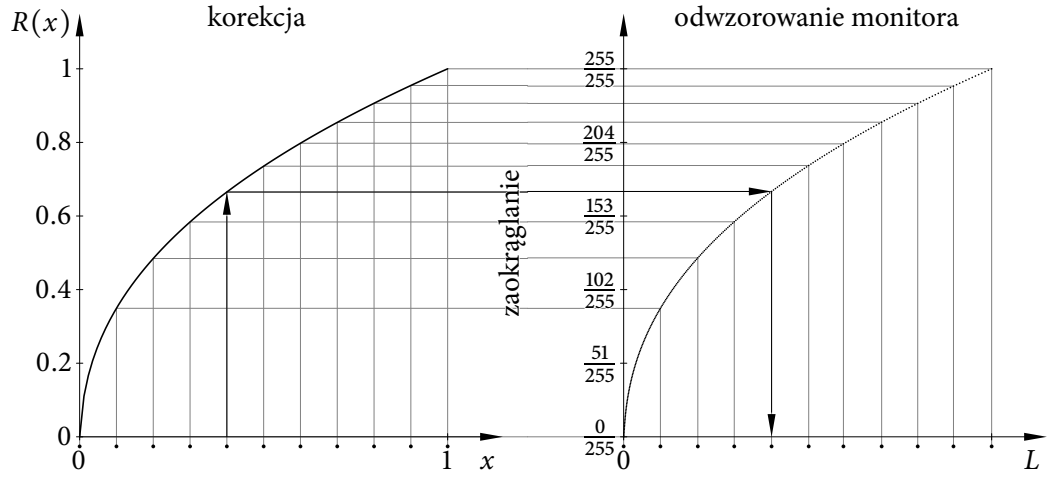
Opisane tu przekształcenie, wprowadzone w związku z monitorami kineskopowymi, ale poprawiające względną dokładność reprezentacji ciemnych barw przy użyciu niewielkiej liczby bitów na piksel, jest nazywane **korekcją gamma**.

Ideę korekcji gamma ilustruje rysunek C.5. Wykres z lewej strony przedstawia funkcję R. Punkty oznaczone kropkami przy osi poziomej są rozmieszczone w jednakowych odstępach. Wartości funkcji R w tych punktach nie są równoodległe, ale *jeśli* zależność mocy L światła emitowanego przez piksel od liczby wpisanej do bufora obrazu[11] jest proporcjonalna do odwrotności funkcji R (monitory spełniają ten warunek w dobrym przybliżeniu), to moc światła emitowanego przez odpowiednią składową piksela jest proporcjonalna do liczby opisującej tę składową przed dokonaniem korekcji.

[9] czyli w taki sposób, że maksymalny poziom składowej jest reprezentowany przez liczbę 255, a liczbom 85 i 170 odpowiadają 1/3 i 2/3 tego poziomu

[10] Zauważmy, że duża zmiana względna jednej składowej powoduje dużą zmianę odcienia, a na to wzrok jest wyczulony.

[11] Na wykresie z prawej strony zmienna niezależna jest związana z osią pionową.



Rysunek C.5. Korekcja gamma

Listing C.1. Procedura korekcji gamma sRGB

GLSL

```
#define GAMMA(x) \
  (x < 0.0031308 ? 12.92 * x : 1.055 * pow ( x, 1.0/2.4 ) - 0.055)

vec3 sRGBGamma ( vec3 colour )
{
  return vec3 ( GAMMA(colour.r), GAMMA(colour.g), GAMMA(colour.b) );
} /*sRGBGamma*/
```

Listing C.1 przedstawia procedurę realizującą korekcję gamma przy użyciu opisanej wyżej funkcji R. Zamiast niej można użyć funkcji potęgowej przyjętej w opisanym niżej układzie współrzędnych Adobe RGB; jest ona prostsza do implementacji, a różnica obrazów otrzymanych tymi sposobami jest niezauważalna: maksymalna różnica funkcji opisujących korekcje w standardach sRGB i Adobe RGB w przedziale $[0,1]$ jest mniejsza niż 0.034, a odwrotności tych funkcji różnią się o mniej niż 0.0087.

Jeśli na powierzchnię obiektu ma być nałożona tekstura zapisana w układzie sRGB, to w aplikacji OpenGL-a należy dokonać odpowiedniej konwersji, bo w obliczeniach oświetlenia są potrzebne współrzędne kartezjańskie (tj. liniowo związane z mocą światła) w przestrzeni barw. Konwersję można przeprowadzać za pomocą ewaluatora tekstury. Jeśli trzeci parametr procedury `glTexImage2D`, określający wewnętrzny format przesyłanej do pamięci GPU tablicy tekseli, ma wartość `GL_SRGB`, `GL_SRGB8`, `GL_SRGB_ALPHA` lub `GL_SRGB8_ALPHA8`, to zakłada się, że składowe R, G, B każdego teksela należy poddać przekształceniu będącemu funkcją odwrotną do funkcji $R(r)$ opisanej podanym wyżej wzorem.

Można również wygenerować obraz z pikselami „od razu” reprezentowanymi w układzie sRGB (np. w celu zapisania go w pliku). W tym celu trzeba utworzyć pozaekranowy bufor ramki i podłączyć do niego załącznik — teksturę o wewnętrznym formacie `GL_SRGB8_`-

`ALPHA8`, w której ma być utworzony obraz, a przed rysowaniem włączyć korekcję gamma, wywołując procedurę `glEnable` z parametrem `GL_FRAMEBUFFER_SRGB`.

Podobnie wygląda przejście od układu CIE XYZ do Adobe RGB, przy czym w pierwszym kroku jest używana nieco inna macierz, a w korekcji gamma jest stosowana funkcja potęgowa $f(x) = x^{1/\gamma}$ z wykładnikiem $\gamma = 563/256 \approx 2.2$. Składowe otrzymane po korekcji gamma można następnie pomnożyć przez 255 lub 65535 i zaokrąglić, otrzymując w wyniku ich reprezentacje ośmio- lub szesnastobitowe.

Choć układ Adobe RGB obejmuje większą część bryły barw (tj. więcej punktów ma w tym układzie współrzędne nieujemne), co jest zaletą, jednak nie jest zbyt rozpowszechniony. Pierwszy tego powód to spore oddalenie punktu G_a od punktu odpowiadającego barwie zielonej większości kolorowych monitorów, a zatem wszystkich barw reprezentowalnych w tym układzie i tak nie da się odtworzyć na ekranie, wyświetlenie zaś obrazu bez odpowiedniej konwersji powoduje zniekształcenie barw (zmniejszenie nasycenia barw zielonych i zamiany bieli na barwę lekko purpurową). Drugim powodem jest fakt, że rozszerzenie obszaru reprezentowalnego ma swoją cenę: mniejszą dokładność reprezentacji barw niż w układzie sRGB, odczuwalną, jeśli składowe są kodowane za pomocą liczb ośmiobitowych.

Istnieje wiele innych układów współrzędnych RGB, przyjętych jako standardy telewizyjne, a także mających specjalne zastosowania. Informacje na ich temat najprościej jest znaleźć w Internecie.

C.4. Układy z luminancją i chrominancją

Wprowadzenie kolorowej telewizji w latach pięćdziesiątych XX wieku wymagało zachowania poprawnego działania odbiorników czarno-białych, odbierających sygnał, który dotąd przenosił tylko informację o luminancji poszczególnych punktów wyświetlanych obrazów. Sygnał ten został więc uzupełniony o dwa dodatkowe sygnały tzw. **chrominancji**, umożliwiające odtworzenie barw, tj. otrzymanie sygnałów sterujących elementami r, g, b triad pikseli kolorowych kineskopów. Po obu stronach Atlantyku były przyjęte inne standardy telewizji kolorowej, ale książka ta nie jest właściwym miejscem na ich szczegółowe opisy. Warto jednak wiedzieć, że w układach współrzędnych YIQ i YUV, stosowanych w dawnej telewizji[12], pasma używane do przesyłania sygnałów chrominancji IQ lub UV były znacznie węższe niż pasmo dla sygnału luminancji Y. Rzecz w tym, że zniekształcenia chrominancji są znacznie mniej dostrzegalne dla ludzi niż zaburzenia luminancji. Ma to znaczenie także obecnie, na przykład w algorytmie kompresji JPEG, który dokonuje przejścia do układu nazwanego $Y'C_BC_R$, ze współrzędną luminancji[13] Y' i dwiema współrzędnymi chrominancji, a następnie dokonuje kompresji stratnej poszczególnych składowych — dopuszczając większe zniekształcenia (i uzyskując dzięki temu większy stopień kompresji) składowych chrominancji C_B i C_R.

[12] przekazującej obrazy za pomocą sygnału analogowego i fal radiowych

[13] Współrzędna Y', o angielskiej nazwie *luma*, jest związana z luminancją w sposób nieliniowy wskutek uwzględnienia korekcji gamma.

Przejście między układami $R'G'B'$ (bardzo zbliżonym do sRGB) i $Y'UV$ określonymi w standardzie telewizyjnym BT.709, jest opisane wzorami

$$\begin{bmatrix} Y' \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.2126 & 0.7152 & 0.0722 \\ -0.09991 & -0.33609 & 0.436 \\ 0.615 & -0.55861 & -0.05639 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix},$$

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.28033 \\ 1 & -0.21482 & -0.38059 \\ 1 & 2.12798 & 0 \end{bmatrix} \begin{bmatrix} Y' \\ U \\ V \end{bmatrix}.$$

Analogiczna macierz określona w starszym standardzie BT.601 ma w pierwszym wierszu liczby 0.299, 0.587, 0.114, użyte w aplikacji 2K do zamieniania obrazów kolorowych na jednobarwne, które dalej zostają wyświetlone w kolorach czerwonym i zielonym jako anaglify (podrozdz. 26.2). Możemy też zauważyć, że w macierzy przejścia w drugą stronę (w obu standardach) jedynki w pierwszej kolumnie sprawiają, że skutkiem zaniku (czyli zastąpienia przez 0) sygnałów chrominancji jest przypisanie współrzędnej Y' wszystkim trzem składowym $R'G'B'$ — otrzymany obraz jest czarno-szaro-biały.

Określony w roku 1976 układ współrzędnych CIELab, często spotykany w systemach zarządzania barwą, w założeniu miał zapewnić możliwość mierzenia subiektywnej różnicy barw za pomocą (euklidesowej) długości różnicy reprezentujących je wektorów[14]. Przejście między układami CIE XYZ a CIELab jest funkcją nieliniową, opisaną wzorami

$$L = 116\sqrt[3]{Y/Y_0} - 16, \quad a = 500\left(\sqrt[3]{X/X_0} - \sqrt[3]{Y/Y_0}\right), \quad b = 200\left(\sqrt[3]{Y/Y_0} - \sqrt[3]{Z/Z_0}\right),$$

ze stałymi $X_0 = 94.81$, $Y_0 = 100$, $Z_0 = 107.3$. Współrzędna L, która wyraża jasność barwy, jest nazywana luminancją, ale nie jest to fizyczna luminancja zdefiniowana w fotometrii. Współrzędne a i b razem określają odcień i nasycenie barwy.

C.5. Układy z subtraktywnym mieszaniem barw

Papier, na którym jest coś wydrukowane, sam światła nie emituje, a tylko je odbija. Światło przechodzi przez warstwy pigmentów w farbach drukarskich lub tonerach, które działają jak filtry pochłaniające. Współrzędne CMY (*Cyan* — turkusowy, *Magenta* — purpurowy, *Yellow* — żółty) opisują stopień pochłaniania światła przez pigmenty w barwach dopełniających barwy czerwoną, zieloną i niebieską. Jeśli zatem $C = M = Y = 0$ i oglądamy kartkę w świetle białym, to widzimy odbite od niej światło białe. Jeśli $C = 0$, $M = Y = 1$, to w świetle odbitym najmniej stłumione pozostaną fale długie (czyli światło czerwone), a jeśli $C = M = Y = 1$, to pigmenty pochłaniają światło o wszystkich długościach i dany punkt na papierze jest czarny.

W praktyce kolorowe pigmenty nie są idealnymi filtrami, a oprócz tego część światła odbija się od nich, nie przechodząc przez pozostałe pigmenty[15], co uniemożliwia otrzymanie

[14] co udało się w znacznym stopniu

[15] Dlatego w drukowaniu kolejność, w jakiej są nanoszone farby o poszczególnych kolorach, jest istotna.

całkowitej czerni we wspomniany wyżej sposób. Czerń można uzyskać za pomocą czwartej, czarnej farby (lub tonera). Wtedy opis koloru jest wektorem czterech współrzędnych, nazwanych CMYK; litera K jest wzięta ze słowa *blacK*. Mając współrzędne R, G, B koloru piksela (z przedziału $[0,1]$), można obliczyć współrzędne $C' = 1-R$, $M' = 1-G$, $Y' = 1-B$, a następnie przyjąć $K = a\max\{C', M', Y'\}$, z pewną stałą $a \in (0,1]$ i obliczyć $C = C' - K$, $M = M' - K$, $Y = Y' - K$; jest to najprostszy sposób przygotowania obrazu do druku. W rzeczywistości proces ten jest znacznie bardziej skomplikowany. Drukarki „fotograficzne" drukują obrazy przy użyciu większej liczby kolorowych atramentów (np. sześciu), a algorytmy zamiany kolorów na ilości atramentu nanoszone na poszczególne punkty na papierze, uwzględniające wzajemne oddziaływanie pigmentów, są bardzo skomplikowane (i tajne). W poligrafii proces przygotowania kolorowych obrazów do druku wysokiej jakości obejmuje weryfikację wydruków próbnych i wprowadzanie korekt przez zajmujących się tym ekspertów.

C.6. Układy HSV i HSL

Opisane wyżej układy współrzędnych (z wyjątkiem CIELab) są blisko związane z technologią wyświetlania lub drukowania kolorowych obrazów, ale wygoda programisty nie jest tożsama z wygodą użytkowników programu. Artyście grafikowi wygodniej jest posługiwać się współrzędnymi opisującymi jasność, odcień i nasycenie barwy.

Nazwy współrzędnych w układzie HSV pochodzą od słów *Hue* (odcień), *Saturation* (nasycenie) i *Value* (wartość). Przejście od układu RGB[16] do HSV opisują następujące wzory:

$$V = \max\{R,G,B\}, \quad a = V - \min\{R,G,B\},$$

$$H = \begin{cases} 0 & \text{jeśli } a = 0, \\ 60(G-B)/a & \text{jeśli } V = R, \\ 120 + 60(B-R)/a & \text{jeśli } V = G, \\ 240 + 60(R-G)/a & \text{jeśli } V = B, \end{cases}$$

$$S = \begin{cases} 0 & \text{jeśli } V = 0, \\ a/V & \text{jeśli } V \neq 0. \end{cases}$$

Współrzędna H jest tradycyjnie mierzona w stopniach[17] i przyjmuje wartości z przedziału $[-60^\circ, 300^\circ)$. Wartości współrzędnej S należą do przedziału $[0,1]$, przy czym jeśli $S = 0$, to barwa jest szara, a odcień jest nieokreślony, choć na podstawie podanego wyżej wzoru przyjmuje się $H = 0$. Współrzędna $V = 0$ odpowiada czerni, $V = 1$ oznacza zaś maksymalny poziom barwy o danym odcieniu i nasyceniu — zatem tę samą współrzędną V ma światło białe ($R = G = B = 1$) i światło niebieskie ($R = G = 0$, $B = 1$), które dla ludzkich oczu jest znacznie ciemniejsze.

Układ współrzędnych HSL (*Hue*, *Saturation*, *Lightness*) uwzględnia fakt, że światło białe jest jaśniejsze niż każda z jego składowych. W tym układzie światło białe ($R = G = B = 1$) ma

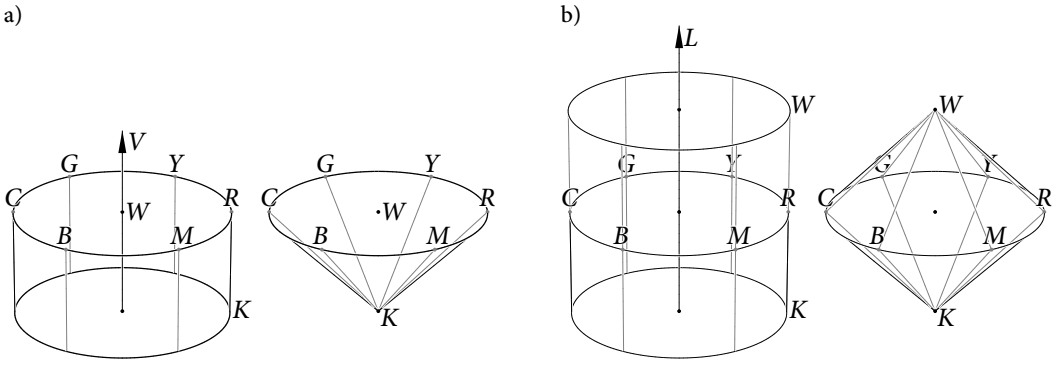
[16] Tu zazwyczaj jest układ RGB określony przez monitor komputera, na którym działa program.

[17] W programach ja bym używał radianów.

współrzędną $L = 2$, czystym barwom składowym (np. $R = 1$, $G = B = 0$) oraz ich barwom dopełniającym (np. $R = 0$, $G = B = 1$) odpowiada natomiast współrzędna $L = 1$, przy czym współrzędną $L = 1$ ma też barwa szara $r = g = b = 0.5$. W przejściu od układu RGB do HSL należy znaleźć liczby V i a i współrzędną H tak jak w przejściu do układu HSV, a następnie obliczyć

$$L = 2V - a,$$

$$S = \begin{cases} a/L & \text{jeśli } L \leqslant 1, \\ a/(2-L) & \text{jeśli } L > 1. \end{cases}$$



Rysunek C.6. Bryły barw: a) w układzie HSV, b) w układzie HSL

Bryły barw w układach HSV i HSL są walcami, choć często bywają przedstawiane jako stożek obrotowy albo dwa takie stożki zestawione podstawami (rys. C.6). Dolna podstawa walca lub wierzchołek dolnego stożka w obu przypadkach reprezentuje czerń. W układzie HSL górna podstawa walca lub wierzchołek górnego stożka odpowiada bieli. Współrzędna S ma wartość 0 na osi walców lub stożków i 1 na ich powierzchniach bocznych.

D

Dżojstik w aplikacjach X Window

Wprawdzie specyfikacja [11] systemu X Window X11R7 definiuje sposób komunikacji między aplikacją a dżojstikiem (który nadaje komunikaty takie jak klawiatura, tj. `KeyPress` i `KeyRelease`), ale jest z nią kłopot. Skonfigurowanie urządzenia wymaga uprawnień administratora i sporych umiejętności[1]. Jeśli aplikacja ma być rozpowszechniana, to nie możemy tego wymagać od użytkowników. Dlatego biblioteki FreeGLUT i GLFW mają własne sposoby komunikowania się z nietypowymi urządzeniami wejściowymi, w tym z dżojstikami. Sposoby te realizują procedury, które wywoływane w regularnych odstępach czasu „odpytują" dżojstik o jego stan (tj. o stan przycisków i kąty obrotu drążka wokół wszystkich osi).

Jądra systemów operacyjnych mają wbudowane sterowniki dżojstików, dzięki którym aplikacje mogą odczytywać odpowiednie informacje. W tym dodatku przedstawiam procedury współpracujące ze sterownikiem systemu Linux. Napisałem je na podstawie lektury kodu źródłowego biblioteki FreeGLUT. Są tu dwa zestawy procedur. Pierwszy zestaw może być używany podobnie, jak procedury z bibliotek FreeGLUT i GLFW — w regularnych odstępach czasu trzeba „pytać" dżojstik o jego stan i odpowiednio reagować na zmiany tego stanu. Drugi zestaw procedur działa w ten sposób, że po otwarciu komunikacji z dżojstikiem aplikacja będzie otrzymywać przekazane przez system X Window komunikaty o każdej zmianie stanu dżojstika, dzięki czemu nie musi co chwila do niego „zaglądać".

D.1. Aktywne sprawdzanie

Makrodefinicje w liniach 1–3 na listingu D.1 określają maksymalną liczbę obsługiwanych jednocześnie dżojstików, maksymalną liczbę osi dżojstka i maksymalną długość napisu, który jest nazwą urządzenia nadaną przez producenta. Kolejne sześć makrodefinicji wprowadza nazwy zdarzeń generowanych przez dżojstik, odpowiednio nic, inicjalizacja osi, inicjalizacja przycisku, zmiana kąta obrotu drążka wokół osi, naciśnięcie lub zwolnienie przycisku i odłączenie (wyjęcie wtyczki) dżojstika.

[1] Mi się nie udało: postępując (chyba) zgodnie ze specyfikacją, zepsułem w swoim komputerze poprawne działanie myszy i klawiatury, a gdy je naprawiłem, wszelka myśl o dalszych próbach była mi obca.

Listing D.1. Nagłówki procedur do aktywnego sprawdzania stanu dżojstików

C

```
#define MAX_JOY              8
#define MAX_JOY_AXES        16
#define MAX_JOY_NAMELENGTH 128

#define JOY_EVENT_NONE   0
#define JOY_INIT_AXIS    1
#define JOY_INIT_BUTTON  2
#define JOY_EVENT_AXIS   3
#define JOY_EVENT_BUTTON 4
#define JOY_EVENT_OFF    5

typedef struct JoyState {
    int          event;                /* rodzaj zdarzenia */
    int          number;               /* numer osi lub przycisku */
    unsigned int btnmask;              /* stan wszystkich przycisków */
    float        axpos[MAX_JOY_AXES];  /* kąty obrotów wokół osi */
  } JoyState;
```

Procedury obsługi dżojstika są przedstawione na listingu D.2. Na początku działania aplikacja powinna wywołać procedurę `InitJoysticks`. Rozpoczęcie komunikacji wykonuje procedura `OpenJoystick`; jej pierwszy parametr jest numerem dżojstika, z którym aplikacja chce nawiązać współpracę. Dżojstik może być podłączony w trakcie pracy komputera; system operacyjny, gdy to zauważy, nada mu kolejny numer, 0, 1 itd. i utworzy plik dający aplikacjom dostęp do urządzenia. Jeśli dżojstik jest podłączony (tj. jego wtyczka tkwi w gniazdku USB), to wartość powrotna procedury jest niezerowa. Ostatnie trzy parametry wskazują tablicę, do której trafi nazwa, oraz zmienne, którym będą przypisane liczby przycisków i osi urządzenia. Na pożegnanie dżojstika aplikacja powinna wywołać procedurę `CloseJoystick`.

Stan dżojstika odczytuje procedura `ReadJoystick`, której wartość powrotna jest niezerowa, jeśli następuje inicjalizacja lub jeśli od ostatniego wywołania użytkownik spowodował dżojstikiem jakieś zdarzenie. Dokładna informacja jest przekazywana w strukturze typu `JoyState` wskazywanej przez parametr `jst`. Wartość pola `event` wskazuje rodzaj ostatniego zdarzenia (w użyciu są makrodefinicje w liniach 5–10 na listingu D.1). Po nawiązaniu komunikacji z dżojstikiem sterownik przekazuje dla każdej osi i dla każdego przycisku jeden komunikat opisujący początkowy kąt lub informujący, czy przycisk jest w tym momencie naciśnięty. Dalsze komunikaty opisują zmiany stanów początkowych.

Pole `number` struktury `JoyState` zawiera numer przycisku albo osi. Poszczególne bity pola `btnmask` opisują stan wszystkich przycisków (`1` — przyciśnięty, `0` — zwolniony), a w tablicy `axpos` są przechowywane bieżące kąty obrotów drążka wokół wszystkich osi. Są to liczby z przedziału $[-1, 1]$, przy czym końce przedziału odpowiadają minimalnym i maksymalnym kątom obrotu dla każdej osi.

Możemy teraz zajrzeć do środka przedstawionych wyżej procedur. Zmienna `js` jest tablicą, której elementy — struktury typu `MyJoystick` — opisują poszczególne urządzenia.

Listing D.2. Procedury odczytujące stan dżojstika

C

```c
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <linux/joystick.h>

#include "ajoystick.h"

#if defined(JS_VERSION) && JS_VERSION >= 0x010000
#define JOY_AXIS_RANGE   32767.0

typedef struct {
    int              fd;
    char             fname[16];
    char             jname[MAX_JOY_NAMELENGTH];
    int              buttons, axes;
    struct js_event ev;
    JoyState         jst;
  } MyJoystick;

static MyJoystick js[MAX_JOY];

void InitJoysticks ( void )
{
  int i;

  for ( i = 0; i < MAX_JOY; i++ )
    js[i].fd = -1;
} /*InitJoysticks*/

char OpenJoystick ( int jsn, char *jname, int *buttons, int *axes )
{
  unsigned char u;

  if ( jsn < 0 || jsn >= MAX_JOY )
    return false;  /* błędny parametr */
  sprintf ( js[jsn].fname, "/dev/input/js%d", jsn );
  if ( (js[jsn].fd = open ( js[jsn].fname, O_RDONLY | O_NONBLOCK )) < 0 )
    return false;  /* nie ma takiego urządzenia */
  ioctl ( js[jsn].fd, JSIOCGBUTTONS, &u );
  js[jsn].buttons = u;
  ioctl ( js[jsn].fd, JSIOCGAXES, &u );
  js[jsn].axes = u;
  ioctl ( js[jsn].fd, JSIOCGNAME( sizeof(js[jsn].jname) ), js[jsn].jname );
```

```
46:   if ( jname )   strncpy ( jname, js[jsn].jname, MAX_JOY_NAMELENGTH );
47:   if ( buttons ) *buttons = js[jsn].buttons;
48:   if ( axes )    *axes = js[jsn].axes;
49:   return true;
50: } /*OpenJoystick*/
51:
52: char CloseJoystick ( int jsn  )
53: {
54:   if ( js[jsn].fd >= 0 ) {
55:     close ( js[jsn].fd );
56:     js[jsn].fd = -1;
57:     return true;
58:   }
59:   return false;
60: } /*CloseJoystick*/
61:
62: char ReadJoystick ( int jsn, JoyState *jst )
63: {
64:   ssize_t nbytes;
65:   int     n;
66:
67:   if ( js[jsn].fd < 0 ) return false;
68:   jst->event = JOY_EVENT_NONE;
69:   errno = 0;
70:   nbytes = read ( js[jsn].fd, &js[jsn].ev, sizeof(struct js_event) );
71:   if ( nbytes < 0 ) {
72:     if ( errno == ENODEV || errno == EBADF ) {
73:       CloseJoystick ( jsn );
74:       jst->event = js[jsn].jst.event = JOY_EVENT_OFF;
75:       return true;
76:     }
77:     else
78:       return true;
79:   }
80:   switch ( js[jsn].ev.type & ~JS_EVENT_INIT ) {
81: case JS_EVENT_AXIS:
82:     if ( (n = js[jsn].ev.number) < js[jsn].axes )
83:       jst->axpos[n] = js[jsn].jst.axpos[n] =
84:           (float)js[jsn].ev.value/JOY_AXIS_RANGE;
85:     jst->event = js[jsn].jst.event =
86:        js[jsn].ev.type & JS_EVENT_INIT ? JOY_INIT_AXIS : JOY_EVENT_AXIS;
87:     jst->number = js[jsn].jst.number = n;
88:     return true;
89: case JS_EVENT_BUTTON:
90:     if ( !js[jsn].ev.value )
91:       jst->btnmask = js[jsn].jst.btnmask &= ~(1 << js[jsn].ev.number);
92:     else
```

```
93:       jst->btnmask = js[jsn].jst.btnmask |= 1 << js[jsn].ev.number;
94:     jst->event = js[jsn].jst.event =
95:        js[jsn].ev.type & JS_EVENT_INIT ? JOY_INIT_BUTTON : JOY_EVENT_BUTTON;
96:     jst->number = js[jsn].jst.number = js[jsn].ev.number;
97:     return true;
98: default:
99:     return false;
100:   }
101: } /*ReadJoystick*/
102: #else
103: #error "No suitable joystick driver"
104: #endif
```

Dla podłączonych dżojstików system Linux tworzy pliki o nazwach `/dev/input/js0`, `/dev/input/js1` itd. Wejście z dżojstika aplikacja będzie czytać z takiego pliku. Musi ona do tego używać procedur składających się na najbardziej „niskopoziomowy" interfejs wejścia/wyjścia w systemie Linux[2].

Otwarty (do czytania przez aplikację) plik jest identyfikowany przez tzw. **deskryptor pliku** — nieujemną liczbę całkowitą pamiętaną w polu `fd` struktury `MyJoystick`. Procedura `OpenJoystick` w linii 38 tworzy nazwę pliku dżojstika, a w linii 39 otwiera plik przy użyciu procedury `open`, której wartość powrotna jest deskryptorem pliku. Drugi parametr tej procedury określa, że plik jest otwierany tylko do czytania i ma to być **operacja nieblokująca** — w razie braku danych do odczytania w pliku procedura czytająca ma nie czekać na ich pojawienie się.

Trzy wywołania procedury `ioctl` w liniach 41, 43 i 45 zadają dżojstikowi pytania o liczby jego przycisków i osi i o nazwę. W liniach 46–48 informacje te są przekazywane aplikacji, jeśli jest nimi zainteresowana, tj. jeśli przekazała niepuste wskaźniki miejsc, w których te informacje mają być zapisane.

Działanie procedur `InitJoysticks` i `CloseJoystick` chyba nie wymaga objaśnień. Zobaczmy zatem, jak działa procedura `ReadJoystick`. W linii 70 procedura próbuje przeczytać ustaloną liczbę bajtów, z których składa się przekazany przez sterownik opis jednego zdarzenia wygenerowanego przez dżojstik. Operacja czytania może się nie powieść i wtedy zmienna `errno` zadeklarowana w systemowym pliku nagłówkowym `errno.h` otrzymuje niezerową wartość określającą rodzaj błędu. Błędy `EBADF` (niepoprawny plik) i `ENODEV` (urządzenie nieobecne) są traktowane jak zawiadomienie, że czytanie stało się niemożliwe (czego prawdopodobną przyczyną jest odłączenie dżojstika).

Jeśli nie udało się przeczytać opisu zdarzenia, ale wystąpił błąd inny niż wymienione wyżej (`EAGAIN` — spróbuj ponownie), to następuje powrót z procedury — nic z dżojstikiem się nie wydarzyło i procedura nie czeka, aż się wydarzy. Dane przeczytane do zdefiniowa-

[2] Interfejs, którego elementami są wskaźniki do struktur typu `FILE` i procedury `fopen`, `fclose`, `fread`, `fwrite`, `fscanf` i `fprintf`, jest „wysokopoziomowy" — taki sam we wszystkich systemach operacyjnych. W systemie Linux interfejs ten ukrywa deskryptory plików i procedury `open`, `close` i `read`, których tu musimy używać, bo potrzebujemy „rozmawiać" z urządzeniem, a operacja czytania ma być nieblokująca.

nej w systemowym pliku nagłówkowym `linux/joystick.h` struktury `js_event` opisują zdarzenie, które nastąpiło. Jeśli jest to obrót drążka, to w liniach 83–84 jest obliczany obecny kąt jego obrotu (sterownik podaje liczbę całkowitą z przedziału $[-32767, 32767]$). Jeśli zdarzenie dotyczy przycisku, to w linii 91 lub 93 odpowiadający przyciskowi bit w polu `btnmask` otrzymuje aktualną wartość.

Aplikacja, wywołując procedurę `ReadJoystick`, może otrzymać wiadomość o odłączeniu dżojstika, ale nie otrzyma wiadomości o jego podłączeniu. Aby wznowić współpracę z dżojstikiem po jego ponownym podłączeniu, aplikacja musi znów wywołać procedurę `OpenJoystick` (sama musi jakoś zdecydować, kiedy to zrobić).

D.2. Komunikacja za pośrednictwem systemu X Window

Opisane w podrozdziale D.1 procedury działają z pominięciem systemu X Window. Teraz przedstawię rozwiązanie, w którym o zdarzeniach spowodowanych przez dżojstiki aplikacja dowiaduje się, otrzymując komunikaty od tego systemu. Zwalnia ją to od aktywnego sprawdzania stanu dżojstików.

Będą tu w użyciu makrodefinicje z listingu D.1, a ponadto aplikacja *może* używać przedstawionej tam struktury `JoyState` do przechowywania informacji o stanie dżojstika lub dżojstików — jeśli zadeklaruje tablicę takich struktur i będzie w niej skrupulatnie zapisywać informacje z komunikatów. Informacje są przekazywane w komunikatach `ClientMessage`, przy czym pole `mesage_type` struktury `XClientMessageEvent` tych komunikatów ma wartość zmiennej `aJoystick` — wartość ta jest atomem zarezerwowanym w systemie X Window przez procedurę `xInitJoysticks`. Zmienna `aJoystick` musi być widoczna dla aplikacji, aby ta mogła ją odczytywać. Ponadto procedury obsługi dżojstika odwołują się do zmiennej `xdisplay`, która identyfikuje serwer X Window.

Listing D.3. Struktura komunikatu od dżojstika

C

```
1:  typedef struct JoyMessage {
2:      char          jsn;       /* numer dżojstika */
3:      char          msg;       /* identyfikator komunikatu */
4:      char          number;    /* numer przycisku lub osi */
5:      char          pressed;   /* czy przycisk został naciśnięty? */
6:      float         angle;     /* kąt obrotu osi */
7:      unsigned int  btnmask;   /* maska bitowa przycisków */
8:      char          pad[8];    /* dopełnienie do 20 bajtów */
9:    } JoyMessage;
10:
11: extern Display *xdisplay;
12: extern Atom    aJoystick;
```

Listing D.4 przedstawia procedury. Aplikacja po otwarciu komunikacji z serwerem powinna wywołać procedurę `xInitJoysticks` w celu wykonania niezbędnych przygotowań, a następnie dla każdego dżojstika, od którego zamierza odbierać komunikaty, ma wywołać procedurę `xOpenJoystick`. Zakończenie współpracy z dżojstikiem następuje przez wywo-

łanie procedury `xCloseJoystick` lub przez odłączenie dżojstika (o czym aplikacja zostanie poinformowana za pomocą komunikatu `JOY_EVENT_OFF`).

W opisanym tu rozwiązaniu założyłem, że procedura czytająca plik urządzenia w razie braku danych do odczytania ma czekać na ich pojawienie się. Zatem operacja czytania *ma być* blokująca, dzięki czemu nie będzie absorbującego procesor aktywnego sprawdzania, czy są już jakieś dane. Ale niedopuszczalne jest też „zawiśnięcie" aplikacji na operacji czytania. Sposobem poradzenia sobie z tym problemem jest utworzenie dla każdego podłączonego dżojstika osobnego wątku obliczeniowego działającego współbieżnie z głównym wątkiem aplikacji. Wątek dżojstika będzie czekał w procedurze czytania do chwili pojawienia się danych. Po powrocie z tej procedury wątek wyśle do wskazanego mu okna komunikat, po czym znów zapadnie się w drzemkę w oczekiwaniu na kolejne dane.

Do realizacji wątków użyłem biblioteki pthread opisanej w podręczniku [20], do którego odsyłam Czytelników chcących ją poznać. Aby użyć opisanych tu procedur, do listy bibliotek dołączanych do aplikacji trzeba dodać bibliotekę pthread, dopisując opcję `-lpthread` w odpowiednim miejscu pliku `Makefile`. Jeszcze jedno: wielowątkowa aplikacja systemu X Window powinna *przed* nawiązaniem komunikacji z serwerem (czyli przed wywołaniem procedury `XOpenDisplay`) wykonać instrukcję `XInitThreads ();` (zobacz listing 3.6).

Struktura `MyXJoystick` zawiera opis jednego dżojstika; tablica `js` zawiera tyle takich struktur, ile maksymalnie dżojstików ma móc jednocześnie obsługiwać aplikacja. Pola `fd` i `fname` zawierają deskryptor i nazwę pliku dżojstika. Pole `jsn` jest numerem dżojstika — jest to indeks elementu tablicy `js` przechowywany w tej strukturze dla uproszczenia kodu. W polach `buttons` i `axes` są pamiętane liczby przycisków i osi. Pole `btnmask` jest maską bitową stanu przycisków. Pole `window` jest identyfikatorem okna, do którego mają być kierowane komunikaty, a pole `thread` jest identyfikatorem wątku dżojstika.

Pierwsze dwa parametry procedury `xOpenJoystick` to identyfikator okna, do którego wątek dżojstika ma wysyłać komunikaty, oraz numer dżojstika. Pozostałe parametry są wskaźnikami zmiennych, w których ma być zapamiętana nazwa urządzenia i liczby jego przycisków i osi. Początkowe instrukcje, nawiązujące kontakt z dżojstikiem, są podobne jak w procedurze `OpenJoystick` na listingu D.2; w linii 113 jest tworzona nazwa pliku dżojstika, który jest otwierany w linii 114. Drugi parametr procedury `open` określa otwieranie tylko do czytania (`O_RDONLY`), ale bit, któremu można by nadać wartość `1` za pomocą makra `O_NONBLOCK`, ma wartość `0`, bo teraz operacja czytania ma być blokująca.

W liniach 116–122 w polach odpowiedniego elementu (struktury `MyXJoystick`) są zapamiętywane potrzebne informacje, w tym także informacje odczytane z urządzenia za pomocą procedury `ioctl`, które w liniach 123–125 są też udostępniane aplikacji.

W liniach 126–129 jest uruchamiany wątek obliczeniowy dżojstika. W zmiennej `attr` są zapisywane atrybuty wątku, przy czym większość z nich otrzymuje domyślne wartości (w szczególności wątek będzie miał minimalną określoną przez bibliotekę pthread wielkość stosu maszynowego, który i tak jest aż nadto pojemny). Wątek ma być **odczepiony** (*detached*) od głównego wątku aplikacji[3]. Wątek utworzony przez procedurę `pthread_create` w linii 128 natychmiast przystępuje do pracy; wykonuje on opisaną dalej procedurę `xJoyThread`.

[3] Po objaśnienia proszę zajrzeć do [20].

Listing D.4. Procedury obsługi dżojstika w X Window

C

```
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <linux/joystick.h>
#include <pthread.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include "xjoystick.h"

#if defined(JS_VERSION) && JS_VERSION >= 0x010000
#define JOY_AXIS_RANGE   32767.0

typedef struct {
    int             fd;
    char            fname[16];
    char            jsn;
    int             buttons, axes;
    unsigned int    btnmask;
    Window          window;
    pthread_t       thread;
  } MyXJoystick;

static MyXJoystick js[MAX_JOY];
Atom               aJoystick;

void xInitJoysticks ( void )
{
  int i;

  memset ( js, 0, MAX_JOY*sizeof(MyXJoystick) );
  for ( i = 0; i < MAX_JOY; i++ )
    js[i].fd = -1;
  aJoystick = XInternAtom ( xdisplay, "aJoystick", False );
} /*xInitJoysticks*/

static void PostJoystickEvent ( Window win, char jsn, char msg, char number,
                                char pressed, float ang, unsigned int btnmask )
{
  JoyMessage jmsg;

  jmsg.jsn = jsn;  jmsg.msg = msg;  jmsg.number = number;
  jmsg.pressed = pressed;  jmsg.angle = ang;  jmsg.btnmask = btnmask;
```

```
46:   memset ( jmsg.pad, 0, 8*sizeof(char) );
47:   PostClientMessageEvent ( win, aJoystick, 8, (void*)&jmsg );
48:   XFlush ( xdisplay );
49: } /*PostJoystickEvent*/
50:
51: static void *xJoyThread ( void *data )
52: {
53:   MyXJoystick *js;
54:   struct js_event ev;
55:   ssize_t     nbytes;
56:   int         number;
57:   char        msg;
58:
59:   js = (MyXJoystick*)data;
60:   for (;;) {
61:     errno = 0;
62:     nbytes = read ( js->fd, &ev, sizeof(struct js_event) );
63:     if ( nbytes < 0 ) {
64:       if ( errno == ENODEV || errno == EBADF ) {
65:         PostJoystickEvent ( js->window, js->jsn, JOY_EVENT_OFF,
66:                             0, 0, 0.0, 0 );
67:         if ( js->fd >= 0 ) {
68:           close ( js->fd );
69:           js->fd = -1;
70:         }
71:         pthread_exit ( NULL );
72:       }
73:       else
74:         continue;
75:     }
76:     switch ( ev.type & ~JS_EVENT_INIT ) {
77:   case JS_EVENT_AXIS:
78:       if ( (number = ev.number) < js->axes ) {
79:         msg = ev.type & JS_EVENT_INIT ? JOY_INIT_AXIS : JOY_EVENT_AXIS;
80:         PostJoystickEvent ( js->window, js->jsn, msg, number, 0,
81:                         (float)ev.value/JOY_AXIS_RANGE, 0 );
82:       }
83:       continue;
84:   case JS_EVENT_BUTTON:
85:       if ( (number = ev.number) < js->buttons ) {
86:         msg = ev.type & JS_EVENT_INIT ?
87:                 JOY_INIT_BUTTON : JOY_EVENT_BUTTON;
88:         if ( !ev.value )
89:           js->btnmask &= ~(1 << ev.number);
90:         else
91:           js->btnmask |= 1 << ev.number;
92:         PostJoystickEvent ( js->window, js->jsn, msg,
```

```
93:                             number, ev.value != 0, 0.0, js->btnmask );
94:       }
95:       continue;
96:   default:
97:       continue;
98:     }
99:   }
100:  return NULL;
101: } /*xJoyThread*/
102:
103: char xOpenJoystick ( Window window, int jsn,
104:                      char *jname, int *buttons, int *axes )
105: {
106:  unsigned char  u;
107:  pthread_attr_t attr;
108:  int            rc;
109:  char           name[MAX_JOY_NAMELENGTH];
110:
111:  if ( jsn < 0 || jsn >= MAX_JOY )
112:    return false;
113:  sprintf ( js[jsn].fname, "/dev/input/js%d", jsn );
114:  if ( (js[jsn].fd = open ( js[jsn].fname, O_RDONLY )) < 0 )
115:    return false;
116:  js[jsn].jsn = jsn;
117:  js[jsn].window = window;
118:  ioctl ( js[jsn].fd, JSIOCGBUTTONS, &u );
119:  js[jsn].buttons = u;
120:  ioctl ( js[jsn].fd, JSIOCGAXES, &u );
121:  js[jsn].axes = u;
122:  ioctl ( js[jsn].fd, JSIOCGNAME( MAX_JOY_NAMELENGTH ), name );
123:  if ( jname )   strncpy ( jname, name, MAX_JOY_NAMELENGTH );
124:  if ( buttons ) *buttons = js[jsn].buttons;
125:  if ( axes )    *axes = js[jsn].axes;
126:  pthread_attr_init ( &attr );
127:  pthread_attr_setdetachstate ( &attr, PTHREAD_CREATE_DETACHED );
128:  rc = pthread_create ( &js[jsn].thread, &attr, xJoyThread,
129:                        (void*)&js[jsn] );
130:  if ( rc ) {
131:    pthread_attr_destroy ( &attr );
132:    xCloseJoystick ( jsn );
133:    return false;
134:  }
135:  return true;
136: } /*xOpenJoystick*/
137:
138: char xCloseJoystick ( int jsn )
139: {
```

```
140:   if ( js[jsn].fd >= 0 ) {
141:     if ( js[jsn].thread ) {
142:       pthread_cancel ( js[jsn].thread );
143:       js[jsn].thread = 0;
144:     }
145:     close ( js[jsn].fd );
146:     js[jsn].fd = -1;
147:     return true;
148:   }
149:   return true;
150: } /*xCloseJoystick*/
151: #else
152: #error "No suitable joystick driver"
153: #endif
```

Jeśli jednak uruchomienie wątku zakończyło się niepowodzeniem, to w linii 133 ta przykra wiadomość zostaje przekazana aplikacji.

Parametr procedury `xJoyThread` (linie 51–101), wykonywanej przez wątek dżojstika, jest wskaźnikiem odpowiedniej struktury `MyXJoystick` w tablicy `js`. Procedura wykonuje nieskończoną pętlę, w której czyta plik dżojstika (czekając, ile trzeba, na dane w procedurze `read`). W razie błędu uniemożliwiającego dalszą współpracę z dżojstikiem (zobacz podrozdz. D.1) w linii 65 jest wysyłany komunikat o odłączeniu dżojstika, po czym wątek składa rezygnację z dalszego działania.

Uwaga: Zmienna `errno` jest jedna i wszystkie wątki, także główny wątek aplikacji, mają do niej dostęp, co jest potencjalnym źródłem błędów. Nie jest bowiem możliwe zidentyfikowanie przyczyny błędu, gdy wiele wątków w tym samym czasie wywołuje procedury systemowe mogące przypisać wartość zmiennej `errno`[4].

Po przeczytaniu danych następuje ich interpretacja: jeśli zgłoszone przez sterownik zdarzenie dotyczy osi, to są wykonywane instrukcje w liniach 78–82, a jeśli przycisku, to 85–94. Po obliczeniu kąta obrotu lub stanu przycisku i aktualnej maski bitowej naciśniętych przycisków (co jest wykonywane identycznie jak w procedurze `ReadJoystick` z listingu D.2) wywoływana jest procedura `PostJoystickEvent` (linie 39–49), która przygotowuje dane do umieszczenia w komunikacie i wywołuje procedurę `PostClientMessageEvent` z listingu 3.8, która z kolei ten komunikat wyśle. Sam fakt *wstawienia* (przez `XSendEvent`) komunikatu do kolejki systemu X Window nie wystarczy, aby ten komunikat od razu dotarł do adresata. Gdy wątek główny aplikacji, „zawieszony" w procedurze `XNextEvent`, czeka na komunikat, trzeba go odpowiednio „szturchnąć"[5]. Takie „szturchnięcie" wykonuje procedura `XFlush` wywołana w linii 48.

[4] Jedynym stuprocentowo poprawnym rozwiązaniem wydaje się zastąpienie wątków przez *procesy* (uruchamiane przy użyciu procedur `fork` i `exec`), ponieważ każdy proces ma własną zmienną `errno`, ale to rozwiązanie wydaje się cokolwiek siłowe.

[5] Co jest niepotrzebne, gdy główny wątek aplikacji sam do siebie wysyła komunikat: wkrótce *po* `XSendEvent` wywoła procedurę `XNextEvent`, która znalazłszy od razu komunikat, nie zacznie na niego czekać. Nie ma też tego problemu z komunikatami wysłanymi przez inne procesy, tj. inne działające w tym samym czasie programy. Tylko wątki tego samego procesu muszą specjalnie „szturchać".

Procedura `xCloseJoystick` może być wywołana przez aplikację, jeśli ta chce zakończyć współpracę z dżojstikiem bez proszenia użytkownika o wyciągnięcie wtyczki. Postanowienie o dymisji wręcza wątkowi procedura `pthread_cancel`, a oprócz tego trzeba zamknąć plik dżojstika i zapamiętać, że został zamknięty, przypisując polu `fd` wartość `-1`.

Listing D.5 przedstawia sposób odbierania przez aplikację komunikatów od dżojstika; analogiczną procedurę dla aplikacji biblioteki GLFW pokazałem na listingu 3.4. Inwencji Czytelników pozostawiam zamianę instrukcji wypisujących komunikaty do terminala (niezastąpionych podczas uruchamiania programu) na instrukcje powodujące zmiany reprezentacji obiektów i powodujące wykonywanie ich nowych obrazów.

Listing D.5. Procedura odbierająca komunikaty dżojstika

C

```
void MyJoystickEvent ( JoyMessage *jmsg )
{
  printf ( "joystick %d:", jmsg->jsn );
  switch ( jmsg->msg ) {
case JOY_INIT_AXIS:
    printf ( " init axis %d, ang = %f\n", jmsg->number, jmsg->angle );
    break;
case JOY_INIT_BUTTON:
    printf ( " init button %2d, %d, %8x\n", jmsg->number, jmsg->pressed,
             jmsg->btnmask );
    break;
case JOY_EVENT_AXIS:
    printf ( " axis %d, ang = %f\n", jmsg->number, jmsg->angle );
    break;
case JOY_EVENT_BUTTON:
    printf ( " button %2d, %d, %8x\n", jmsg->number, jmsg->pressed,
             jmsg->btnmask );
    break;
case JOY_EVENT_OFF:
    printf ( " off\n" );
    break;
  }
} /*MyJoystickEvent*/

void MyWinClientMessage ( XClientMessageEvent *ev )
{
  if ( ev->message_type == aJoystick )
    MyJoystickEvent ( (JoyMessage*)ev->data.b );
  else .... /* obsługa innych komunikatów ClientMessage */
} /*MyWinClientMessage*/
```

E

Rzutowanie nieliniowe

Przekształcenie przestrzeni trójwymiarowej na płaszczyznę, które zachowuje współliniowość każdej trójki punktów, jest rzutem równoległym albo perspektywicznym. Niekiedy są potrzebne inne sposoby rzutowania. Przedstawiam zatem rzuty panoramiczne (na rzutnię walcową, czyli rozwijalną) i rzuty na sferę (która nie jest rozwijalna), razem z propozycją sposobu ich realizowania w aplikacjach OpenGL-a.

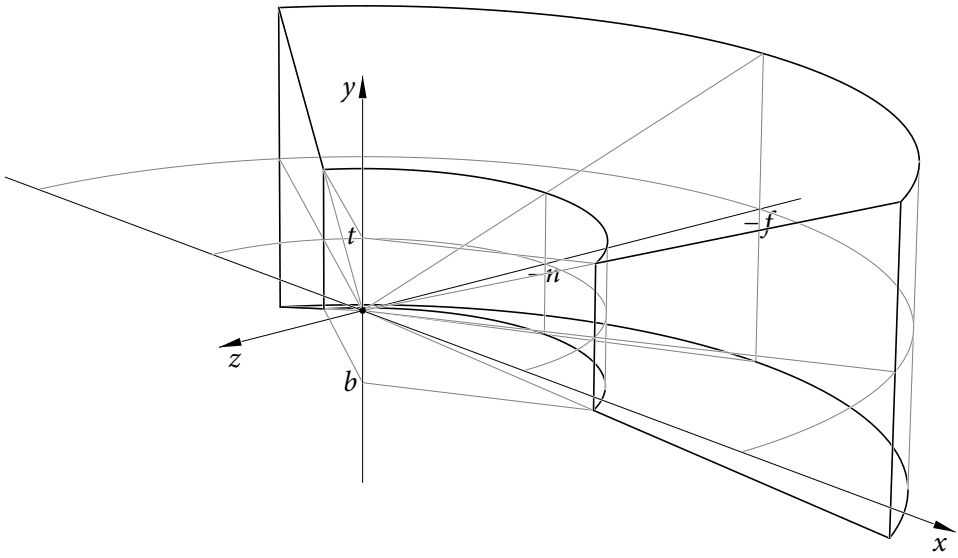
Odwzorowanie punktów danych w układzie współrzędnych modelu do układu kostki standardowej w rzutowaniu perspektywicznym lub równoległym jest złożeniem przejścia do układu świata, przejścia od układu świata do układu obserwatora i przejścia od układu obserwatora do układu kostki standardowej (czego dokładniejszy opis jest w rozdz. 6). Aby uzyskać każdy z rzutów opisanych niżej, trzeba wprowadzić przekształcenie nieliniowe między ostatnimi dwoma z tych przejść.

E.1. Panorama punktowa

Rzutnia dla panoramy jest fragmentem powierzchni walcowej, który po rozwinięciu jest prostokątem, odwzorowywanym następnie na klatkę w oknie. Obserwator znajduje się w punkcie na osi walca. Zatem bryła widzenia jest ograniczona sześcioma powierzchniami: walcowymi przednią i tylną (zakładamy, że rzutnia jest powierzchnią przednią), stożkowymi górną i dolną oraz dwiema płaszczyznami bocznymi, zobacz rysunek E.1.

Przyjąłem, że oś walcowej rzutni jest osią y układu obserwatora i wyznacza kierunek pionowy na obrazie w oknie. Wtedy bryłę widzenia w układzie obserwatora można opisać za pomocą pięciu parametrów. Dwa z nich, które oznaczymy n i f (*near* i *far*), są promieniami walców. Kolejne dwa, b i t (*bottom* i *top*), są współrzędnymi y punktów na dolnej i górnej krawędzi wycinka przedniego walca, który będzie odwzorowany w okno. Ostatni parametr, φ, jest kątem między bocznymi płaszczyznami bryły widzenia[1]; bryła ta jest symetryczna względem płaszczyzny yz układu obserwatora.

[1] Wszystkie kąty mierzę w radianach.



Rysunek E.1. Bryła widzenia panoramy punktowej

Długość dolnego i górnego brzegu przedniej ściany bryły widzenia (która po rozwinięciu jest szerokością prostokąta z obrazem) jest równa $n\varphi$. Aby zatem osiągnąć jednakowe skalowanie w pionie i poziomie na ekranie o współczynniku aspektu a dla klatki o szerokości w pikseli i wysokości h pikseli, trzeba spełnić następujący warunek:

$$aw : h = n\varphi : t - b.$$

Stąd dla ustalonej wysokości $t - b$ przedniej ściany bryły widzenia[2] należy przyjąć

$$\varphi = \frac{t-b}{n}\frac{aw}{h}.$$

Przekształcenie nieliniowe wstawione przed przejściem do układu kostki standardowej jest zamianą współrzędnych x, y, z na x', y', z', opisaną wzorami

$$x' = \operatorname{arc\,tg}\frac{x}{-z}, \quad y' = y/r, \quad z' = -r, \quad \text{w których } r = \sqrt{x^2+z^2}.$$

Przekształcając punkt reprezentowany za pomocą współrzędnych jednorodnych X, Y, Z, W, możemy użyć wzorów

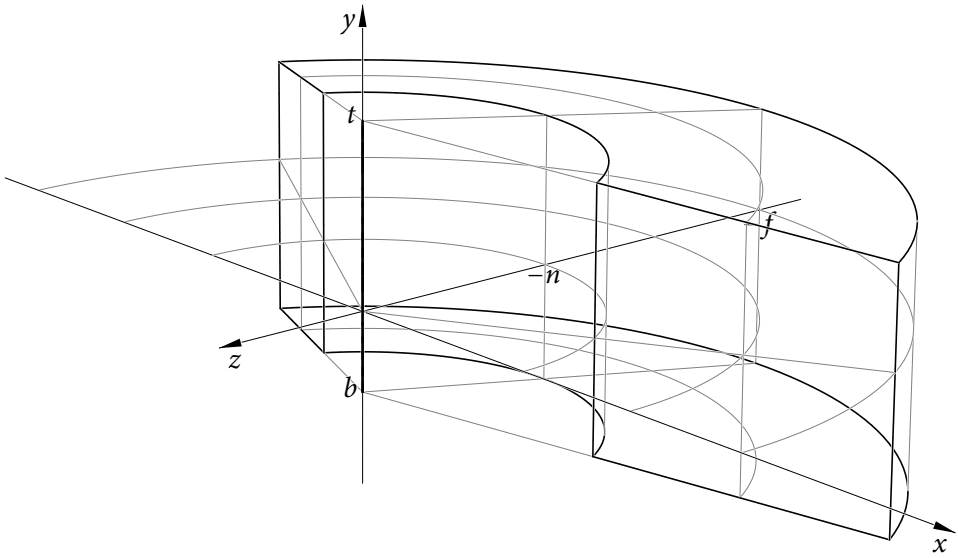
$$X' = \operatorname{arc\,tg}\frac{X}{-Z}, \quad Y' = Y/R, \quad Z' = -R/W, \quad W' = 1, \quad \text{podstawiając } R = \sqrt{X^2+Z^2}.$$

Przekształcenie to odwzorowuje bryłę widzenia na prostopadłościan. Macierz P przekształcającą go na kostkę standardową skonstruuje procedura `M4x4Orthof` (listing 6.2) wywołana z parametrami `left` $= -\varphi/2$, `right` $= \varphi/2$, `bottom` $= b/n$, `top` $= t/n$, `near` $= n$, `far` $= f$.

[2] Możemy ją wybrać tak samo jak dla rzutowania perspektywicznego w pierwszej aplikacji, zobacz s. 152.

E.2. Panorama linearna

Wszystkie proste łączące punkty w przestrzeni i ich obrazy na walcowej rzutni w opisanej wyżej panoramie przecinają się w jednym punkcie — położeniu obserwatora, stąd nazwa: **panorama punktowa**. Możemy określić takie rzutowanie na powierzchnię walca, w którym analogiczne proste przecinają oś walca pod ustalonym kątem; wtedy „środki rzutowania" zajmują odcinek na tej osi, w związku z czym takie odwzorowanie przestrzeni na płaszczyznę wypada nazwać **panoramą linearną**.



Rysunek E.2. Bryła widzenia panoramy linearnej

Bryła widzenia panoramy linearnej jest ograniczona dwoma walcami (wewnętrznym, czyli „przednim" i zewnętrznym, „tylnym"), dwiema powierzchniami stożkowymi („dolną" i „górną") i dwiema płaszczyznami bocznymi. Można ją opisać za pomocą sześciu parametrów: liczby n i f są promieniami walców, liczby b i t wyznaczają końce odcinka środków rzutowania na osi walców i stożków (osi y układu współrzędnych obserwatora), liczba φ jest miarą kąta między płaszczyznami bocznymi, a liczba ϑ określa kąt między prostymi rzutowania a płaszczyzną xz.[3] Ale potrzebny jest jeszcze jeden parametr — odległość d od osi walców obiektów, dla których skalowanie wymiarów poziomych i pionowych na obrazie ma być takie samo (zapewne przyjmiemy $n < d < f$). Rzutnia w tym przypadku jest walcem o promieniu d, którego rozwinięty i odwzorowany na klatkę fragment ma szerokość $d\varphi$ i wysokość $t - b$. W panoramie linearnej skalowanie wymiarów pionowych jest stałe, a poziomych jest odwrotnie proporcjonalne do tej odległości[4]. Dla klatki o wymiarach $w \times h$ pikseli (na ekra-

[3] Jeśli $\vartheta = 0$, to górna i dolna powierzchnia bryły widzenia są płaskie.

[4] W panoramie punktowej skalowanie *obu* osi zmienia się z tą odległością.

nie o współczynniku aspektu a) należy przyjąć

$$\varphi = \frac{t-b}{d}\frac{aw}{h}.$$

Przekształcenie nieliniowe współrzędnych kartezjańskich punktu danego w układzie obserwatora opiszemy wzorami

$$x' = \operatorname{arc\,tg}\frac{x}{-z}, \quad y' = y + r\operatorname{tg}\vartheta, \quad z' = -r, \quad r = \sqrt{x^2+z^2},$$

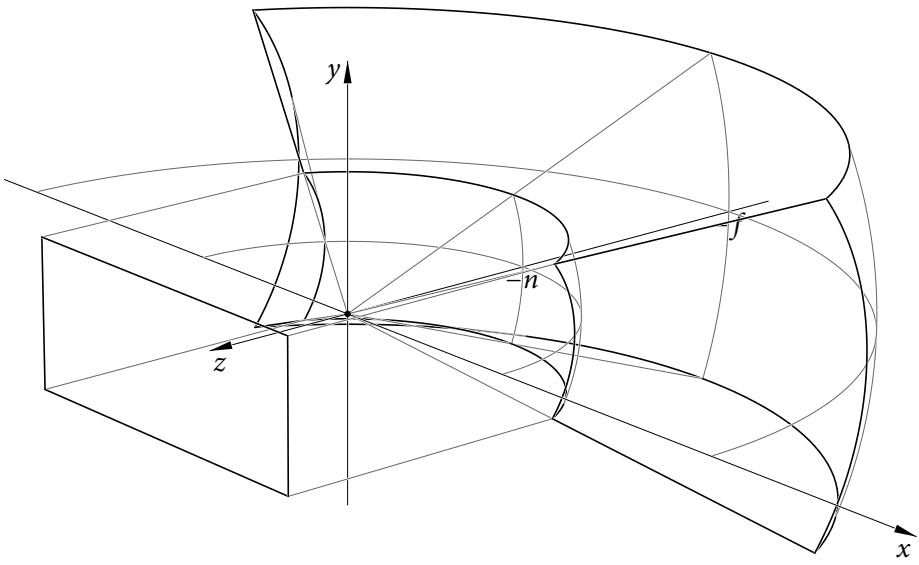
a równoważne przekształcenie współrzędnych jednorodnych jest takie:

$$X' = \operatorname{arc\,tg}\frac{X}{-Z}, \quad Y' = \big(Y + R\operatorname{tg}\vartheta\big)/W, \quad Z' = -R/W, \quad W' = 1, \quad R = \sqrt{X^2+Z^2}.$$

Przejście od tak obliczonych współrzędnych X', Y', Z', W' do układu kostki standardowej zapewni macierz P skonstruowana przez procedurę `M4x4Orthof` (listing 6.2) na podstawie parametrów `left` $= -\varphi/2$, `right` $= \varphi/2$, `bottom` $= b$, `top` $= t$, `near` $= n$, `far` $= f$, przy czym liczby b i t mają spełniać warunek opisany wcześniej; można przyjąć „symetrycznie” $t = -b = d\varphi h/(2aw)$.

E.3. Rzutowanie na sferę

Płaski obraz przestrzeni możemy otrzymać jako złożenie dwóch rzutów, z których pierwszy przekształca przestrzeń na sferę, a drugi jest rzutem równoległym wycinka tej sfery na prostokąt. W ten sposób powstają obrazy przypominające fotografie wykonane przez obiektywy typu „rybie oko”.



Rysunek E.3. Bryła widzenia w rzutowaniu na sferę

Aby opisać bryłę widzenia pokazaną na rysunku E.3, trzeba podać parametry n i f, które określają minimalną i maksymalną odległość jej punktów od obserwatora (czyli promienie sfery „wewnętrznej" i „zewnętrznej", wyznaczającej przednią i tylną ścianę bryły) i cztery kąty, oznaczone symbolami φ_l, φ_r, ϑ_b i ϑ_t, które muszą spełniać nierówności $-\pi/2 < \varphi_l < \varphi_r < \pi/2$ i $-\pi/2 < \vartheta_b < \vartheta_t < \pi/2$. Powierzchnie ograniczające bryłę widzenia z dołu, z góry i z boków są stożkami obrotowymi. Najczęściej przyjmiemy $\varphi_l = -\varphi_r$ i $\vartheta_b = -\vartheta_t$ i wtedy bryła widzenia będzie symetryczna względem płaszczyzn yz i xz.

Przekształcenie nieliniowe współrzędnych kartezjańskich punktu w układzie obserwatora możemy opisać wzorami

$$x' = \frac{x}{r}, \quad y' = \frac{y}{r}, \quad z' = -r, \quad r = \sqrt{x^2 + y^2 + z^2},$$

a wtedy współrzędne jednorodne trzeba przekształcić według wzorów

$$X' = \frac{X}{R}, \quad Y' = \frac{Y}{R}, \quad Z' = -R/W, \quad W' = 1, \quad R = \sqrt{X^2 + Y^2 + Z^2}.$$

W ten sposób otrzymujemy rzut środkowy przestrzeni na sferę jednostkową[5]. Aby otrzymać poprawne skalowanie w pionie i poziomie, przy dobieraniu kątów do wymiarów klatki trzeba spełnić proporcję

$$\frac{\sin\varphi_r - \sin\varphi_l}{aw} = \frac{\sin\vartheta_t - \sin\vartheta_b}{h}.$$

Przejście do układu kostki standardowej zapewni macierz P skonstruowana przez procedurę `M4x4Orthof`, której trzeba podać parametry `left` = $\sin\varphi_l$, `right` = $\sin\varphi_r$, `bottom` = $\sin\vartheta_b$, `top` = $\sin\vartheta_t$, `near` = n, `far` = f.

E.4. Rozdrabnianie w rzutowaniu nieliniowym

Listing E.1 przedstawia procedurę, która dokonuje jednego z opisanych wyżej przekształceń nieliniowych. Przekształcenie jest określone przez zawartość bloku zmiennych jednolitych `NLProjection`, którego pole `type` określa rodzaj przekształcenia, a pole `tantheta` ma wartość tg ϑ, potrzebną w panoramie linearnej. Wywoływane przez tę procedurę funkcje `atan` i `length` są opisane w podrozdziale 9.13.

Listing E.1. Procedura przekształceń nieliniowych

GLSL

```
#define PROJ_POINT_PANORAMA  1
#define PROJ_LINEAR_PANORAMA 2
#define PROJ_SPHERICAL       3

```

[5] Rysunek E.3 przedstawia sytuację, w której $n = 1$.

```
5: uniform NLProjection {
6:     int   type;
7:     float tantheta;
8:   } nlp;
9:
10: vec4 NonlinTransformation ( vec4 p )
11: {
12:   float R;
13:
14:   switch ( nlp.type ) {
15: case PROJ_POINT_PANORAMA:
16:     R = length ( p.xz );
17:     return vec4 ( atan ( -p.z, p.x ), p.y/R, -R/p.w, 1.0 );
18: case PROJ_LINEAR_PANORAMA:
19:     R = length ( p.xz );
20:     return vec4 ( atan ( -p.z, p.x ), (y+R*nlp.tantheta)/p.w, -R/p.w, 1.0 );
21: case PROJ_SPHERICAL:
22:     R = length ( p.xyz );
23:     return vec4 ( p.x/R, p.y/R, -R/p.w, 1.0 );
24: default: return p;
25:   }
26: } /*NonlinTransformation*/
```

W implementacji rzutowania nieliniowego wyprowadzenie i oprogramowanie wzorów, na podstawie których odbywa się rzutowanie, jest tą łatwą częścią zadania. Znacznie większy problem sprawia to, że obrazem odcinka w tych rzutach jest na ogół zakrzywiony łuk (w panoramach szerszych niż π to mogą być dwa łuki), a ponadto bryły widzenia dla tych rzutów nie są wypukłe i nie są wielościenne. Dlatego po przekształceniu wierzchołków prymitywu do narysowania — odcinka lub trójkąta — trzeba go obciąć (przynajmniej zgrubnie, do wielościanu otaczającego bryłę widzenia), a następnie rozdrobnić na dostatecznie krótkie odcinki lub dostatecznie małe trójkąty i poddać przekształceniu (nieliniowemu, a następnie przejściu do układu kostki standardowej) wierzchołki tych odcinków lub trójkątów. Przy tym zachodzi konieczność obcięcia i rozdrobnienia boków trójkątów w taki sposób, aby na obrazie powierzchni złożonej z wielu trójkątów nie było szczelin między trójkątami o wspólnych bokach.

Przykładowy szader geometrii na listingu E.2 zamienia odcinek na łamaną. W kwalifikatorze wyjścia takiego szadera (linia 6) trzeba podać maksymalną liczbę wierzchołków, które szader ten może wyprowadzić, przy czym w trakcie pracy może ich wyprowadzić mniej, jeśli mniejszy stopień rozdrobnienia łamanej wystarczy do otrzymania dostatecznej dokładności obrazu. Wierzchołki łamanej, tj. końce fragmentów rozdrobnionego odcinka, są przekształcane zgodnie z opisem wybranego rzutowania (perspektywicznego lub nieliniowego) do układu kostki standardowej. Szader na listingu nie zawiera procedury obcinania odcinka, koniecznej w zastosowaniach bardziej zaawansowanych niż tylko wykonanie ilustracji do jednej książki.

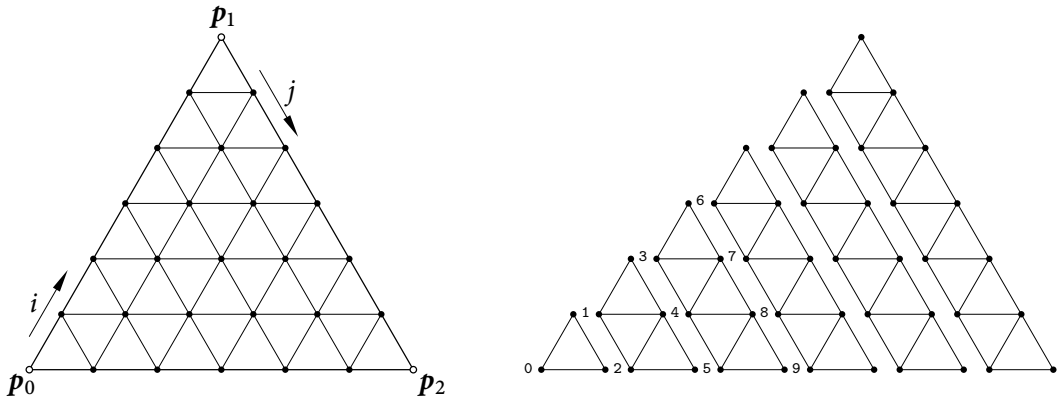
Listing E.2. Rozdrabnianie i rzutowanie nieliniowe odcinka

GLSL

```
#version 420

#define N 30

layout(lines) in;
layout(line_strip,max_vertices=N) out;

uniform TransBlock {
    mat4 mm, mmti, vm, pm;
    vec4 eyepos;
  } trb;

vec4 NonlinTransformation ( vec4 p ) { .... /* listing E.1 */ }

void main ( void )
{
  int  i;
  vec4 p;

  for ( i = 0; i < N; i++ ) {
    p = mix ( gl_in[0].gl_Position, gl_in[1].gl_Position,
              float(i)/float(N-1) );
    gl_Position = trb.pm*(NonlinTransformation ( trb.vm*(trb.mm*p) ));
    EmitVertex ();
  }
  EndPrimitive ();
} /*main*/
```

Listing E.3 przedstawia najprostszy sposób rozdrabniania trójkąta przez szader geometrii. Dla ustalonej liczby N powstaje z niego N taśm trójkątowych, złożonych odpowiednio z $1, 3, 5, \ldots, 2N-1$ trójkątów. Trójkąty te mają wiele wierzchołków wspólnych, zatem obliczenie przebiega w dwóch etapach: najpierw wierzchołki są wyznaczane, przekształcane i zapamiętywane w tablicach, z których zostaną wyprowadzone w drugim etapie. Całkowita liczba wierzchołków, czyli potrzebna długość tablic, jest równa $(N+1)(N+2)/2$, liczba wierzchołków wyprowadzonych jest natomiast większa, bo wierzchołki wspólne dla dwóch taśm są wyprowadzane dwukrotnie. Stąd liczba wierzchołków przekazywanych na wyjście, zadeklarowana w kwalifikatorze w linii 5 jest równa $N(N+2)$.

Rysunek E.4 przedstawia schemat podziału trójkąta. Wierzchołki jego fragmentów są ponumerowane parami liczb (i, j), przy czym $0 \leqslant j \leqslant i \leqslant N$. Makrodefinicja `TR_MAT_INDEX` zamienia taką parę na indeks do tablic jednowymiarowych, w których szader zapisuje wyniki obliczeń pierwszego etapu. Pokazany szader przekazuje następujące atrybuty: wektor współrzędnych jednorodnych położenia wierzchołka w układzie kostki standardowej (w tablicy `ppp`), wektor współrzędnych kartezjańskich w układzie świata (w tablicy `pos`), kolor (w tablicy `ccc`) i położenie obserwatora (w tablicy `epos`).



Rysunek E.4. Schemat podziału trójkąta na taśmy

Listing E.3. Rozdrabnianie i rzutowanie nieliniowe trójkąta

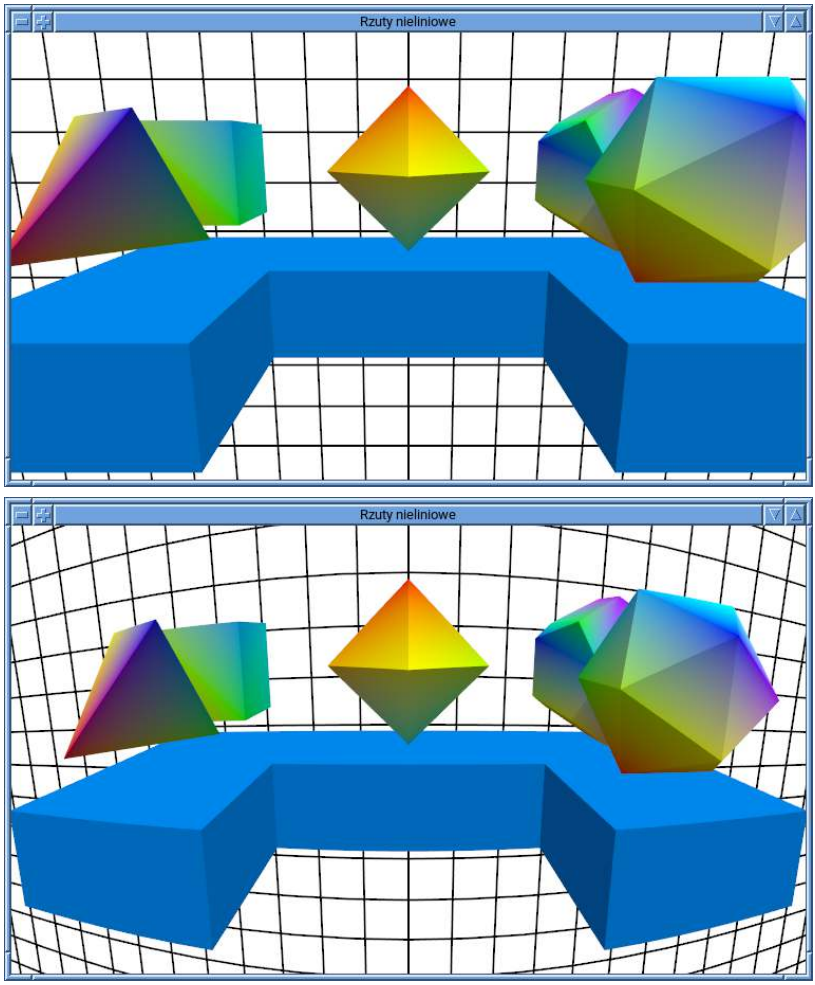
GLSL

```
#version 420

#define N   6
#define TN 28 /* (N+1)*(N+2)/2 */
#define NN 48 /* N*(N+2) */

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

in Vertex {
    vec4 Colour;
  } In[];

out FVertex {
    vec4 eyepos;
    vec4 Colour;
    vec3 Position;
    vec3 Normal;
  } Out;

uniform TransBlock { .... /* listing E.2 */ } trb;
uniform NLProjection { .... /* listing E.2*/ } nlp;

#define TR_MAT_INDEX(i,j) ( (i)*((i)+1)/2+(j) )

vec3 nv;
vec4 ppp[TN], ccc[TN], epos[TN];
vec3 pos[TN];

```

```
30: vec4 NonlinTransformation ( vec4 p ) { .... /* listing E.1 */ }
31:
32: void Emit ( int i, int j )
33: {
34:   int k;
35:
36:   k = TR_MAT_INDEX ( i, j );
37:   gl_Position = ppp[k];
38:   Out.eyepos = epos[k];
39:   Out.Position = pos[k];
40:   Out.Normal = nv;
41:   Out.Colour = ccc[k];
42:   EmitVertex ();
43: } /*Emit*/
44:
45: void main ( void )
46: {
47:   int   i, j, k;
48:   vec3  v1, v2;
49:   vec4  p, q;
50:   mat4  vmi;
51:   float x, y, z;
52:
53:   v1 = (trb.mm*(gl_in[1].gl_Position - gl_in[0].gl_Position)).xyz;
54:   v2 = (trb.mm*(gl_in[2].gl_Position - gl_in[0].gl_Position)).xyz;
55:   nv = normalize ( cross ( v1, v2 ) );
56:   if ( nlp.type == PROJ_LINEAR_PANORAMA )
57:     vmi = inverse ( trb.vm );
58:   for ( i = k = 0;  i <= N;  i++ ) {
59:     x = float(N-i)/float(N);
60:     for ( j = 0;  j <= i;  j++, k++ ) {
61:       y = float(i-j)/float(N);
62:       z = 1.0 - x - y;
63:       p = trb.mm * (x*gl_in[0].gl_Position + y*gl_in[1].gl_Position +
64:                     z*gl_in[2].gl_Position);
65:       pos[k] = p.xyz/p.w;
66:       q = NonlinTransformation ( trb.vm*p );
67:       ppp[k] = trb.pm * q;
68:       if ( nlp.type == PROJ_LINEAR_PANORAMA ) {
69:         q.x = q.z = 0.0;
70:         epos[k] = vmi*q;
71:       }
72:       else
73:         epos[k] = trb.eyepos;
74:       ccc[k] = x*In[0].Colour + y*In[1].Colour + z*In[2].Colour;
75:     }
76:   }
```

```
77:   for ( i = 1; i <= N; i++ ) {
78:     Emit ( i, 0 );
79:     for ( j = 0; j < i; j++ ) {
80:       Emit ( i-1, j );
81:       Emit ( i, j+1 );
82:     }
83:     EndPrimitive ();  /* koniec taśmy */
84:   }
85: } /*main*/
```

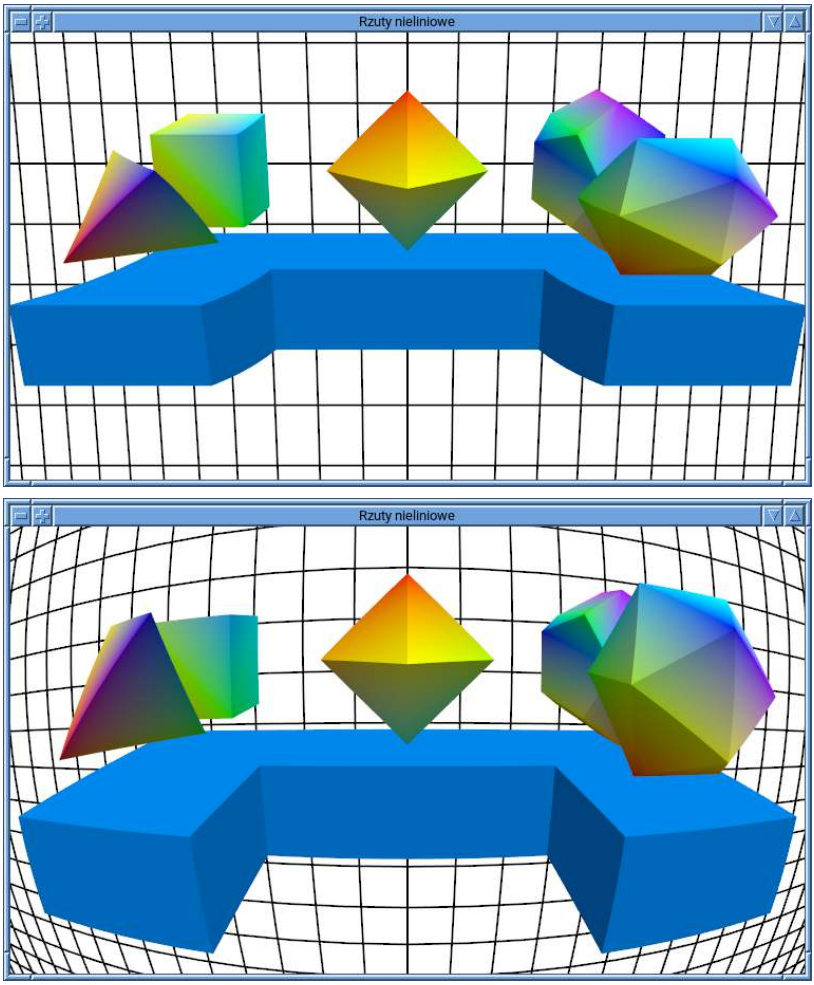
Ten ostatni atrybut wymaga wyjaśnienia. W przypadku rzutu perspektywicznego, panoramy punktowej i rzutowania na sferę położenie obserwatora, tj. środek rzutowania, jest



Rysunek E.5. Rzut perspektywiczny i panorama punktowa

jednym punktem, którego współrzędne są podane w zmiennej jednolitej `trb.eyepos` i już. W panoramie linearnej jest wiele położeń obserwatora, które tworzą odcinek (rys. E.2). Szader fragmentów potrzebuje znać położenie obserwatora do badania, czy poszczególne źródła światła znajdują się po tej samej stronie co obserwator, czy po przeciwnej, i do obliczania odblasków. Dlatego zestaw atrybutów punktu na trójkącie, przetwarzanego przez szader fragmentów, powinien zawierać położenie obserwatora, z którego został otrzymany obraz *tego punktu*.

Szader oblicza położenia obserwatora dla poszczególnych wierzchołków trójkąta, po czym są one interpolowane, tak jak każdy inny atrybut, w etapie rasteryzacji. Jeśli jest wybrana panorama linearna, to obliczenie położenia obserwatora polega na znalezieniu jego położenia w układzie współrzędnych obserwatora (co jest robione w liniach 66 i 69) i przejś-



Rysunek E.6. Panorama linearna (z kątem $\vartheta = 0$) i rzut na sferę

ciu do układu świata w linii 70. Dla pozostałych rzutów współrzędne położenia obserwatora są kopiowane z bloku zmiennych jednolitych `trb`.

W liniach 53–55 obliczany jest wektor normalny trójkąta. Zauważmy, że to jest wspólny wektor normalny wszystkich fragmentów tego trójkąta.

Adaptacyjne (dostosowane do wielkości i kształtu obrazu) rozdrabnianie trójkątów przez szader geometrii jest znacznie trudniejsze niż rozdrabnianie odcinków, bo zazwyczaj trójkąty przylegają bokami do innych trójkątów, tworząc powierzchnie, które nie powinny mieć szczelin. Dlatego adaptacyjne algorytmy rozdrabniania, dające poprawne wizualnie wyniki, są dość skomplikowane. Dla każdego boku trójkąta trzeba indywidualnie ustalić liczbę odcinków, na które ten bok zostanie podzielony, przy czym liczba ta *nie może* zależeć od trzeciego wierzchołka trójkąta. Po podzieleniu boków trzeba dokonać podziału wnętrza trójkąta, co może prowadzić do wygenerowania różnych zbiorów taśm trójkątowych i algorytm musi obsługiwać wszystkie możliwe (w ramach przyjętego ograniczenia stopnia rozdrobnienia) przypadki. Kolejną komplikacją jest konieczność wstępnego obcięcia trójkątów do bryły widzenia (lub wielościanu zawierającego bryłę widzenia), przy czym nie ma prostego sposobu wykorzystania „gotowych" algorytmów wbudowanych w etap obcinania prymitywów w potoku przetwarzania grafiki. Alternatywą jest skorzystanie z szaderów rozdrabniania zamiast geometrii, co jednak ma tę wadę, że wszystkie obiekty trzeba rysować jako płaty[6]. Ograniczywszy się do pokazania na rysunkach E.5 i E.6 obrazów otrzymanych (bez adaptacji) w rzucie perspektywicznym i w opisanych w podrozdziałach E.1–E.3 rzutach nieliniowych, pozostawiam te zagadnienia jako problemy otwarte.

[6] Perspektywę punktową i rzut na sferę można zrealizować znacznie prościej, za pomocą przetwarzania obrazu. Wystarczy wykonać obraz w rzucie perspektywicznym, a następnie nałożyć go jako teksturę na prostokąt, wprowadzając odpowiednią dystorsję. Nie da się jednak otrzymać w ten sposób obrazów z wieloma położeniami obserwatora, takich jak perspektywa linearna.

F

Rysowanie fraktali

F.1. Zbiór Mandelbrota

Wódz indiański znad Missisipi
Zapytany, ile ma tipi,
Zamiast skończyć kolację,
Zapadł się w medytację
I oznajmił: $e^{2i\pi}$. Howgh.

F.1.1. Liczby zespolone

Licząc się z tym, że do lektury tej książki przystąpią Czytelnicy nieznający jeszcze liczb zespolonych, zamieszczam niezbędne minimum informacji na ich temat. Liczby zespolone są parami liczb rzeczywistych, (a, b) lub (x, y), czyli wektorami współrzędnych kartezjańskich punktów na płaszczyźnie[1]. Często zapisuje się je w postaci $a + b\mathrm{i}$ lub $x + y\mathrm{i}$. Pierwszy element pary jest nazywany **częścią rzeczywistą**, a drugi **częścią urojoną** liczby zespolonej.

Dodawanie liczb zespolonych jest zwykłym dodawaniem wektorów w przestrzeni $\mathbb{R}^2$, a mnożenie jest określone wzorem

$$(a_1, b_1)(a_2, b_2) = (a_1a_2 - b_1b_2, a_1b_2 + b_1a_2). \tag{F.1}$$

Tak określone mnożenie jest działaniem łącznym, przemiennym i rozdzielnym względem dodawania — zbiór liczb zespolonych z tymi działaniami, oznaczany symbolem $\mathbb{C}$, jest ciałem. Zerem w tym ciele jest liczba $(0, 0)$, a jedynką liczba $(1, 0)$. Ograniczając dodawanie i mnożenie do liczb, których części urojone są równe 0, otrzymamy wyniki, których części rzeczywiste są sumami albo iloczynami części rzeczywistych argumentów tych działań, a części urojone są zerem. W ten sposób ciało liczb rzeczywistych $\mathbb{R}$ jest „zanurzone" w ciele $\mathbb{C}$: możemy wykonywać dodawanie i mnożenie liczb zespolonych i rzeczywistych, doczepiając do tych ostatnich część urojoną 0.

[1] Płaszczyznę, której punkty traktujemy jak liczby zespolone, nazywamy **płaszczyzną Gaussa** lub **płaszczyzną zespoloną**.

Wartość bezwzględna liczby $z = (x, y)$ jest to liczba rzeczywista $|z| = \sqrt{x^2 + y^2}$. Dla dowolnych liczb z_1, z_2 jest $|z_1 z_2| = |z_1||z_2|$. **Liczba sprzężona** z z jest to liczba $\overline{z} = (x, -y)$. Iloczyn $z\overline{z}$ jest równy $(x^2 + y^2, 0)$, a zatem jego część rzeczywista jest kwadratem liczby $|z|$. **Odwrotność** liczby $z \neq (0, 0)$, czyli liczba z^{-1}, taka że $zz^{-1} = (1, 0)$, jest równa $\frac{1}{|z|^2}\overline{z}$.

Jedynka urojona jest to liczba $(0, 1)$, oznaczana najczęściej literą i. Jest $\mathrm{i}^2 = (-1, 0)$, a więc liczba i (a także $-\mathrm{i} = (0, -1)$) jest pierwiastkiem kwadratowym z -1.

Liczby zespolone można przedstawiać w **postaci trygonometrycznej**, tj. jako iloczyn liczby rzeczywistej $|z|$ i liczby zespolonej o wartości bezwzględnej 1:

$$z = |z|(\cos\varphi, \sin\varphi).$$

Jeśli liczba z nie jest zerem, to liczba rzeczywista $\varphi \in [-\pi, \pi)$, zwana **argumentem** liczby z, jest jednoznacznie określona. Liczby $|z|$, φ są współrzędnymi biegunowymi punktu na płaszczyźnie, którego współrzędnymi kartezjańskimi są części x, y liczby z.

Iloczyn dwóch liczb zespolonych zapisanych w postaci trygonometrycznej jest równy

$$z_1 z_2 = |z_1|(\cos\varphi_1, \sin\varphi_1)|z_2|(\cos\varphi_2, \sin\varphi_2) = |z_1||z_2|\big(\cos(\varphi_1 + \varphi_2), \sin(\varphi_1 + \varphi_2)\big).$$

Jego wartość bezwzględna jest iloczynem wartości bezwzględnych czynników, a argument jest sumą ich argumentów[2]. Liczby zespolone o wartości bezwzględnej 1 reprezentują obroty płaszczyzny; chcąc obrócić punkt (x, y) wokół początku układu współrzędnych o kąt φ, wystarczy pomnożyć go, jako liczbę zespoloną, przez liczbę $(\cos\varphi, \sin\varphi)$.

Całkowite potęgi liczb zespolonych opisuje **wzór de Moivre'a**:

$$z^k = |z|^k(\cos k\varphi, \sin k\varphi).$$

Ten sam wzór definiuje potęgowanie także wtedy, gdy wykładnik jest dowolną liczbą rzeczywistą[3]. Ostatni wzór, który tu podam, zamienia postać trygonometryczną liczby zespolonej na **postać wykładniczą**, zapisaną przy użyciu podstawy logarytmu naturalnego (liczby Eulera) $\mathrm{e} = 2.71828\ldots$; ma miejsce równość $|z|(\cos\varphi, \sin\varphi) = |z|\mathrm{e}^{\mathrm{i}\varphi}$. Wódz o tym wiedział.

F.1.2. Iterowanie wielomianu kwadratowego

Zbiór Mandelbrota składa się z tych liczb zespolonych c, dla których nieskończony ciąg liczb $z_0, z_1, z_2, \ldots$ określonych wzorami $z_0 = 0$ oraz $z_{k+1} = z_k^2 + c$ jest ograniczony. Wartości bezwzględne wszystkich takich liczb nie są większe niż 2, a więc cały zbiór Mandelbrota jest zawarty w kole o promieniu 2. Po utożsamieniu wybranego prostokąta w płaszczyźnie zespolonej z obszarem okna na ekranie można dla każdego piksela znaleźć odpowiadającą mu liczbę c, a następnie obliczać kolejne wyrazy ciągu, kończąc po osiągnięciu ustalonego limitu liczby wyrazów lub po wcześniejszym otrzymaniu takiej liczby z_k, że $|z_k| > R$, dla ustalonego

[2] Do tej sumy może być potrzebne dodanie lub odjęcie 2π, aby otrzymać argument z przedziału $[-\pi, \pi)$.

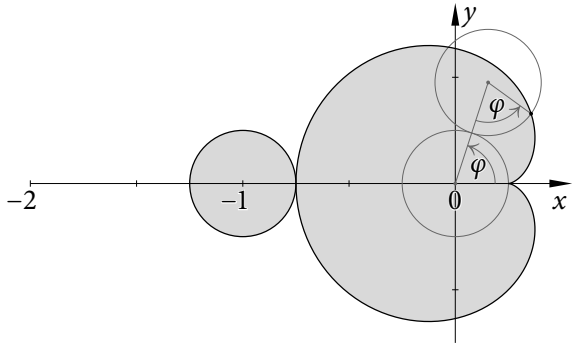
[3] **Uwaga:** Wprawdzie $\big(\cos(\varphi + 2m\pi), \sin(\varphi + 2m\pi)\big) = (\cos\varphi, \sin\varphi)$ dla każdej liczby całkowitej m, ale jeśli t nie jest liczbą całkowitą, a $m \neq 0$, to nie musi być $\big(\cos t(\varphi + 2m\pi), \sin t(\varphi + 2m\pi)\big) = (\cos t\varphi, \sin t\varphi)$.

$R \geqslant 2$. Kolor piksela można wybrać na podstawie liczby k, otrzymując pokolorowany obraz *otoczenia* zbioru Mandelbrota.

Funkcja, której wartością dla każdego punktu c płaszczyzny zespolonej jest najmniejsza liczba całkowita k, taka że $|z_k| > R$, lub ∞, jeśli taka liczba nie istnieje, jest nieciągła. Na potrzeby grafiki warto ją „uciąglić", czyli tak zmodyfikować, aby otrzymać funkcję ciągłą poza zbiorem Mandelbrota[4]. Najprostsza taka funkcja jest określona wzorem

$$f(c) = k + \frac{R - |z_{k-1}|}{|z_k| - |z_{k-1}|},$$

w którym $|z_{k-1}| \leqslant R < |z_k|$. Jej ciągłość wynika stąd, że dla każdego $k > 0$ funkcja $w_k(c) = z_k$ jest wielomianem (stopnia 2^{k-1}), a więc jest funkcją ciągłą.



Rysunek F.1. Obszary wewnątrz zbioru Mandelbrota

Duża część zbioru Mandelbrota składa się z **koła** o środku $(-1, 0)$ i promieniu $\frac{1}{4}$ oraz obszaru, którego brzegiem jest krzywa zwana **kardioidą** opisana przez punkt okręgu o promieniu $\frac{1}{4}$ toczącego się po okręgu o środku $(0, 0)$ i tym samym promieniu (rys. F.1). Warto sprawdzić, czy odpowiadająca pikselowi liczba $c = (x, y)$ leży w tym kole lub w tym obszarze, bo jeśli tak, to ciąg jest ograniczony i można od razu zaniechać obliczania jego wyrazów.

Punkt (x, y) leży we wspomnianym kole, gdy $(x+1)^2 + y^2 \leqslant \frac{1}{16}$. Nieco bardziej skomplikowane jest badanie, czy punkt leży w obszarze ograniczonym przez kardioidę. Jeśli okrąg, po którym toczy się drugi okrąg, przesuniemy tak, aby jego środek był w punkcie $(-\frac{1}{4}, 0)$, to opis kardioidy we współrzędnych biegunowych ma postać

$$r(\varphi) = \frac{1}{2}(1 - \cos\varphi), \quad \varphi \in [0, 2\pi).$$

Na tej podstawie można (najpierw spróbuj samodzielnie, a potem zajrzyj na stronę [60]) znaleźć funkcję opisaną wzorem

$$w(x, y) = (\hat{x}^2 + y^2)^2 + \hat{x}(\hat{x}^2 + y^2) - y^2/4,$$

[4]Dokładniej, ciągłą w każdym kole, którego brzeg nie przecina zbioru Mandelbrota; funkcja pozostanie w jego otoczeniu nieograniczona, a zatem nieciągła.

w którym $\hat{x} = x - \frac{1}{4}$. Funkcja ta ma wartość 0 we wszystkich punktach kardioidy, ujemną w ograniczonym przez nią obszarze i dodatnią poza tym obszarem.

Dwie rzeczy w aplikacji wyświetlającej obrazy zbioru Mandelbrota mogą być zmieniane w czasie rzeczywistym: prostokąt, który można przesuwać i zmniejszać lub powiększać, oraz **paleta**, czyli odwzorowanie wartości funkcji $f(c)$ na kolor. Zazwyczaj limit N liczby iteracji, po osiągnięciu którego obliczenia są przerywane, jest rzędu kilkuset do kilku tysięcy, wskutek czego wykonanie obrazu trwa zbyt długo, aby można było osiągnąć płynną animację wybierania prostokąta. Ale przekształcanie liczby na kolor zabiera bardzo mało czasu, a po zmianie palety nie trzeba na nowo liczyć iteracji dla każdego piksela. Z tego powodu sensowne wydaje się wykonywanie obrazów w dwóch etapach: w pierwszym dla każdego piksela trzeba znaleźć odpowiednią liczbę c, a potem obliczyć i zapamiętać wartość funkcji $f(c)$, albo liczbę przyjętą do zakodowania nieskończoności, jeśli $|z_N| \leqslant R$. W drugim etapie trzeba narysować prostokąt w oknie, kolorując go na podstawie zapamiętanych liczb i bieżącej palety.

Listing F.1. Szader wierzchołków

GLSL

```
#version 450 core

void main ( void )
{
  const vec4 p[4] = { vec4(-1,-1,0,1), vec4(1,-1,0,1),
                      vec4(1,1,0,1), vec4(-1,1,0,1) };

  gl_Position = p[gl_VertexID];
} /*main*/
```

Listing F.1 przedstawia szader, którego zadaniem jest wyprowadzenie jednego wierzchołka kwadratu w układzie kostki standardowej; obraz kwadratu o boku 2, położonego w płaszczyźnie xy i o środku w początku układu wypełni klatkę (zajmującą całe okno aplikacji). Szader ten wchodzi w skład programów używanych w obu etapach wykonywania obrazu. Odwzorowanie współrzędnych fragmentu (piksela) w oknie w płaszczyznę zespoloną wykona szader fragmentów pierwszego z tych programów.

Blok zmiennych jednolitych zawierający dane potrzebne do tego odwzorowania jest przedstawiony na listingu F.2. Wartości pól zmiennej `rxy` opisują wybrany prostokąt; `x` i `y` przechowują współrzędne x lewego i prawego boku, a `z` i `w` współrzędne y dolnego i górnego boku tego prostokąta. W zmiennej `wh` są przechowywane wymiary (szerokość i wysokość) klatki w pikselach. Pozostałe pola są opisane dalej.

Listing F.2. Blok `TransBlock` szaderów fragmentów

GLSL

```
uniform TransBlock {
    dvec4 rxy;
    dvec2 wh;
    int   width, mag, maxit, currit, diter;
  } tr;
```

Najefektowniejsze obrazy zbioru Mandelbrota przedstawiają fragmenty jego otoczenia w *bardzo dużym* powiększeniu. Użytkownik aplikacji może odwzorować w okno tak mały fragment płaszczyzny zespolonej, że współrzędne punktów odpowiadających sąsiednim pikselom (czyli części rzeczywiste i urojone liczb zespolonych będących tymi punktami) mogą być reprezentowane przez te same liczby zmiennopozycyjne, przez co jakość obrazów spada. Użyjemy podwójnej precyzji: zmienne `rxy` i `wh` są typu `dvec4` i `dvec2`, ale i tak konieczne są ograniczenia. Jeśli obliczenia elementów ciągu są prowadzone w podwójnej precyzji, to można dopuścić minimalną średnicę prostokąta rzędu 10^{-13}.

W polu `maxit` jest podany limit liczby iteracji. Wartość m pola `mag` (1 lub 3) określa powiększenie obrazu wykonywanego w pierwszym etapie; obrazy fraktali *powinny* być antyaliasowane i dlatego obraz ten (czyli tablica, w której będą zapamiętywane liczby wykonanych iteracji) ma rozdzielczość m razy większą niż obraz końcowy.

Jeśli limit N liczby iteracji jest rzędu kilku lub kilkunastu tysięcy, to nawet największa moc obliczeniowa GPU[5] nie wystarczy do wykonywania obliczeń dostatecznie szybko, aby można było, zmieniając prostokąt, otrzymać płynną animację obrazów[6]; opóźnienia mogą być nawet rzędu sekundy. Można się z tym pogodzić lub opracować pewien kompromis: iteracje wykonywać podetapami i wyświetlać obrazy tego, co GPU zdążyła policzyć w ustalonym limicie czasu, dostatecznie krótkim, aby aplikacja płynnie reagowała na działania użytkownika. Jeśli podczas trwania podetapu prostokąt został zmieniony, to dla nowego prostokąta trzeba obliczenia zacząć od początku, a jeśli nie, to można wykonać kolejny podetap, w którym liczba iteracji będzie dostosowana do mocy obliczeniowej GPU. To rozwiązanie ma swoją cenę: jest nią duże zapotrzebowanie na pamięć GPU. Na przykład obraz o rozdzielczości 4K ma wymiary 3480×2160 pikseli. Z włączonym antyaliasingiem dla każdego piksela będziemy iterować wielomiany odpowiadające 9 punktom płaszczyzny zespolonej. Dla każdego takiego punktu c będzie trzeba pamiętać wartość funkcji $f(c)$ w zmiennej typu `float`, oraz liczbę zespoloną z_{k-1} w zmiennej typu `dvec2`. Zajmie to w sumie 1492992000 bajtów, czyli prawie półtora gigabajta.

Listing F.3 przedstawia szader fragmentów pierwszego etapu rysowania. Wyniki jego obliczeń, czyli wartości funkcji f, trafiają do obrazu `img` przy użyciu procedury `imageStore`, dlatego ten szader zawsze kończy działanie instrukcją `discard`. Kwalifikator `r32f` w deklaracji zmiennej `img` oznacza, że każdy piksel w tym obrazie ma tylko jedną składową, reprezentowaną jako 32-bitowa liczba zmiennopozycyjna. Niestety, dostępne w OpenGL-u formaty obrazów nie dają możliwości pamiętania pikseli o składowych podwójnej precyzji[7], dlatego liczby z_k, które trzeba pamiętać między podetapami pierwszego etapu rysowania, są przechowywane w bloku magazynowym `Cmap`, zawierającego odpowiednio długą tablicę.

Symbole `INF0`, `INF1` i `INF2` reprezentują trzy liczby większe niż największy przewidywany limit liczby iteracji; pierwsza z nich ma sygnalizować odkrycie, że punkt c leży w kole, druga, że w obszarze ograniczonym kardioidą, a trzecia oznacza, że wykonane iteracje nie wystarczyły do stwierdzenia rozbieżności ciągu. Umożliwia to późniejsze nadanie odpowiednim pikselom innych kolorów.

[5] istniejących w chwili pisania tego tekstu

[6] Istotna jest też wielkość obrazu i to, czy jest włączony antyaliasing.

[7] a szkoda

Listing F.3. Szader fragmentów pierwszego etapu rysowania zbioru Mandelbrota

GLSL

```
#version 440 core

#define INF0 65533.0
#define INF1 65534.0
#define INF2 65535.0
#define R        5.0
#define RR      25.0

uniform TransBlock { .... } tr;  /* listing F.2 */

layout(r32f,binding=1) uniform image2D img;
layout(std430,binding=0) buffer Cmap { dvec2 cmap[]; };

void main ( void )
{
  dvec2  c, z;
  double xx, x2, y2, r2, rk2, fr;
  int    i;
  ivec2  xy;

  xy = ivec2 ( int(gl_FragCoord.x), int(gl_FragCoord.y) );
  c = dvec2 ( tr.rxy.x + double(gl_FragCoord.x)/tr.wh.x*(tr.rxy.y-tr.rxy.x),
              tr.rxy.z + double(gl_FragCoord.y)/tr.wh.y*(tr.rxy.w-tr.rxy.z) );
  if ( tr.currit == 0 ) {
    xx = c.x+1.0;  y2 = c.y*c.y;
    if ( xx*xx+y2 <= 0.0625 ) {
      imageStore ( img, xy, vec4 ( INF0, 0.0, 0.0, 0.0 ) );
      cmap[xy.y*tr.width + xy.x] = c;
      discard;
    }
    else {
      xx = c.x - 0.25;  x2 = xx*xx;  r2 = x2+y2;
      if ( (r2 + xx)*r2 <= 0.25 * y2 ) {
        imageStore ( img, xy, vec4 ( INF1, 0.0, 0.0, 0.0 ) );
        cmap[xy.y*tr.width + xy.x] = c;
        discard;
      }
    }
    imageStore ( img, xy, vec4 ( INF2, 0.0, 0.0, 0.0 ) );
    z = dvec2 ( 0.0, 0.0 );
    rk2 = 0.0;
  }
  else {
    if ( imageLoad ( img, xy ).r < INF0 )
      discard;
```

```
46:     z = cmap[xy.y*tr.width + xy.x];
47:     rk2 = z.x*z.x + z.y*z.y;
48:   }
49:   for ( i = tr.currit;  i < tr.currit+tr.diter;  i++ ) {
50:     z = dvec2 ( (z.x + z.y)*(z.x - z.y) + c.x, (z.x + z.x)*z.y + c.y );
51:     if ( (r2 = z.x*z.x + z.y*z.y) > RR ) {
52:       fr = (R-sqrt ( rk2 )) / (sqrt ( r2 )-sqrt ( rk2 ));
53:       imageStore ( img, xy, vec4 ( float(i) + fr, 0.0, 0.0, 0.0 ) );
54:       cmap[xy.y*tr.width + xy.x] = z;
55:       discard;
56:     }
57:     rk2 = r2;
58:   }
59:   cmap[xy.y*tr.width + xy.x] = z;
60:   discard;
61: } /*main*/
```

W linii 21 współrzędne fragmentu są zaokrąglane i składane w wektor xy potrzebny do zapisania wyniku obliczeń w obrazie img. Części rzeczywista i urojona liczby c odpowiadającej fragmentowi są obliczane w liniach 22–23.

Wartość zmiennej jednolitej currit w bloku TransBlock jest liczbą iteracji wykonanych od początku pierwszego etapu rysowania (w poprzednich podetapach). Jeśli jest równa 0, to szader sprawdza, czy liczba c leży w kole lub w obszarze, którego brzegiem jest kardioida. Zmienne xx i y2 otrzymują wartości $x+1$ i y^2, po czym w linii 26 następuje sprawdzenie, czy punkt c leży w kole. Jeśli nie, to w linii 32 zmiennym xx, x2 i r2 kolejno przypisywane są wartości $\hat{x}$, $\hat{x}^2$ i $\hat{x}^2+y^2$ i następuje badanie, czy jest spełniona nierówność $(\hat{x}^2+y^2)^2+\hat{x}(\hat{x}^2+y^2) \leqslant y^2/4$, czyli czy $w(x,y) \leqslant 0$. Zależnie od wyników testów, do obrazu img trafia jedna z liczb reprezentujących nieskończoność, a w tablicy cmap zostaje zapamiętana liczba c.

Jeśli zmienna currit ma wartość niezerową, to szader został wywołany w celu kontynuowania iteracji. Jeśli wcześniej w obrazie img została zapisana liczba mniejsza niż INF0, to obliczenia dla tego punktu są już zakończone i szader natychmiast kończy działanie instrukcją w linii 45. W przeciwnym razie zmiennej z jest przypisywana wartość z_k odczytana z tablicy cmap, a w linii 47 jest obliczana liczba $|z_k|^2$.

Iteracje wielomianu kwadratowego są wykonywane w liniach 49–58; zmienna jednolita diter przechowuje bieżący limit liczby iteracji, określony przez aplikację[8]. Warunkiem przerwania pętli jest wykonanie danej liczby iteracji w bieżącym podetapie lub wykonanie w sumie N iteracji we wszystkich podetapach pierwszego etapu rysowania.

Obliczenie kolejnej liczby z_k następuje w linii 50, a zaraz potem szader sprawdza, czy należy przerwać iteracje, bo $|z_k| > R$; wartość zmiennej r2, czyli $|z_k|^2$, jest porównywana z R^2. W linii 52 jest obliczana część ułamkowa funkcji $f(c)$, która jest dodawana do części całkowitej (liczby wykonanych iteracji) w linii następnej. Wartość zmiennej rk2 jest równa

[8] Nie ma możliwości przerwania rozpoczętych obliczeń na GPU, a zatem aplikacja nie może spowodować ich przerwania po ustalonym czasie. Zamiast tego aplikacja dobiera bieżący limit liczby iteracji na podstawie szybkości obliczeń zmierzonej w poprzednim podetapie.

$|z_{k-1}|^2$. W linii 59 ostatnia otrzymana liczba z_k, która leży w kole o promieniu R, zostaje zapamiętana, aby można było wznowić iteracje w kolejnym podetapie.

Listing F.4 przedstawia procedurę, która wywołuje program zbudowany z szaderów opisanych wyżej, a potem program drugiego etapu rysowania, wykonujący obraz bieżącego stanu obliczeń. Działanie procedury zależy od wartości zmiennych globalnych, których deklaracje są pokazane na listingu. Zmienna `mappingchanged` otrzymuje wartość `true` po każdej zmianie prostokąta odwzorowanego w okno, po zmianie wymiarów okna i po zmianie limitu N liczby iteracji, pamiętanego w zmiennej `maxiter`. Jeśli zmienna `mappingchanged` ma wartość niezerową, to pierwszy etap rysowania trzeba zacząć od początku; zmienna `curriter` otrzymuje wartość 0. Zmienna `finished` nie otrzyma wartości `true`, dopóki suma liczb iteracji wykonanych w poszczególnych podetapach będzie mniejsza niż N.

W linii 24 są obliczane wymiary obrazu w pikselach — takie jak okno lub 3 razy większe, jeśli jest włączony antyaliasing. W liniach 26–32 zmiennym jednolitym w bloku `TransBlock` są nadawane wartości; w szczególności zmiennej `curriter` zostaje przypisana liczba iteracji wykonanych we wcześniejszych podetapach, a zmienna `diter` otrzymuje wartość początkową podaną w deklaracji w linii 12 albo wartość obliczoną w liniach 42–43 w poprzednim wywołaniu procedury rysującej.

Podetapy pierwszego etapu wykonują obraz w pozaekranowym buforze ramki, którego sposób tworzenia jest opisany dalej. Instrukcje w liniach 39 i 41 mierzą czas trwania podetapu, w którym wykonane zostało `diter` iteracji, po czym (na potrzeby kolejnego podetapu) obliczana jest liczba iteracji, dla której spodziewany czas obliczeń (w sekundach) jest podany w makrodefinicji `FTIME` — celem jest wyświetlanie 30 klatek na sekundę, co zapewnia wystarczającą płynność animacji. Aby pomiary czasu obliczeń na GPU miały sens, wywołania procedury `TimerToc` są poprzedzone wywołaniami procedury `glFinish`, która czeka na dokończenie wszelkich obliczeń przez GPU. Bez tego byłby mierzony tylko czas „wprawiania obliczeń w ruch".

Listing F.4. Procedura rysowania zbioru Mandelbrota

C

```
1: #define MINMAXIT   100
2: #define MAXMAXIT 16000
3: #define FTIME       0.032
4: #define UNB GL_UNIFORM_BUFFER
5:
6: int            wdt, hgh, magaa;
7: static double  width, height;
8: static OffsFBO *fbo;
9: static char   mappingchanged = true, finished = false, antialias1 = true;
10: static GLuint program_id[2], trbuf, trbpoint;
11: static GLint  trofs[7];
12: static GLint  maxiter = MINMAXIT, curriter = 0, diter = 100;
13:
14: char RedrawMyObject1 ( void )
15: {
```

```
16:   GLdouble xyc[2];
17:   GLint    dit;
18:   double   t0, t1, di;
19:
20:   if ( mappingchanged ) { finished = false;  curriter = 0; }
21:   if ( !finished ) {
22:     mappingchanged = false;
23:     magaa = antialias1 ? MAGAA : 1;
24:     xyc[0] = magaa*width;  xyc[1] = magaa*height;
25:     glBindBufferBase ( UNB, trbpoint, trbuf );
26:     glBufferSubData ( UNB, trofs[1], 2*sizeof(GLdouble), xyc );
27:     glBufferSubData ( UNB, trofs[2], sizeof(GLint), &maxiter );
28:     glBufferSubData ( UNB, trofs[3], sizeof(GLint), &magaa );
29:     glBufferSubData ( UNB, trofs[4], sizeof(GLint), &curriter );
30:     glBufferSubData ( UNB, trofs[5], sizeof(GLint), &fbo->width );
31:     dit = curriter+diter <= maxiter ? diter : maxiter-curriter;
32:     glBufferSubData ( UNB, trofs[6], sizeof(GLint), &dit );
33:     glBindFramebuffer ( GL_FRAMEBUFFER, fbo->fbo );
34:     glViewport ( 0, 0, magaa*wdt, magaa*hgh );
35:     glBindImageTexture ( 1, fbo->txt, 0, GL_FALSE, 0, GL_WRITE_ONLY,
36:                          GL_R32F );
37:     glUseProgram ( program_id[0] );
38:     glBindVertexArray ( empty_vao );
39:     glFinish ();  t0 = TimerToc ();
40:     glDrawArrays ( GL_TRIANGLE_FAN, 0, 4 );
41:     glFinish ();  t1 = TimerToc ();
42:     di = (double)dit*FTIME/(t1-t0);
43:     diter = di < MINMAXIT ? MINMAXIT : (di > MAXMAXIT ? MAXMAXIT : (int)di);
44:     glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
45:     finished = (curriter += dit) >= maxiter;
46:   }
47:   glViewport ( 0, 0, wdt, hgh );
48:   glBindImageTexture ( 1, fbo->txt, 0, GL_FALSE, 0, GL_READ_ONLY, GL_R32F );
49:   glUseProgram ( program_id[1] );
50:   glDrawArrays ( GL_TRIANGLE_FAN, 0, 4 );
51:   glBindVertexArray ( 0 );
52:   if ( showtext1 )
53:     DisplayTextObject ( mytext );
54:   glUseProgram ( 0 );
55:   glFlush ();
56:   ExitIfGLError ( "RedrawMyObject1" );
57:   return finished;
58: } /*RedrawMyObject1*/
```

Po wykonaniu podetapu lub po jego pominięciu, jeśli zmienna `finished` miała wartość `false`, drugi program szaderów, opisany dalej, wykonuje obraz w oknie. Wartość `true` przekazywana przez procedurę `RedrawMyObject1` sygnalizuje, że pierwszy etap rysowania jest zakończony. Listing F.5 przedstawia przykład użycia tej procedury w aplikacji biblioteki

GLFW; zmienna `redraw1`, której wartość `true` powoduje wywołanie procedury `Redraw1` w głównej pętli komunikatów aplikacji, otrzymuje wartość `false` dopiero po wykonaniu wszystkich podetapów pierwszego etapu rysowania. Pomiędzy wywołaniami procedury rysowania aplikacja obsługuje komunikaty wejściowe, powodowane działaniami użytkownika.

Listing F.5. Procedura wykonywania obrazu w części okienkowej

C

```
1: void Redraw1 ( GLFWwindow *win )
2: {
3:   glfwMakeContextCurrent ( win );
4:   if ( opti1 == 0 ) {
5:     if ( (redraw1 = !RedrawMyObject1 ()) )
6:       glfwPostEmptyEvent ();
7:   }
8:   else {
9:     opti1 --;
10:    glfwPostEmptyEvent ();
11:  }
12:  glfwSwapBuffers ( win );
13:  ExitIfGLError ( "Redraw" );
14: } /*Redraw1*/
```

F.1.3. Obliczanie koloru piksela

Zadaniem szadera, którego procedura `main` widnieje na listingu F.6, jest zamiana liczb $f(c)$, wpisanych w pierwszym etapie do obrazu `img`, na kolor piksela. Wymiary (szerokość i wysokość w pikselach) tego obrazu są takie same jak wymiary obrazu końcowego (czyli klatki zajmującej całe okno aplikacji) lub $m = 3$ razy większe. W pierwszym przypadku (w linii 31) kolor piksela na końcowym obrazie jest obliczany przy użyciu wybranej palety na podstawie jednego piksela obrazu wejściowego.

W drugim przypadku (w liniach 33–48) jest wykonywany antyaliasing: kolor każdego piksela jest średnią ważoną kolorów wielu pikseli obrazu wejściowego. Kolor piksela (ξ, η) obrazu wynikowego q jest obliczany na podstawie pikseli obrazu wejściowego p według wzoru

$$q(\xi,\eta) = \sum_{i=-3}^{3} N_i \sum_{j=-3}^{3} N_j p(3\xi + i, 3\eta + j), \tag{F.2}$$

w którym liczby N_i są wartościami średnimi funkcji rozkładu normalnego Gaussa $\mathcal{N}_1$ w przedziałach $[i - 1/2, i + 1/2]$ (zobacz podrozdz. 27.2); przyjęte odchylenie standardowe $\sigma = 1$ odpowiada tu szerokości lub wysokości jednego piksela obrazu wejściowego.

Pomocnicza funkcja `itx` odczytuje liczbę $f(c)$ zapamiętaną w pikselu obrazu wejściowego, przy czym jeśli podane współrzędne określają piksel poza obrazem, to wartością funkcji jest liczba `INF0`, którą opisane dalej palety zamienią na kolor czarny. Tablica `mgf` zawiera

Listing F.6. Procedura `main` szadera fragmentów drugiego etapu rysowania

GLSL

```
#version 450 core

uniform TransBlock { .... } tr;  /* listing F.2 */

layout(location=0) out vec4 out_Colour;

layout(r32f,binding=1) uniform image2D img;

ivec2 wh;

float itx ( ivec2 xy )
{
  if ( xy.x < 0 || xy.y < 0 || xy.x >= wh.x || xy.y >= wh.y )
    return float(INF0);
  else
    return imageLoad ( img, xy ).r;
} /*itx*/

void main ( void )
{
  const float mgf[4] =
               {0.3831031644, 0.2418428568, 0.0606257426, 0.0059798184};
  int   i, j;
  ivec2 xy;
  float tx;
  vec3  Colour, col;

  wh = imageSize ( img );
  xy = ivec2 ( int(gl_FragCoord.x), int(gl_FragCoord.y) );
  if ( tr.mag == 1 )
    Colour = Palette ( uint(itx ( xy )) );
  else {  /* mag == 3 */
    xy = tr.mag * xy - ivec2 (-1,-1);
    col = mgf[0] * Palette ( itx ( xy ) );
    for ( j = 1; j < 4; j++ )
      col += mgf[j] * (Palette ( itx ( xy+ivec2(0, j) ) ) +
                       Palette ( itx ( xy+ivec2(0,-j) ) ));
    Colour = mgf[0] * col;
    for ( i = 1; i < 4;  i++ ) {
      col = mgf[0] * (Palette ( itx ( xy+ivec2( i,0) ) ) +
                      Palette ( itx ( xy+ivec2(-i,0) ) ));
      for ( j = 1; j < 4; j++ )
        col += mgf[j] * (Palette ( itx ( xy+ivec2( i, j) ) ) +
                         Palette ( itx ( xy+ivec2( i,-j) ) ) +
                         Palette ( itx ( xy+ivec2(-i, j) ) ) +
```

```
46:                          Palette ( itx ( xy+ivec2(-i,-j) ) ));
47:       Colour += mgf[i] * col;
48:     }
49:   }
50:   out_Colour = vec4 ( AGamma ( Colour ), 1.0 );
51: } /*main*/
```

liczby N_i, te same co na listingu 27.7. Instrukcje w liniach 34–48 realizują obliczanie koloru piksela według wzoru (F.2), po czym jest on poddawany korekcji gamma i wyprowadzany.

Na listingu F.7 są przedstawione procedury realizujące dwie palety i wywołująca je procedura pomocnicza `Palette`; pierwsza paleta (linie 1–10) jest bardzo prosta: punktom w kole nadaje kolor niebieski, punktom w obszarze, którego brzegiem jest kardioida kolor zielony, punktom, dla których ciąg liczb z_k po N iteracjach nie opuścił koła o promieniu R kolor czerwony, a pozostałym punktom kolor czarny albo biały, zależnie od parzystości liczby wykonanych iteracji (czyli podanej jako parametr części całkowitej wartości funkcji $f(c)$).

Procedura w liniach 32–47 realizuje bardziej wyrafinowaną paletę, określoną przy użyciu wymiernej krzywej Béziera $\boldsymbol{p}$ (zobacz podrozdz. 15.1 i 15.2), którą użytkownik aplikacji może zmieniać. Jeśli w pierwszym etapie zostało wykonane N iteracji lub test spowodował ich zaniechanie, to punkt jest czarny. Jeśli obliczenia zostały przerwane po mniej niż N iteracjach, to w linii 43 jest obliczany parametr $t = f(c)/N$ krzywej, będący liczbą z przedziału $(0,1)$. Krzywa jest położona w trójwymiarowej przestrzeni kolorów; współrzędne xy punktu $\boldsymbol{p}(t)$ określają odcień, a współrzędna z odpowiada za jasność koloru.

Blok zmiennych jednolitych `PaletteBlock` zawiera reprezentację krzywej; wartość pola `deg` to stopień krzywej, tablica `cp` zawiera wektory współrzędnych jednorodnych punktów kontrolnych, a wektory w tablicy `bpcp` reprezentują te same punkty, przy czym współrzędne wagowe są równe 1. Ta ostatnia reprezentacja jest potrzebna do wykonania obrazu krzywej, aby użytkownik mógł ją wygodnie zmieniać; rozwinięcie tego tematu jest dalej.

Po obliczeniu (przez procedurę wywołaną w linii 41) punktu $\boldsymbol{p}(t)$ następuje obliczenie koloru. Do współrzędnych kartezjańskich x, y jest dołączana liczba $1 - x - y$ i w ten sposób powstają współrzędne barycentryczne opisujące składowe koloru. W linii 43 składowe te są obcinane do przedziału $[0,1]$; w tym momencie zostaje ustalony odcień, zależny od proporcji

Listing F.7. Podprogramy realizujące palety

GLSL

```
1: vec3 Palette0 ( uint itr )
2: {
3:   switch ( itr ) {
4: case INF0: return vec3 ( 0.0, 0.0, 1.0 );
5: case INF1: return vec3 ( 0.0, 1.0, 0.0 );
6: case INF2: return vec3 ( 1.0, 0.0, 0.0 );
7:   default: return (itr & 0x01) != 0 ? vec3 ( 1.0, 1.0, 1.0 ) :
8:                   vec3 ( 0.0, 0.0, 0.0 );
9:   }
10: } /*Palette0*/
11:
```

```
12: uniform PaletteBlock {
13:     int  deg;
14:     vec4 cp[MAXDEG+1], bpcp[MAXDEG+1];
15:   } pal;
16:
17: vec3 BCHorner3Rf ( int n, vec4 bcp[MAXDEG+1], float t )
18: {
19:   int   i, b;
20:   float s, d;
21:   vec4  p;
22:
23:   s = 1.0-t;  d = t;  b = n;
24:   p = bcp[0];
25:   for ( i = 1; i <= n; i++ ) {
26:     p = s*p + (b*d)*bcp[i];
27:     d *= t;  b = (b*(n-i))/(i+1);
28:   }
29:   return p.xyz/p.w;
30: } /*BCHorner3Rf*/
31:
32: vec3 Palette1 ( float itr )
33: {
34:   float t;
35:   vec3  p, c;
36:
37:   if ( itr >= INF0 )
38:     return vec3 ( 0.0, 0.0, 0.0 );
39:   else {
40:     t = itr / float(trb.maxit);
41:     p = BCHorner3Rf ( pal.deg, pal.cp, t );
42:     c.rg = p.xy;  c.b = 1.0 - p.x - p.y;
43:     c = clamp ( c, vec3(0.0), vec3(1.0) );
44:     t = max ( c.r, c.g );  t = max ( t, c.b );
45:     return p.z*c/t;
46:   }
47: } /*Palette1*/
48:
49: uniform int pn = 1;
50:
51: vec3 Palette ( float tx )
52: {
53:   switch ( pn ) {
54:  default: return Palette0 ( uint(tx) );
55:   case 1: return Palette1 ( tx );
56:   case 2: return Palette1 ( trunc(tx) );
57:   }
58: } /*Palette*/
```

składowych, czyli od kierunku wektora (r, g, b) zapamiętanego w zmiennej `c`. Pozostaje ustalenie jasności, czyli długości wektora współrzędnych koloru. W linii 44 największa składowa jest przypisywana zmiennej `t`. Obliczony w linii 45 wektor ma największą współrzędną równą współrzędnej z punktu $\boldsymbol{p}(t)$ (ma ona wartość z przedziału $[0,1]$).

Wywołując `Palette1`, procedura `Palette` podaje jako parametr liczbę $f(c)$ albo część całkowitą tej liczby — do wyboru przez użytkownika aplikacji.

F.1.4. Pozaekranowy bufor ramki

W pierwszym etapie rysowanie odbywa się poza ekranem. Obraz, w którym mają być zapamiętane wartości funkcji $f(c)$ dla każdego piksela, jest załącznikiem koloru używanego wtedy pozaekranowego bufora ramki[9]. Oprócz tego obrazu potrzebna jest tablica, w której będą pamiętane liczby z_k.

Listing F.8 przedstawia strukturę `OffsFBO` będącą opakowaniem pozaekranowego bufora ramki, oraz procedury obsługi tego bufora. Procedura `NewOffsFBO` tworzy bufor, którego załącznikiem jest tekstura o podanych wymiarach (tekstura ta zawiera obraz, w którym są zapisywane wartości funkcji $f(c)$), oraz bufor zawierający blok magazynowy `Cmap`; bufor ten zostaje przywiązany do punktu dowiązania `0` w celu `GL_SHADER_STORAGE_BUFFER` (zobacz listing F.3, linia 12). Procedura `SetOffsFBOSize` służy do zmiany wymiarów tekstury i bufora z blokiem `Cmap`, które trzeba dostosowywać do wymiarów okna aplikacji. Wartość `GL_R32F` trzeciego parametru procedury `glTexStorage2D` wywołanej w linii 10 oznacza, że każdy element tekstury ma mieć tylko jedną składową reprezentowaną jako 32-bitowa liczba zmiennopozycyjna. Procedura `DeleteOffsFBO` likwiduje bufor ramki z załącznikiem i bufor z blokiem magazynowym `Cmap`.

Listing F.8. Procedury obsługi pozaekranowego bufora ramki

C

```
1:  typedef struct {
2:      GLuint fbo, txt, cmap;
3:      int    width, height;
4:    } OffsFBO;
5:
6:  static char AllocOffsTexture ( OffsFBO *fbo, int w, int h )
7:  {
8:    glGenTextures ( 1, &fbo->txt );
9:    glBindTexture ( GL_TEXTURE_2D, fbo->txt );
10:   glTexStorage2D ( GL_TEXTURE_2D, 1, GL_R32F, w, h );
11:   glBindFramebuffer ( GL_FRAMEBUFFER, fbo->fbo );
12:   glFramebufferTexture ( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, fbo->txt,
```

[9] Bufor ramki jest gotowy do pracy, gdy ma określoną szerokość i wysokość w pikselach; wystarczy podanie załącznika, tj. tekstury o określonych wymiarach. Można też podać wymiary za pomocą procedury `glFramebufferParameteri` i w ten sposób przygotować do pracy bufor ramki bez załączników (przykład jest pokazany w p. 29.2.5); zwróćmy uwagę, że szader fragmentów z listingu F.3 zawsze wykonuje instrukcję `discard`; wynik jego obliczeń jest zapisywany z pominięciem ostatniego etapu potoku przetwarzania grafiki.

```
13:                           0 );
14:   if ( glCheckFramebufferStatus ( GL_FRAMEBUFFER ) !=
15:        GL_FRAMEBUFFER_COMPLETE )
16:     ExitOnError ( "AllocOffsTexture" );
17:   glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
18:   fbo->cmap = NewStorageBuffer ( w*h*2*sizeof(GLdouble), 0 );
19:   fbo->width = w;  fbo->height = h;
20:   ExitIfGLError ( "AllocOffsTexture" );
21:   return true;
22: } /*AllocOffsTexture*/
23:
24: OffsFBO *NewOffsFBO ( int w, int h )
25: {
26:   OffsFBO *fbo;
27:
28:   if ( (fbo = malloc (sizeof(OffsFBO))) ) {
29:     glGenFramebuffers ( 1, &fbo->fbo );
30:     if ( AllocOffsTexture ( fbo, w, h ) )
31:       return fbo;
32:     else {
33:       glDeleteFramebuffers ( 1, &fbo->fbo );
34:       free ( fbo );
35:     }
36:   }
37:   return NULL;
38: } /*NewOffsFBO*/
39:
40: char SetOffsFBOSize ( OffsFBO *fbo, int w, int h )
41: {
42:   if ( w != fbo->width || h != fbo->height ) {
43:     glDeleteTextures ( 1, &fbo->txt );
44:     glDeleteBuffers ( 1, &fbo->cmap );
45:     if ( !AllocOffsTexture ( fbo, w, h ) ) {
46:       glDeleteFramebuffers ( 1, &fbo->fbo );
47:       free ( fbo );
48:       return false;
49:     }
50:   }
51:   return true;
52: } /*SetOffsFBOSize*/
53:
54: void DeleteOffsFBO ( OffsFBO *fbo )
55: {
56:   glDeleteFramebuffers ( 1, &fbo->fbo );
57:   glDeleteTextures ( 1, &fbo->txt );
58:   glDeleteBuffers ( 1, &fbo->cmap );
59:   ExitIfGLError ( "DeleteOffsFBO" );
```

```
60:   free ( fbo );
61: } /*DeleteOffsFBO*/
```

F.1.5. Odwzorowanie prostokąta w okno

Prostokąt w płaszczyźnie zespolonej, którego obraz wypełnia okno, jest określony za pomocą środka i promienia okręgu opisanego na tym prostokącie; długości jego boków można obliczyć, znając ten promień i wymiary okna w pikselach. Współrzędne środka i promień są przechowywane w zmiennych `xc`, `yc` i `rc`; deklaracje pozostałych zmiennych globalnych, których wartości zależą od prostokąta i od wymiarów okna, są pokazane na listingu F.4.

Obliczenie współrzędnych wierzchołków prostokąta wykonuje procedura pokazana na listingu F.9. Instrukcja w linii 11 oblicza długość przekątnej okna w pikselach (przy założeniu, że współczynnik aspekt jest równy 1). W linii 12 są obliczane sinus i kosinus kąta między przekątną a poziomym bokiem prostokąta, co umożliwia znalezienie w linii 13 połówek szerokości i wysokości prostokąta. W liniach 14 i 15 liczby te są odejmowane od i dodawane do współrzędnych środka prostokąta, po czym następuje przesłanie otrzymanej czwórki liczb do zmiennej `rxy` w bloku `TransBlock`.

Wywołana w linii 18 procedura `NotifyRectangle` przygotowuje tekstowy opis prostokąta, który może być dodatkowo wyświetlony w oknie.

Listing F.9. Procedura znajdująca odwzorowanie płaszczyzny zespolonej w okno

C

```
1: static double xc = -0.5, yc = 0.0, rc = 3.0, dxc, dyc;
2:
3: void FindTheMapping ( int w, int h )
4: {
5:   double   d, s, c;
6:   GLdouble xyc[4];
7:
8:   if ( !SetOffsFBOSize ( fbo, w*MAGAA, h*MAGAA ) )
9:     ExitOnError ( "FindTheMapping" );
10:  width = (double)(wdt = w);  height = (double)(hgh = h);
11:  d = sqrt ( width*width + height*height );
12:  s = height/d;  c = width/d;
13:  dxc = rc*c;    dyc = rc*s;
14:  xyc[0] = xc - dxc;  xyc[1] = xc + dxc;
15:  xyc[2] = yc - dyc;  xyc[3] = yc + dyc;
16:  glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
17:  glBufferSubData ( GL_UNIFORM_BUFFER, trofs[0], 4*sizeof(GLdouble), xyc );
18:  NotifyRectangle ();
19:  mappingchanged = true;
20:  ExitIfGLError ( "FindTheMapping" );
21: } /*FindTheMapping*/
```

Listing F.10. Procedury wybierania prostokąta

C

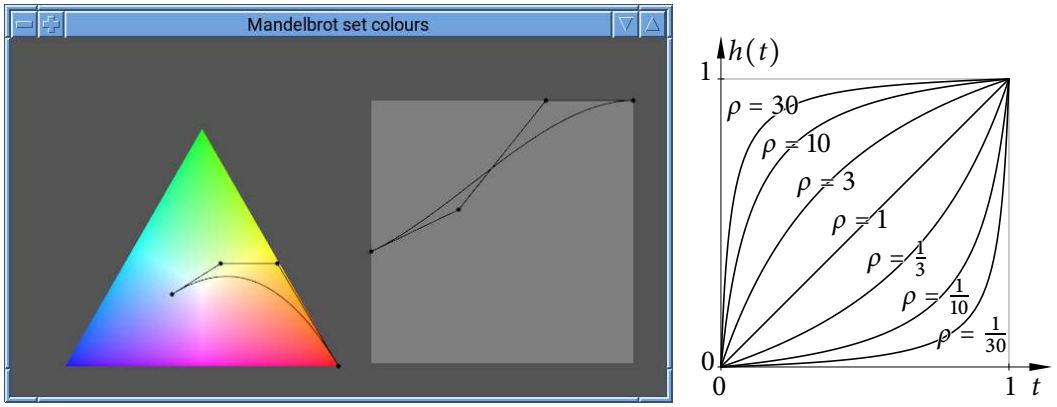
```
static int zoomcnt = 0;

char Zoom ( char zoom_in )
{
  int    cnt;
  double r;

  if ( zoom_in ) {
    if ( (r = 3.0*pow ( 1.05, ++cnt )) < MIN_RC )
      return false;
  }
  else {
    if ( (r = 3.0*pow ( 1.05, --cnt )) > MAX_RC )
      return false;
  }
  rc = r;  zoomcnt = cnt;
  FindTheMapping ( wdt, hgh );
  return true;
} /*Zoom*/

char Pan ( double dx, double dy )
{
  xc -= (dx/width)*2.0*dxc;  yc += (dy/height)*2.0*dyc;
  FindTheMapping ( wdt, hgh );
  return true;
} /*Pan*/
```

Procedury na listingu F.10 umożliwiają zmienianie promienia koła i środka prostokąta. Parametr procedury `Zoom` określa, czy należy wykonać najazd, tj. powiększyć fragment obrazu przez zmniejszenie promienia okręgu. Parametry procedury `Pan` są współrzędnymi wektora przesunięcia kursora w oknie — na ich podstawie procedura oblicza przesunięcie w płaszczyźnie zespolonej i znajduje środek nowego prostokąta.

F.1.6. Paleta i wymierne krzywe Béziera

Rysunek F.2 przedstawia okno z przykładowymi wihajstrami umożliwiającymi interakcyjne zmienianie palety; przypomnijmy, że jej zadaniem jest przyporządkowanie każdej liczbie $f(c) \in [1, N]$ wektora o współrzędnych $r, g, b \in [0, 1]$ reprezentującego kolor do nadania pikselom, dla których szader fragmentów w pierwszym etapie przerwał obliczenia po wyznaczeniu $k = \lfloor f(c) \rfloor$ elementów ciągu zespolonego. Wihajster z lewej strony przedstawia trójkąt, na którego tle jest narysowana płaska krzywa Béziera i jej łamana kontrolna. Kolor każdego piksela trójkąta jest opisany przez wektor (R, G, B), którego współrzędne są jednorodnymi współrzędnymi barycentrycznymi (zobacz podrozdz. 5.3) odpowiedniego punktu w układzie określonym przez wierzchołki trójkąta. Największa współrzędna jest równa 1, dzięki czemu

każdy punkt trójkąta ma maksymalną jasność możliwą do otrzymania przy ustalonym odcieniu. Użytkownik aplikacji może dowolnie przesuwać punkty kontrolne krzywej, sprawiając, że przechodzi ona przez punkty o wybranych odcieniach.



Rysunek F.2. Obraz krzywej Béziera określającej paletę i wykresy funkcji h

Wihajster z prawej strony wyświetla krzywą Béziera, która jest wykresem funkcji skalarnej (wielomianu) $z(t)$ opisującej jasność kolorów; punkty kontrolne tej krzywej można przesuwać do dołu i do góry, aby jasność zmniejszyć lub zwiększyć. Współrzędne R, G punktów płaskiej krzywej i wartość funkcji skalarnej opisanych wyżej opisują współrzędne x, y, z *wielomianowej* krzywej Béziera $\boldsymbol{q}$ położonej w przestrzeni trójwymiarowej.

Opisana w p. F.1.3 krzywa wymierna $\boldsymbol{p}$ powstaje przez reparametryzację krzywej $\boldsymbol{q}$: jest $\boldsymbol{p}(t) = \boldsymbol{q}\big(h(t)\big)$, przy czym h jest to funkcja homograficzna skonstruowana tak, aby spełniała następujące warunki: ma być rosnąca w przedziale $[0,1]$ i ma być $h(0) = 0$ i $h(1) = 1$. Każda taka funkcja ma postać

$$h(t) = \frac{\rho t}{\rho t + (1-t)},$$

z dodatnim parametrem ρ.

Podstawiając funkcję h w miejsce parametru krzywej Béziera $\boldsymbol{q}$, której punkty kontrolne oznaczymy $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_n$ (zobacz p. 15.1), dostaniemy parametryzację

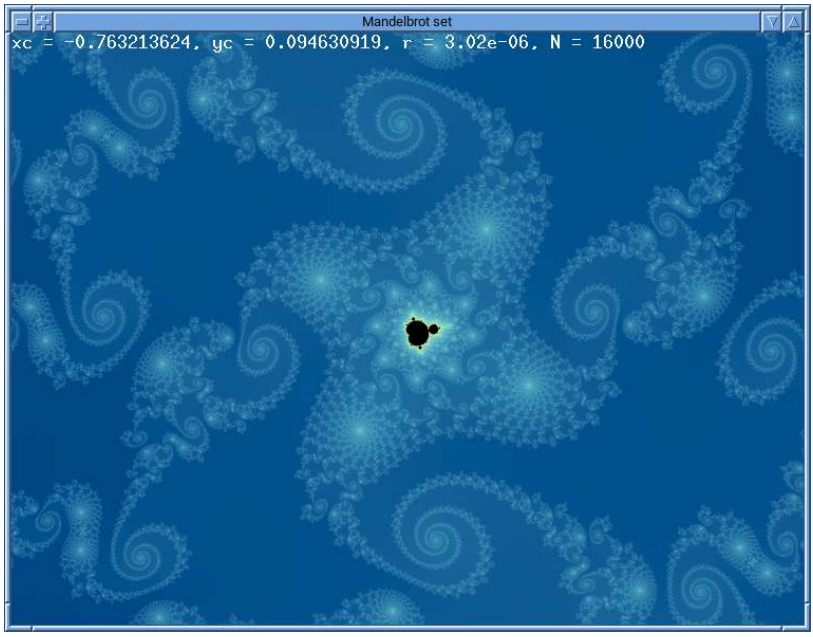
$$\boldsymbol{p}(t) = \sum_{i=0}^{n} \boldsymbol{p}_i \binom{n}{i} h(t)^i \big(1-h(t)\big)^{n-i} = \sum_{i=0}^{n} \boldsymbol{p}_i \binom{n}{i} \frac{\rho^i t^i (1-t)^{n-i}}{\big(\rho t + (1-t)\big)^n}.$$

Wszystkie składniki mają wspólny mianownik, równy $\big(\rho t + (1-t)\big)^n = \sum_{i=0}^{n} \rho^i \binom{n}{i} t^i (1-t)^i = \sum_{i=0}^{n} \rho^i B_i^n(t)$, zatem

$$\boldsymbol{p}(t) = \frac{\sum_{i=0}^{n} \rho^i \boldsymbol{p}_i B_i^n(t)}{\sum_{i=0}^{n} \rho^i B_i^n(t)}.$$

Wektorem współrzędnych jednorodnych punktu $\boldsymbol{p}(t)$ jest więc punkt $\boldsymbol{P}(t)$ krzywej Béziera, której punkty kontrolne $\boldsymbol{P}_0, \ldots, \boldsymbol{P}_n$ są wektorami współrzędnych jednorodnych punktów $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_n$, takimi że współrzędna wagowa W_i wektora $\boldsymbol{P}_i$ jest równa ρ^i. Punkt $\boldsymbol{p}(t)$ znajduje pokazana na listingu F.7 procedura `BCHorner3Rf`, która najpierw oblicza wektor $\boldsymbol{P}(t)$, a potem (w linii 29) dokonuje przejścia od współrzędnych jednorodnych do kartezjańskich.

Do manipulowania wartością parametru ρ najlepiej nadaje się rolka myszy; w odpowiedzi na komunikat o jej obróceniu wartość zmiennej przechowującej ten parametr (początkowo 1) należy pomnożyć lub podzielić na przykład przez czynnik 1.05. Użytecznym zakresem wartości parametru ρ okazał się przedział $[10^{-4}, 10^4]$. Rysując wihajster taki jak w oknie na rysunku F.2, lepiej jest używać reprezentacji, w której współrzędne wagowe wszystkich punktów kontrolnych są równe 1 (są one przechowywane w tablicy `bpcp`). Więcej wiadomości na temat wizualizacji zbioru Mandelbrota można znaleźć w sieci, polecam stronę [61].



Rysunek F.3. Obraz fragmentu zbioru Mandelbrota i jego otoczenia

Wszystkie opisane w tym podrozdziale (i wszystkie inne) sposoby wizualizacji zbioru Mandelbrota mogą być też użyte do wykonywania obrazów **zbiorów Julii**. Dla ustalonej liczby zespolonej c zbiór Julii J_c składa się z tych punktów z_0 płaszczyzny zespolonej, dla których ciąg nieskończony określony wzorami $z_0 = 0$, $z_{k+1} = z_k^2 + c$ jest ograniczony. Liczba c jest zatem *ta sama* dla wszystkich punktów, dla których wykonuje obliczenia szader łatwy do otrzymania przez modyfikację szadera z listingu F.3, i można ją przekazać w zmiennej jednolitej. Aplikacja może liczbę c przeczytać z pliku lub z klawiatury, ale znacznie lepszym pomysłem jest umożliwienie użytkownikowi wskazywania punktu c w dodatkowym oknie, w którym jest wyświetlany zbiór Mandelbrota (lub powiększony fragment tego zbioru).

F.2. Piramida Sierpińskiego i gąbka Mengera

Znana konstrukcja zbioru Cantora — dzielimy odcinek na trzy równe części, usuwamy część środkową (zostawiając jej końce) i powtarzamy w nieskończoność to samo z odcinkami, które pozostały — ma swoje uogólnienia w dwóch, trzech i większej liczbie wymiarów. Narysujemy dwie najbardziej znane figury trójwymiarowe otrzymane tą metodą.

Piramida Sierpińskiego powstaje z czworościanu foremnego przez usunięcie ośmiościanu foremnego, którego wierzchołki są środkami krawędzi tego czworościanu. W wyniku otrzymamy cztery czworościany, z których każdy jest obrazem czworościanu danego w jednokładności o współczynniku skali 1/2 i o środku w jednym z jego wierzchołków. Czworościany te drążymy dalej w ten sam sposób.

Na obrazach przedstawiamy oczywiście przybliżenia piramidy otrzymane po skończenie wielu krokach usuwania; ponieważ takie przybliżenie składa się z czworościanów, których ściany są trójkątami, możemy je oświetlić. Zauważmy jednak, że liczba czworościanów zależy wykładniczo od liczby N wykonanych iteracji — jest ona równa 4^N, czyli dla $N = 12$ mamy ponad 16 milionów czworościanów. Nie ma sensu tworzenie tablic z wszystkimi ich wierzchołkami; znacznie lepiej i prościej jest do ich wygenerowania zatrudnić GPU.

Szader geometrii z listingu F.11 oblicza i wyprowadza ściany jednego czworościanu; program z tym szaderem trzeba wywołać za pomocą procedury `glDrawArraysInstanced`, każąc jej narysować *jeden punkt* w 4^N instancjach. Szader wierzchołków musi przekazać tylko numer instancji (otrzymany w zmiennej `gl_InstanceID`), przy czym przed przystąpieniem do rysowania należy nadać zmiennej jednolitej `level` wartość N.

Listing F.11. Szader geometrii do rysowania piramidy Sierpińskiego

GLSL

```
#version 440

#define A 0.57735027  /* sqrt(1/3) */

layout(points) in;
layout(triangle_strip,max_vertices=12) out;

in int instanceID[];

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

uniform TransBlock {
    mat4 mm, mmti, vm, pm, vpm;
    vec4 eyepos;
  } trb;

uniform int level;

const vec4 p[4] = { vec4(-A,-A,-A,1.0), vec4(A,A,-A,1.0),
```

```
22:                       vec4(-A,A,A,1.0), vec4(A,-A,A,1.0) };
23: const vec3 nv[4] = { vec3(-A,A,-A), vec3(-A,-A,A),
24:                        vec3(A,-A,-A), vec3(A,A,A) };
25: const vec3 col[4] = { vec3(1.0,0.0,0.0), vec3(0.0,1.0,0.0),
26:                       vec3(0.0,0.0,1.0), vec3(0.5,0.5,0.5) };
27: const int t[4][3] = {{1,2,3},{0,2,3},{0,1,3},{0,1,2}};
28: const int f[4][3] = {{0,1,2},{0,2,3},{0,3,1},{3,2,1}};
29:
30: void main ( void )
31: {
32:   vec4 q[4], pos[4];
33:   vec3 c[4], n[4];
34:   int  i, j, k, l;
35:
36:   for ( i = 0; i < 4; i++ ) { q[i] = p[i];  c[i] = col[i]; }
37:   for ( i = 0, j = instanceID[0];  i < level;  i++, j /= 4 ) {
38:     k = j % 4;
39:     for ( l = 0; l < 3; l++ )
40:       q[t[k][l]] = 0.5*(q[k] + q[t[k][l]]);
41:     for ( l = 0; l < 3; l++ )
42:       c[t[k][l]] = 0.5*(c[k] + c[t[k][l]]);
43:   }
44:   for ( i = 0; i < 4; i++ ) {
45:     n[i] = normalize ( mat3(trb.mmti) * nv[i] );
46:     pos[i] = trb.mm * q[i];
47:     q[i] = trb.vpm * pos[i];
48:   }
49:   for ( i = 0; i < 4; i++ ) {
50:     for ( j = 0; j < 3; j++ ) {
51:       k = f[i][j];
52:       gl_Position = q[k];
53:       Out.Position = pos[k].xyz;
54:       Out.Colour = c[k];
55:       Out.Normal = Out.TNormal = n[i];
56:       EmitVertex ();
57:     }
58:     EndPrimitive ();
59:   }
60: } /*main*/
```

Położenia i kolory wierzchołków „dużego” czworościanu są podane w tablicach `p` i `col`. W linii 36 dane te są kopiowane do tablic roboczych, po czym w pętli w liniach 37–43 następuje N iteracji przekształcenia czworościanu. W każdej iteracji, na podstawie numeru instancji jest wybierany (i przypisywany zmiennej `k`) numer wierzchołka, który ma być środkiem jednokładności. Posługując się indeksami z pomocniczej tablicy `t`, szader zastępuje pozostałe trzy wierzchołki środkami odpowiednich krawędzi czworościanu, czyli ich obrazami w tej jednokładności. W taki sam sposób są przetwarzane (interpolowane) kolory wierzchołków.

Pętla w liniach 44–48 ma na celu przejście od układu współrzędnych modelu do układów świata i kostki standardowej. Jednokładności zachowują wektory normalne ścian czworościanu, zatem w linii 45 przekształceniu (przejściu do układu świata i normalizacji) są poddawane wektory wzięte z tablicy `nv`. Ściany są wyprowadzane w pętli w liniach 49–59; w każdym przebiegu wewnętrznej pętli jest wyprowadzany jeden wierzchołek trójkąta.

Gąbka Mengera powstaje przez podzielenie sześcianu na 27 sześcianów 3 razy mniejszych i odrzuceniu tych sześcianów, których żadna krawędź nie leży na krawędzi sześcianu oryginalnego. Pozostałe 20 sześcianów w ten sam sposób przekształca się dalej. Praktyczne ograniczenie liczby iteracji w konstruowaniu przybliżenia gąbki Mengera jest znacznie silniejsze, bo w każdej iteracji liczba sześcianów rośnie dwudziestokrotnie, więc po wykonaniu 5 iteracji dostaniemy 3200000 sześcianów i na tym wypadałoby poprzestać.

Implementując przekształcenia sześcianu, którego krawędzie są równoległe do osi układu współrzędnych (modelu), skorzystamy z faktu, że każda potrzebna jednokładność może być zrealizowana przez niezależne przekształcenie współrzędnych x, y, z. „Skompresowana" informacja o przekształceniach jest umieszczona w tablicy `t`; jeśli w danej iteracji sześcian ma być poddany przekształceniu o numerze $k \in \{0, \ldots, 19\}$, to liczby `t[k][0]`, `t[k][1]` i `t[k][2]` oznaczają odpowiednio numery trzech przekształceń, którym należy poddać te współrzędne. Aby umożliwić przetwarzanie współrzędnych w pętli, zamiast nadać zmiennym `p0` i `p1` typ `vec3`, zadeklarowałem je jako tablice liczb typu `float`.

Niech s oznacza współrzędną x, y lub z; indeks `l` jest numerem tej współrzędnej. Jej przedział zmienności $[s_0, s_1]$ dla poddawanego przekształceniu sześcianu, zależnie od liczby `t[k][l]` należy zastąpić przez $[s_0, a]$, $[a, b]$ lub $[b, s_1]$, gdzie liczby a i b dzielą przedział $[s_0, s_1]$ odpowiednio w 1/3 i 2/3 jego długości; odpowiednie obliczenia są wykonywane w liniach 46, 49–51 albo 54. Zwracam uwagę, że wzory zaprogramowane w liniach 46 i 49 oraz 50 i 54 są identyczne, dzięki czemu wyniki obliczeń liczb a i b na danym poziomie podziału są obarczone identycznymi błędami zaokrągleń. Trzeba dbać o takie szczegóły.

W liniach 58–61 z liczb znajdujących się w tablicach `p0` i `p1` składane jest 8 wierzchołków sześcianu, które zostają zapamiętane w tablicy `p`. Czwórki indeksów do tej tablicy, razem z odpowiednimi wektorami normalnymi ścian sześcianu, są przekazywane procedurze `OutputFacet`, której zadaniem jest przekazanie na wyjście szadera jednej ściany sześcianu. Warunki sprawdzane przed każdym wywołaniem tej procedury mają na celu pominięcie ścian, które nie mogą być widoczne na obrazie. Po pierwsze, jeśli ostatnie przekształcenie współrzędnej s zamieniło przedział $[s_0, s_1]$ na $[a, b]$ (czyli odrzuciło początek i koniec przedziału), to ściany w płaszczyznach $s = a$ i $s = b$ przylegają do ścian innych sześcianów. Po drugie, szader odrzuca ściany odwrócone tyłem do obserwatora.

Wektor normalny każdej ściany sześcianu ma kierunek jednej z osi układu współrzędnych modelu. Dzięki temu wystarczy sprawdzić, czy różnica współrzędnej s położenia obserwatora i dowolnego punktu na ścianie ma taki sam znak jak współrzędna s (zorientowanego na zewnątrz sześcianu) wektora normalnego tej ściany. Aby umożliwić te testy, w linii 62 szader oblicza położenie obserwatora w układzie współrzędnych modelu. W zmiennej `trb.mmti` jest przechowywana potrzebna do przekształcenia wektorów normalnych transpozycja od-

Listing F.12. Szader geometrii do rysowania gąbki Mengera

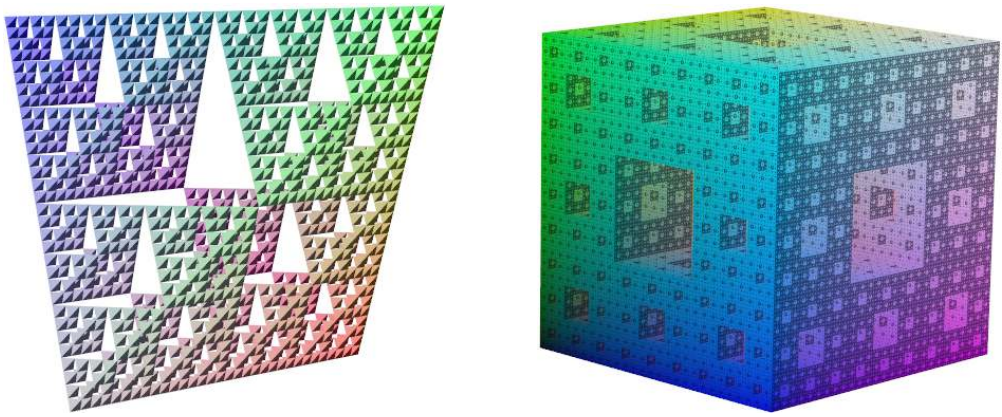
GLSL

```
#version 440

.... /* linie 3-19 takie same, jak na listingu F.11 */

const int t[21][3] =
  {{0,0,0},{1,0,0},{2,0,0},{0,1,0},{2,1,0},{0,2,0},{1,2,0},
   {2,2,0},{0,0,1},{2,0,1},{0,2,1},{2,2,1},{0,0,2},{1,0,2},
   {2,0,2},{0,1,2},{2,1,2},{0,2,2},{1,2,2},{2,2,2},{0,0,0}};
const int ip0[4] = {0,3,4,7}, ip1[4] = {1,5,2,6}, ip2[4] = {1,0,5,4},
          ip3[4] = {2,6,3,7}, ip4[4] = {1,2,0,3}, ip5[4] = {5,4,6,7};

vec3 p[8];

void OutputVertex ( vec3 p, vec3 nv )
{
  vec4 q;

  Out.Colour = vec3( p.x/(2*A)+0.5, p.y/(2.0*A)+0.5, p.z/(2.0*A)+0.5 );
  Out.Position = (q = trb.mm*vec4(p,1.0)).xyz;
  Out.Normal = Out.TNormal = mat3(trb.mmti) * nv;
  gl_Position = trb.vpm * q;
  EmitVertex ();
} /*OutputVertex*/

void OutputFacet ( int ip[4], vec3 nv )
{
  int i;

  for ( i = 0; i < 4; i++ )
    OutputVertex ( p[ip[i]], nv );
  EndPrimitive ();
} /*OutputFacet*/

void main ( void )
{
  float p0[3], p1[3], a;
  vec4  epm;
  int   i, j, k, l;

  p0[0] = p0[1] = p0[2] = -A;  p1[0] = p1[1] = p1[2] = A;
  for ( i = 0, j = instanceID[0];  i < level;  i++, j /= 20 ) {
    k = j % 20;
    for ( l = 0; l < 3; l++ )
      switch ( t[k][l] ) {
    case 0:
```

```
46:         p1[l] = (2.0*p0[l] + p1[l])/3.0;
47:         break;
48:     case 1:
49:         a = (2.0*p0[l] + p1[l])/3.0;
50:         p1[l] = (p0[l] + 2.0*p1[l])/3.0;
51:         p0[l] = a;
52:         break;
53:     case 2:
54:         p0[l] = (p0[l] + 2.0*p1[l])/3.0;
55:         break;
56:       }
57:   }
58:   p[0] = vec3(p0[0],p0[1],p0[2]);  p[1] = vec3(p1[0],p0[1],p0[2]);
59:   p[2] = vec3(p1[0],p1[1],p0[2]);  p[3] = vec3(p0[0],p1[1],p0[2]);
60:   p[4] = vec3(p0[0],p0[1],p1[2]);  p[5] = vec3(p1[0],p0[1],p1[2]);
61:   p[6] = vec3(p1[0],p1[1],p1[2]);  p[7] = vec3(p0[0],p1[1],p1[2]);
62:   epm = transpose(trb.mmti) * trb.eyepos;
63:   if ( level == 0 ) k = 20;
64:   if ( t[k][0] != 1 && epm.x - p0[0] < 0.0 )
65:     OutputFacet ( ip0, vec3(-1.0,0.0,0.0) );
66:   if ( t[k][0] != 1 && epm.x - p1[0] > 0.0 )
67:     OutputFacet ( ip1, vec3(1.0,0.0,0.0) );
68:   if ( t[k][1] != 1 && epm.y - p0[1] < 0.0 )
69:     OutputFacet ( ip2, vec3(0.0,-1.0,0.0) );
70:   if ( t[k][1] != 1 && epm.y - p1[1] > 0.0 )
71:     OutputFacet ( ip3, vec3(0.0,1.0,0.0) );
72:   if ( t[k][2] != 1 && epm.z - p0[2] < 0.0 )
73:     OutputFacet ( ip4, vec3(0.0,0.0,-1.0) );
74:   if ( t[k][2] != 1 && epm.z - p1[2] > 0.0 )
75:     OutputFacet ( ip5, vec3(0.0,0.0,1.0) );
76: } /*main*/
```

wrotności macierzy przejścia od układu modelu do układu świata, czyli transpozycja macierzy przejścia od układu świata do układu modelu. Dlatego w tym miejscu jest użyta funkcja `transpose`, która likwiduje skutki transpozycji wykonanej przez CPU. Z sześciu ścian sześcianu odwrócone przodem do obserwatora są zawsze tylko jedna, dwie lub trzy, zatem szader wyprowadzi co najwyżej trzy taśmy trójkątowe z czterema wierzchołkami (reprezentujące po dwie trójkątne połówki ścian). Dlatego limit liczby wierzchołków podany w kwalifikatorze wyjścia tego szadera jest też równy 12 (zob. listing F.11, linia 6). Procedura `OutputFacet` wyprowadza jedną taką taśmę, za pomocą procedury `OutputVertex`. W linii 18 jest obliczany kolor wierzchołka — przez poddanie położenia wierzchołka takiemu przekształceniu, które sześcian $[-\sqrt{1/3}, \sqrt{1/3}]^3$ przeprowadza na kostkę jednostkową $[0,1]^3$. W linii 19 następuje przejście od układu modelu do układu świata, w linii 20 obliczany jest wektor normalny ściany w układzie świata, a w linii 21 położenie wierzchołka jest przekształcane do układu współrzędnych kostki standardowej.

Za wygląd obiektów na obrazie odpowiada szader fragmentów, który może stosować dowolny model oświetlenia i może też realizować algorytm cieni. Obrazy na rysunku F.4 zostały wykonane przy użyciu szadera z listingów 18.1–18.4.



Rysunek F.4. Obrazy piramidy Sierpińskiego i gąbki Mengera

Rozszerzymy zadanie, otrzymując obrazy przecięcia gąbki (a raczej jej przybliżeń) z półprzestrzenią. W tym celu trzeba określić płaszczyznę, która jest brzegiem tej półprzestrzeni i poddać ściany rysowanych sześcianów obcinaniu tą płaszczyzną, ale to nie wystarczy: trzeba jeszcze narysować ściany brył otrzymanych z przeciętych sześcianów. Użyjemy do tego programów z dwoma nowymi szaderami geometrii; szadery wierzchołków i fragmentów pozostaną te same.

Listing F.13 przedstawia szader powodujący obcinanie ścian sześcianów; jest do niego dodana deklaracja tablicy `gl_ClipDistance`, do której są wpisywane odległości wierzchołków od płaszczyzny obcinającej i blok zmiennych jednolitych opisujący tę płaszczyznę. Pole `nv` w tym bloku przechowuje wektor normalny $\boldsymbol{n}$, a w tablicy `square` są podane wierzchołki $\boldsymbol{w}_0, \ldots, \boldsymbol{w}_3$ leżącego w tej płaszczyźnie czworokąta; każdy z nich, razem z wektorem normalnym, wyznacza tę płaszczyznę jednoznacznie.

Listing F.13. Pierwszy szader geometrii do rysowania obciętej gąbki Mengera

GLSL

```
#version 440

.... /* linie 3-19 takie jak na listingu F.11 */

out float gl_ClipDistance[1];

uniform ClipPlane {
    vec4 nv;
    vec4 square[4];
  } clp;
```

```
11:
12: vec3  p[8];
13: float c[8];
14:
15: void OutputVertex ( vec3 p, vec3 nv, float c )
16: {
17:   .... /* początek procedury bez zmian */
18:   gl_ClipDistance[0] = c;
19:   EmitVertex ();
20: } /*OutputVertex*/
21:
22: void OutputFacet ( int ip[4], vec3 nv )
23: {
24:   int i;
25:
26:   for ( i = 0; i < 4; i++ )
27:     OutputVertex ( p[ip[i]], nv, c[ip[i]] );
28:   EndPrimitive ();
29: } /*OutputFacet*/
30:
31: void main ( void )
32: {
33:   .... /* zmienne lokalne procedury main bez zmian */
34:   bool  pos;
35:
36:   .... /* obliczanie wierzchołków sześcianu bez zmian */
37:   for ( i = 0, pos = false;  i < 8;  i++ ) {
38:     c[i] = dot ( clp.nv.xyz, p[i]-clp.square[0].xyz );
39:     if ( c[i] > 0.0 ) pos = true;
40:   }
41:   if ( !pos ) return;
42:   epm = transpose(trb.mmti) * trb.eyepos;
43:   .... /* wyprowadzanie ścian sześcianu bez zmian */
44: } /*main*/
```

Obliczenie wierzchołków sześcianu jest wykonywane tak samo jak przez szader z listingu F.12. Zadaniem instrukcji w liniach 37–40 jest obliczenie odległości ze znakiem wierzchołków od płaszczyzny obcinającej, a dokładniej, dla wierzchołka $\boldsymbol{p}_i$ jest znajdowana liczba $c_i = \langle \boldsymbol{n}, \boldsymbol{p}_i - \boldsymbol{w}_0 \rangle$. Jeśli żadna liczba c_i nie jest dodatnia, szader kończy działanie, oszczędzając niepotrzebnej pracy etapowi obcinania w potoku przetwarzania grafiki. Modyfikacje procedur `OutputFacet` i `OutputVertex` mają na celu wyprowadzenie razem z wierzchołkiem $\boldsymbol{p}_i$ liczby c_i.

Listing F.14 przedstawia szader geometrii, którego zadaniem jest znalezienie i wyprowadzenie wielokątów otrzymanych z przecięcia sześcianów, z których składa się gąbka, płaszczyzną obcinającą. Taki wielokąt musi być podzielony na trójkąty i wyprowadzony w postaci taśmy trójkątowej; może ona mieć co najwyżej 6 wierzchołków.

Listing F.14. Drugi szader geometrii do rysowania obciętej gąbki Mengera

GLSL

```
#version 440

.... /* zmienne interfejsu takie jak na listingu F.12 */
uniform ClipPlane { .... } clp;  /* listing F.13 */

vec3 cp[20];

int SHClip ( vec3 nv, vec3 p, int n, int k, int l )
{
  vec3  s, t;
  float ds, dt;
  int   i, m;
#define OUTPUT(P) { cp[l+m] = P;  m ++; }

  s = cp[k+n-1];  ds = dot ( nv, q-s );
  for ( i = m = 0; i < n; i++ ) {
    t = cp[k+i];  dt = dot ( nv, q-t );
    if ( ds >= 0.0 ) {
      if ( dt >= 0.0 ) OUTPUT ( t )
      else OUTPUT ( mix ( s, t, ds/(ds-dt) ) )
    }
    else {
      if ( dt >= 0.0 ) {
        OUTPUT ( mix ( s, t, ds/(ds-dt) ) )
        OUTPUT ( t )
      }
    }
    s = t;  ds = dt;
  }
  return m;
#undef OUTPUT
} /*SHClip*/

void OutputVertex ( vec3 p )
{
  vec4 q;

  Out.Colour = vec3( p.x/(2*A)+0.5, p.y/(2.0*A)+0.5, p.z/(2.0*A)+0.5 );
  Out.Position = (q = trb.mm*vec4(p,1.0)).xyz;
  Out.Normal = Out.TNormal = mat3(trb.mmti) * clp.nv.xyz;
  gl_Position = trb.vpm * q;
  EmitVertex ();
} /*OutputVertex*/

void OutputPolygon ( int n )
```

```
46: {
47:   int i, k;
48:
49:   for ( i = 0, k = n-1;  i < k;  i++, k- ) {
50:     OutputVertex ( cp[i] );
51:     OutputVertex ( cp[k] );
52:   }
53:   if ( i == k )
54:     OutputVertex ( cp[i] );
55:   EndPrimitive ();
56: } /*OutputPolygon*/
57:
58: void main ( void )
59: {
60:   .... /* zmienne jak na listingu F.12 */
61:   float c;
62:   bool  pos, neg;
63:
64:   .... /* obliczanie wierzchołków sześcianu bez zmian */
65:   for ( i = 0, pos = neg = false;  i < 8;  i++ ) {
66:     c = dot ( clp.nv.xyz, p[i]-clp.square[0].xyz );
67:     if ( c > 0.0 ) pos = true;
68:     else if ( c < 0.0 ) neg = true;
69:   }
70:   if ( !(pos && neg) ) return;
71:   for ( i = 0; i < 4; i++ )
72:     cp[i] = clp.square[i].xyz;
73:   i = SHClip ( vec3(-1.0,0.0,0.0), p[0], 4, 0, 10 );
74:   i = SHClip ( vec3(1.0,0.0,0.0),  p[1], i, 10, 0 );
75:   i = SHClip ( vec3(0.0,-1.0,0.0), p[1], i, 0, 10 );
76:   i = SHClip ( vec3(0.0,1.0,0.0),  p[2], i, 10, 0 );
77:   i = SHClip ( vec3(0.0,0.0,-1.0), p[1], i, 0, 10 );
78:   i = SHClip ( vec3(0.0,0.0,1.0),  p[5], i, 10, 0 );
79:   if ( i > 2 )
80:     OutputPolygon ( i );
81: } /*main*/
```

Procedura `main`, na podstawie numeru instancji sześcianu, oblicza jego wierzchołki tak samo jak w szaderach na listingach F.12 i F.13. W liniach 65–69 są obliczane odległości ze znakiem wierzchołków od płaszczyzny obcinającej, po czym, jeśli wszystkie one mają ten sam znak, to szader kończy działanie. Jeśli zaś płaszczyzna obcinająca przecina sześcian, to przecięcie jest wyznaczane. Czworokąt, którego wierzchołki są podane w tablicy `square`, musi być tak duży, aby zawierał przecięcie płaszczyzny z całą gąbką. Przecięcie jest znajdowane za pomocą opisanego w p. 19.8.7 algorytmu Sutherlanda-Hodgmana obcinania wielokąta do półprzestrzeni; sześcian (część przybliżenia gąbki określona przez numer instancji) jest przecięciem sześciu półprzestrzeni, a zatem procedurę obcinania trzeba wywołać sześciokrotnie.

Parametrami procedury obcinania są wektor normalny i punkt płaszczyzny obcinającej (którą w tym przypadku jest płaszczyzna zawierająca ścianę sześcianu), liczba wierzchołków obcinanego wielokąta i dwa indeksy do globalnej tablicy cp, w której są przechowywane te wierzchołki. Tablica ta jest podzielona na połowy; wierzchołki są odczytywane z jednej połowy (której początek jest wskazywany przez pierwszy indeks) i zapisywane w drugiej połowie; w następnym wywołaniu procedury role tych połówek są zamieniane. Choć podczas obcinania dużego czworokąta do sześcianu nie pojawi się więcej niż 6 wierzchołków, tablica cp ma długość 20; jest to zabezpieczenie na wypadek, gdyby (wskutek zmian aplikacji) podany w tablicy square czworokąt stał się za mały i wynik jego obcinania mógł mieć nawet 10 wierzchołków[10].

W liniach 71–72 wierzchołki czworokąta są przepisywane do pierwszej połowy tablicy cp. Procedura SHClip jest podobna do tej z listingu 19.15, ale w tym przypadku obliczenie jest wykonywane we współrzędnych kartezjańskich. Wzór opisujący parametr t punktu przecięcia krawędzi $\boldsymbol{st}$ wielokąta z płaszczyzną obcinającą ma postać

$$t = \langle \boldsymbol{q} - \boldsymbol{s}, \boldsymbol{n} \rangle / \langle \boldsymbol{t} - \boldsymbol{s}, \boldsymbol{n} \rangle = d_{\boldsymbol{s}} / (d_{\boldsymbol{s}} - d_{\boldsymbol{t}}),$$

w której liczby $d_{\boldsymbol{s}}$ i $d_{\boldsymbol{t}}$ są odległościami ze znakiem punktów $\boldsymbol{s}$ i $\boldsymbol{t}$ od płaszczyzny obcinającej.

Procedura OutputPolygon wyprowadza wynik obcinania. Jego wierzchołki są uporządkowane w kolejności występowania na brzegu. Gdyby była możliwość wyprowadzenia wachlarza trójkątów przez szader geometrii, to taka kolejność byłaby odpowiednia. Ale szader może wyprowadzić tylko taśmę trójkątową. Instrukcje procedury OutputPolygon realizują kolejność wyprowadzania wierzchołków odpowiednią dla taśmy.

Przedstawiona na listingu F.15 procedura wykonuje obraz obciętej gąbki przy użyciu programów z szaderami opisanymi wyżej. Liczba instancji obiektu (tj. sześcianów, z których składa się przybliżenie gąbki) jest wyznaczana tak jak dla gąbki nieobciętej. Podczas rysowania ścian sześcianów (w linii 16) obcinanie ma być włączone, ale jest ono wyłączane w linii 17, przed rysowaniem przekroju gąbki.

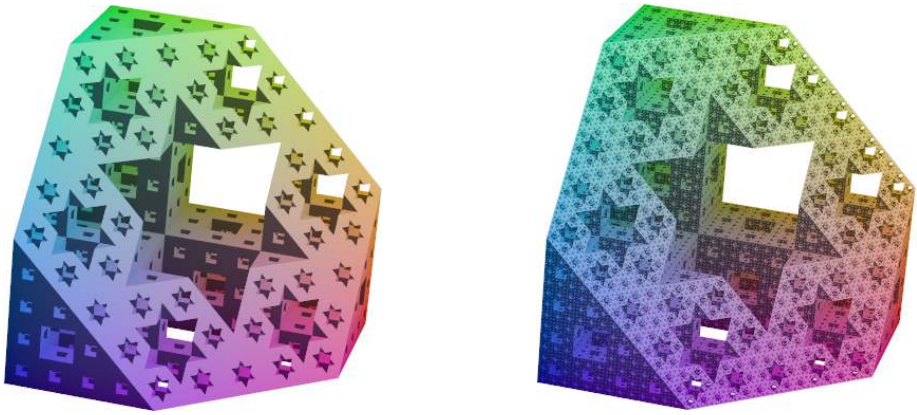
Listing F.15. Procedura rysowania obciętej gąbki Mengera

C

```
1:  void DrawClippedSponge ( void )
2:  {
3:    GLint ninst;
4:    int   i;
5:
6:    glBindVertexArray ( empty_vao );
7:    for ( i = 1, ninst = 20;  i < level[1];  i++ )
8:      ninst *= 20;
9:    glEnable ( GL_CLIP_DISTANCE0 );
10:   glEnable ( GL_CULL_FACE );
11:   glCullFace ( GL_FRONT );
```

[10] Czasami trzeba dmuchać na zimne i moim zdaniem to jest taka sytuacja. Z każdym obcinaniem wielokąta wypukłego do półprzestrzeni liczba wierzchołków może się zwiększyć co najwyżej o 1.

```
12:   glFrontFace ( GL_CCW );
13:   glUseProgram ( program_id[2] );  /* program z szaderem z listingu F.13 */
14:   glUniform1i ( levelloc[2], level[1] );
15:   glUniform1i ( ColourSourceLoc[2], 0 );
16:   glDrawArraysInstanced ( GL_POINTS, 0, 1, ninst );
17:   glDisable ( GL_CLIP_DISTANCE0 );
18:   glUseProgram ( program_id[3] );  /* program z szaderem z listingu F.14 */
19:   glUniform1i ( levelloc[3], level[1] );
20:   glUniform1i ( ColourSourceLoc[3], 0 );
21:   glDrawArraysInstanced ( GL_POINTS, 0, 1, ninst );
22:   glBindVertexArray ( 0 );
23:   ExitIfGLError ( "DrawClippedSponge" );
24: } /*DrawClippedSponge*/
```



Rysunek F.5. Obrazy części wspólnej gąbki Mengera i półprzestrzeni

Rysunek F.5 przedstawia obrazy przecięć dwóch przybliżeń gąbki Mengera z półprzestrzenią $\{(x, y, z): x + y + z \leqslant 0\}$.

G

GPGPU

Przedstawiona w rozdziale 31 implementacja metody zagęszczania siatek jest przykładem zastosowania GPU do obliczeń niezwiązanych bezpośrednio z grafiką, czyli GPGPU. W tym dodatku są opisane przykłady implementacji algorytmów ogólnego stosowania, wykorzystujących moc obliczeniową pracujących równolegle procesorów GPU. Część z nich została użyta w implementacji metody bilansu energetycznego w rozdziale 29 i w procedurach zagęszczania siatek opisanych w rozdziale 31.

G.1. Działania parami

Mając dany ciąg n liczb lub wektorów, można obliczyć sumę ich wszystkich w $\lceil \log_2 n \rceil$ krokach, dodając jednocześnie pary składników, a potem pary sum częściowych. Implementacja najprostszego algorytmu sumowania parami jest pokazana na listingach G.1 i G.2. Pierwszy z nich przedstawia szader obliczeniowy, który wywołuje procedurę dodajacą do i-tego elementu w tablicy element j-ty; ta procedura, niepokazana tu, realizuje działanie odpowiednie dla konkretnych obiektów, które mogą być liczbami całkowitymi lub zmiennopozycyjnymi, ale też wektorami lub macierzami. Szczegóły budowy tablicy w pamięci GPU, w której są umieszczone obiekty (np. układ danych w buforze) i rodzaj obiektów, zna tylko ta procedura. Zadaniem procedury `main` jest wyznaczenie liczb i oraz j — pierwsza z nich jest numerem instancji szadera w grupie roboczej i jest mniejsza niż $n/2$, a druga jest o $\lceil n/2 \rceil$ większa.

Przedstawiony tu szader jest wywoływany przez procedurę `GPUSumUp`, która przedtem przywiązuje do odpowiedniego celu podany jako parametr bufor magazynowy zawierający dane i nadaje wartości zmiennym jednolitym — zmienna `n0` określa początek ciągu do zsumowania w tablicy, a zmienna `n` określa bieżącą liczbę składników. W kolejnych krokach liczba ta maleje o połowę (z zaokrąglaniem w górę), a gdy zmaleje do 1, w miejscu `n0` w tablicy jest gotowa suma wszystkich elementów ciągu.

Rozwinięciem makrodefinicji `COMPUTE` (listing 9.1) jest instrukcja złożona, w której po wywołaniu procedury `glDispatchCompute`, uruchamiającym obliczenia na GPU, następuje wywołanie procedury `glMemoryBarrier`, z której powrót następuje, gdy wszystkie operacje zapisu do buforów magazynowych są zakończone.

Listing G.1. Procedura `main` szadera sumowania parami

GLSL

```
#version 450 core

layout(local_size_x=1) in;

uniform uint n;

void AddTwoTerms ( uint i, uint j );

void main ( void )
{
  uint i, j;

  i = uint ( gl_GlobalInvocationID.x );
  if ( (j = i+(n+1)/2) < n )
    AddTwoTerms ( i, j );
} /*main*/
```

Listing G.2. Procedura sumowania parami na GPU

C

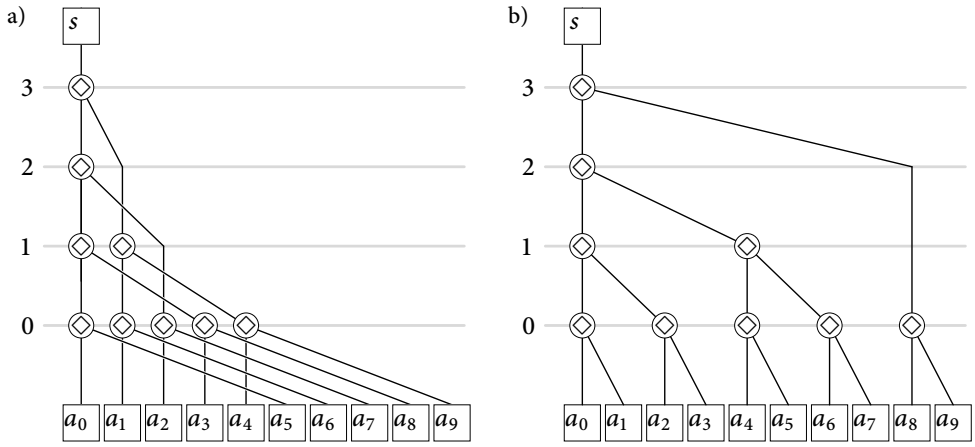
```
static GLuint program_id, GLuint uloc[2];

void GPUSumUp ( GLuint n, GLuint n0, GLuint databuf )
{
  glUseProgram ( program_id );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, databuf );
  glUniform1ui ( uloc[1], n0 );
  for ( ; n > 1; n = (n+1)/2 ) {
    glUniform1ui ( uloc[0], n );
    COMPUTE ( n/2, 1, 1 )
  }
  ExitIfGLError ( "GPUSumUp" );
} /*GPUSumUp*/
```

Wypada uczynić dwie uwagi: po pierwsze, algorytm ma spore wymagania pamięciowe, bo „psuje" początkową zawartość tablicy. Jeśli oryginalny ciąg będzie później potrzebny, to trzeba utworzyć dodatkowy bufor roboczy, skopiować do niego dane i zepsuć kopię. Po drugie, algorytm zakłada łączność i przemienność wykonywanego działania; dodawanie liczb, a także wektorów i macierzy ma te własności w algebrze i w arytmetyce stałopozycyjnej, ale nie w arytmetyce zmiennopozycyjnej, ponieważ w niej występują błędy zaokrągleń[1]. Ich

[1]Zamiana argumentów jednego dodawania nie zmieni wyniku, ale przestawienie wielu składników lub inne ich pogrupowanie (np. $a+(b+c)$ zamiast $(a+b)+c$) już może. Matematyk powiedziałby, że dodawanie zmiennopozycyjne jest działaniem łącznym z dokładnością do błędów zaokrągleń, co jest przyznaniem, że dodawanie zmiennopozycyjne *nie jest* łączne, ale stara się najlepiej jak się da.

obecność sprawia, że *obliczona* suma liczb zmiennopozycyjnych prawie zawsze jest tylko *prawie dokładną* sumą tych liczb.

Uwaga: Skutki błędów zaokrągleń mogą być interpretowane jako względne zaburzenia danych; różne algorytmy sumowania liczb zmiennopozycyjnych obliczą dokładne sumy trochę innych liczb. Wielkość tych zaburzeń zależy od względnej dokładności reprezentacji (w przybliżeniu 10^{-7} dla pojedynczej i 10^{-15} dla podwójnej precyzji) i od tzw. stałych kumulacji, które im są mniejsze, tym lepszy jest algorytm. Dla sumowania n liczb „po kolei" stałe kumulacji są rzędu n, a dla sumowania parami tylko $\log_2 n$. A więc algorytmy sumowania parami, dla dużych n, mają przewagę nad sumowaniem „po kolei" także pod tym względem.



Rysunek G.1. Drzewa działań parami realizowanych przez dwa algorytmy równoległe dla $n = 10$

Pewne działania dwuargumentowe są łączne, ale nie są przemienne. Działaniami takimi są na przykład mnożenie macierzy $n \times n$ (dla $n > 1$) i mnożenie kwaternionów. Oznaczmy takie działanie symbolem „$\diamond$". Jeśli trzeba obliczyć wyrażenie $a_0 \diamond a_1 \diamond \cdots \diamond a_{n-1}$, w którym nie wolno przestawiać argumentów, ale można (dzięki łączności) dowolnie rozmieścić nawiasy, to możemy użyć innego algorytmu równoległego, który również wykonuje $\lceil \log_2 n \rceil$ kroków. Algorytm ten jest przedstawiony na listingach G.3 i G.4; szader obliczeniowy na pierwszym listingu jest wywoływany odpowiednią liczbę razy przez procedurę z drugiego listingu. Działanie realizowane przez procedurę `AddTwoTerms` może być dowolnym działaniem łącznym[2], przy czym procedura wykonuje działanie na obiektach obecnych na pozycjach `n0+i` i `n0+j` i wpisuje wynik do tablicy na miejsce pierwszego z tych argumentów. Na rysunku G.1 są pokazane drzewa wyrażeń obliczanych przez oba opisane tu algorytmy, a także miejsca w tablicy z danymi, na których są zapisywane wyniki pośrednie. W obu przypadkach wynik końcowy zostaje na miejscu pierwszego elementu ciągu[3].

[2] ewentualnie łącznym z dokładnością do błędów zaokrągleń

[3] Gdy działanie „$\diamond$" jest dodawaniem liczb zmiennopozycyjnych, stałe kumulacji są takie same jak dla pierwszego algorytmu — od obu algorytmów sumowania parami można oczekiwać podobnej dokładności.

Listing G.3. Procedura `main` drugiego algorytmu sumowania parami

GLSL

```
#version 450 core

layout(local_size_x=1) in;

uniform uint n, q;

void AddTwoTerms ( uint i, uint j );  /* dowolne działanie łączne */

void main ( void )
{
  uint i, j;

  i = uint ( (q+q)*gl_GlobalInvocationID.x );
  if ( (j = i+q) < n )
    AddTwoTerms ( i, j );
} /*main*/
```

Listing G.4. Druga procedura sumowania parami na GPU

C

```
static GLuint program_id, GLuint uloc[3];

void GPUAltSumUp ( GLuint n, GLuint n0, GLuint databuf )
{
  GLuint q;

  glUseProgram ( program_id );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, databuf );
  glUniform1ui ( uloc[0], n );    /* uniform n = n; */
  glUniform1ui ( uloc[1], n0 );   /* uniform n0 = n0; */
  for ( q = 1;  n > 1;  q += q, n = (n+1)/2 ) {
    glUniform1ui ( uloc[2], q );  /* uniform q = q; */
    COMPUTE ( n/2, 1, 1 )
  }
} /*GPUAltSumUp*/
```

W obu przedstawionych wyżej algorytmach możemy wykonywać dowolne działanie łączne i przemienne (w drugim przypadku wystarczy tylko łączność), na przykład wybieranie mniejszego (albo większego) elementu, co umożliwia znalezienie minimalnego (albo maksymalnego) elementu ciągu w $\lceil \log_2 n \rceil$ krokach. Dokładnie tyle samo kroków wystarczy, aby oba elementy skrajne — minimalny i maksymalny — znaleźć jednocześnie.

Szader jednocześnie znajdujący elementy skrajne w danym ciągu posługuje się trzema procedurami, które mają dostęp do tablicy zawierającej ten ciąg i które są dostosowane do konkretnego rodzaju elementów, ich reprezentacji i relacji ustalającej porządek między nimi.

Jedną z tych procedur jest **komparator**, czyli procedura porównująca dwa elementy ciągu na wskazanych pozycjach i przestawiająca je tak, aby na pierwszej pozycji znalazł się element mniejszy. Pozostałe dwie procedury porównują dwa elementy i jeśli drugi element jest odpowiednio mniejszy albo większy, to przepisują go na pierwszą pozycję. Przykład takich procedur (dla ciągu liczb całkowitych) jest na listingu G.5.

Listing G.5. Procedury do znajdowania minimum i maksimum w ciągu liczb

GLSL

```
1: #version 450 core
2:
3: layout(std430,binding=0) buffer Data { int d[]; } data;
4:
5: uniform uint n, n0;
6:
7: void CompSwap ( uint i, uint j )
8: {
9:   int x;
10:
11:   if ( data.d[i += n0] > data.d[j += n0] )
12:     { x = data.d[i];  data.d[i] = data.d[j];  data.d[j] = x; }
13: } /*CompSwap*/
14:
15: void ChooseMin ( uint i, uint j )
16: {
17:   if ( data.d[i += n0] > data.d[j += n0] )
18:     data.d[i] = data.d[j];
19: } /*ChooseMin*/
20:
21: void ChooseMax ( uint i, uint j )
22: {
23:   if ( data.d[i += n0] < data.d[j += n0] )
24:     data.d[i] = data.d[j];
25: } /*ChooseMax*/
```

Procedura `main` na listingu G.6 działa nieco inaczej w pierwszym kroku niż w kolejnych. Wartość zmiennej jednolitej `n`, która musi być większa niż 1, jest liczbą elementów w tablicy. W pierwszym kroku zmienna `s` ma wartość `true`; wtedy komparator (wywołany w linii 19) porządkuje parę sąsiednich elementów, wskutek czego mniejsze elementy wszystkich par znajdą się w tablicy na pozycjach parzystych, a większe na nieparzystych. Dodatkowo jeśli długość ciągu jest nieparzysta, to ostatni jego element jest porównywany z elementami pierwszej pary i jeśli jest mniejszy od pierwszego z nich albo większy od drugiego, to jest wpisywany na odpowiednie miejsce w tej parze. Odtąd liczba miejsc w tablicy zajętych przez potrzebne dalej dane, będąca wartością zmiennej `n`, jest parzysta.

Listing G.6. Procedura `main` szadera znajdowania minimum i maksimum

GLSL

```
#version 450 core

layout(local_size_x=1) in;

uniform uint n;
uniform bool s;

void CompSwap ( uint i, uint j );
void ChooseMin ( uint i, uint j );
void ChooseMax ( uint i, uint j );

void main ( void )
{
  uint i, j;

  i = uint ( gl_GlobalInvocationID.x );  i += i;
  if ( s ) {                          /* porządkowanie par */
    if ( (j = i+1) < n )
      CompSwap ( i, j );
    if ( i == 0 && (n & 0x01) != 0 ) {  /* n nieparzyste */
      ChooseMin ( 0, n-1 );
      ChooseMax ( 1, n-1 );
    }
  }
  else if ( (j = i + 2*((n+3)/4)) < n ) {
    ChooseMin ( i, j );
    ChooseMax ( i+1, j+1 );
  }
} /*main*/
```

Gdy zmienna `s` ma wartość `false`, w tablicy znajduje się `n`/2 uporządkowanych par, przy czym któraś z tych par zawiera element najmniejszy, a inna lub ta sama para zawiera element największy danego ciągu. Wątek i-ty szadera przetwarza pary o numerach i oraz $j = i+\lceil n/4 \rceil$; mniejszy z mniejszych oraz większy z większych elementów obu par zostają zapamiętane w pierwszej parze. W ten sposób istotne dane zostają przeniesione do pierwszej połowy dotychczas istotnego fragmentu tablicy.

Procedura na listingu G.7 po zakończeniu działania zostawia elementy najmniejszy i największy na pierwszych dwóch miejscach tablicy zajmowanej początkowo przez dany ciąg, skąd aplikacja może je odczytać. Zawartość tablicy zostaje oczywście „zepsuta”. Wydaje mi się, że procedura ta nie wymaga dalszych objaśnień. Polecam ćwiczenie: napisanie szadera, który dla tablicy punktów w przestrzeni znajduje boks otaczający, tj. najmniejszy prostopadłościan o krawędziach równoległych do osi układu współrzędnych zawierający te punkty, i wypróbowanie tego szadera w aplikacji 1E (rozdz. 13) lub w innych, własnych aplikacjach.

Listing G.7. Procedura znajdowania minimum i maksimum

C

```
static GLuint program_id, uloc[3];

void GPUFindMinMax ( GLuint n, GLuint n0, GLuint databuf )
{
  glUseProgram ( program_id );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, databuf );
  glUniform1ui ( uloc[0], n );
  glUniform1ui ( uloc[1], n0 );
  glUniform1ui ( uloc[2], GL_TRUE );   /* uniform s = true; */
  COMPUTE ( n/2, 1, 1 )
  glUniform1ui ( uloc[2], GL_FALSE );  /* uniform s = false; */
  for ( n &= ~0x01; n > 2; n = 2*((n+3)/4) ) {
    glUniform1ui ( uloc[0], n );
    COMPUTE ( n/4, 1, 1 )
  }
  ExitIfGLError ( "GPUFindMinMax" );
} /*GPUFindMinMax*/
```

G.2. Obliczanie sum prefiksowych

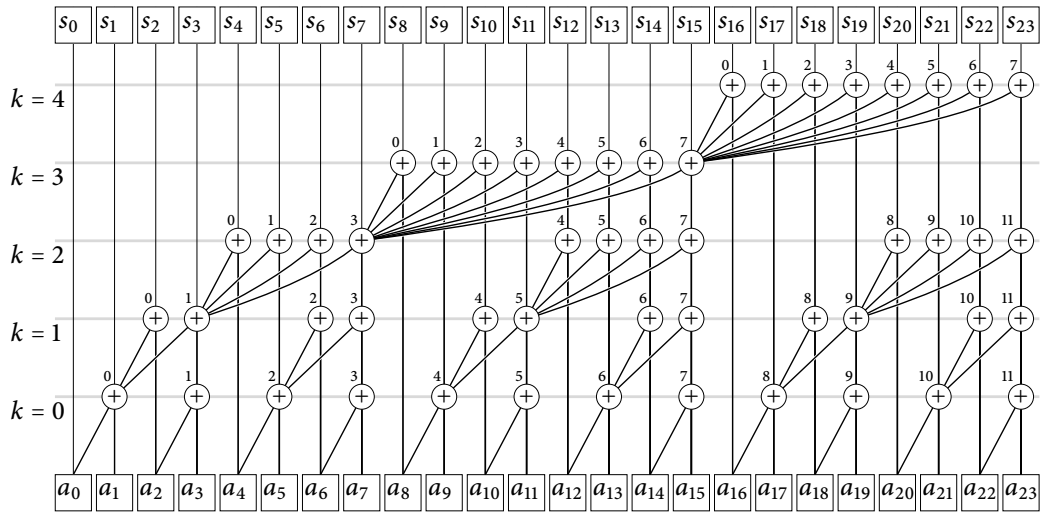
W wielu zastosowaniach zachodzi konieczność znalezienia **sum prefiksowych** danego ciągu liczb $a_0, \ldots, a_{n-1}$, czyli sum częściowych tego ciągu od początku do każdego miejsca:

$$s_l \stackrel{\text{def}}{=} \sum_{j=0}^{l} a_j, \quad l = 0, \ldots, n-1.$$

Łatwo jest to zrobić sekwencyjnie (wykonując kolejno $n-1$ dodawań), ale uruchamiając wątki działające równolegle, można to zadanie wykonać w $\lceil \log_2 n \rceil$ krokach, tak jak zsumowanie wszystkich liczb (czyli obliczenie tylko ostatniego elementu ciągu sum prefiksowych, s_{n-1}).

Zobaczmy implementację (w postaci procedury w C i szadera obliczeniowego) algorytmu obliczania sum prefiksowych. Wątki szadera są zorganizowane w jednowymiarową grupę roboczą. Algorytm równoległy obliczania sum prefiksowych składa się z $s = \lceil \log_2 n \rceil$ kroków (ponumerowanych od 0 do $s-1$, rys. G.2). W każdym kroku wątek wykonuje jedno dodawanie elementów danego ciągu lub ich sum.

Ciąg $a_0, \ldots, a_{n-1}$ dany w tablicy zostanie zastąpiony przez ciąg $s_0, \ldots, s_{n-1}$. W kolejnych krokach pewne elementy tablicy są dodawane do innych elementów, które zostają zastąpione przez obliczone sumy. Na rysunku można zauważyć, że $k+1$ najmniej znaczących cyfr w rozwinięciu dwójkowym indeksów elementów dodawanych w k-tym kroku do elementów na innych pozycjach to 0 i k jedynek. Każdy taki element jest dodawany do 2^k kolejnych elementów. W szczególności dla $k = 0$ każdy element o indeksie parzystym zostaje dodany do swojego prawego sąsiada (i tylko do niego).



Rysunek G.2. Nałożone drzewa binarne sum prefiksowych

Wątek o numerze i ma w kroku k-tym dodać elementy o indeksach

$$i_a = 2^{k+1}\lfloor i/2^k \rfloor + 2^k - 1, \quad i_b = i_a + (i \bmod 2^k) + 1$$

i wpisać sumę na miejsce i_b. Indeksy i_a, i_b dla kolejnych kroków można obliczać za pomocą działań na cyfrach rozwinięcia dwójkowego liczby i i maskach bitowych. Gdyby dopuszczalna wielkość lokalnej grupy roboczej była większa lub równa $n/2$, to opisany wyżej algorytm obliczania sum prefiksowych mogłaby realizować jedna taka grupa, w której i-ty wątek wykonywałby następującą instrukcję:

```
for ( ii = i+i, m0 = 0x01, m1 = 0;  m0 < n;  m1 = (m0 += m0)-1 ) {
  ia = (ii & ~m0) | m1;
  if ( (ib = ia + (i & m1) + 1) < N )
    a[ib] += a[ia];
  groupMemoryBarrier ();
}
```

Rolę zmiennej określającej numer kroku k pełni tu zmienna `m0`, która w k-tym kroku ma wartość 2^k. Warunek zakończenia pętli opisuje nierówność $2^k \geqslant n$ równoważną $k \geqslant \log_2 n$. Zmienna `m1` ma w k-tym kroku wartość $2^k - 1$, zatem k jej najmniej znaczących bitów ma wartość 1, a pozostałe to zera. W wyrażeniu, którego wartość jest przypisywana zmiennej `ia`, bit rozwinięcia dwójkowego liczby $2i$ na pozycji k jest kasowany, a wszystkie bity na pozycjach $0, \ldots, k-1$ otrzymują wartość 1, co daje wynik równoważny zastosowaniu podanego wcześniej wzoru na i_a. Wartość wyrażenia `(i & m1)` jest resztą z dzielenia i przez 2^k.

Procedura `groupMemoryBarrier` ma zapewnić dokończenie działania w k-tym kroku przez wszystkie wątki (całą grupę roboczą) przed rozpoczęciem wykonywania kroku następnego. Jednak działanie tej procedury jest ograniczone do *lokalnej* grupy roboczej, której

maksymalny rozmiar ogranicza dopuszczalną długość ciągu możliwego do przetworzenia przy użyciu powyższej instrukcji[4]. Dlatego obliczenie sum prefiksowych dla ciągów ponad dwa razy dłuższych niż dopuszczalna wielkość lokalnej grupy roboczej wymaga użycia rozwiązania, w którym za synchronizację dostępu do pamięci odpowiada CPU; dla lokalnych grup roboczych przyjmiemy rozmiary $1\times1\times1$, a długość (jednowymiarowej) globalnej grupy roboczej dobierzemy do długości ciągu.

Listing G.8 przedstawia procedurę, którą procedura `main` szadera ma wywołać, aby wykonać jeden krok algorytmu sumowania. Jej parametrem jest liczba i będąca wartością zmiennej wbudowanej `gl_GlobalInvocationID` — określa ona numer wątku w globalnej grupie roboczej. Zmienna jednolita `prStep` przechowuje numer k bieżącego kroku algorytmu. Zmienne `prN0` i `prN` określają miejsce początku ciągu i jego długość w tablicy `seq.a` znajdującej się w buforze magazynowym przywiązanym do punktu `0` w celu `GL_SHADER_STORAGE_BUFFER`. Układ `std430`[5] określa, że elementy tablicy (liczby całkowite 32-bitowe) są upakowane bez przerw.

Listing G.8. Procedura realizująca krok obliczania sum prefiksowych

GLSL

```
1: layout(std430,binding=0) buffer prSequence { int a[]; } seq;
2:
3: void iPrefixSum ( uint i )
4: {
5:   uint ii, m0, m1, ia, ib;
6:
7:   ii = i+i;  m0 = 0x01 << prStep;  m1 = m0-1;
8:   ia = (ii & ~m0) | m1;
9:   if ( (ib = ia + (i & m1) + 1) < prN )
10:    seq.a[prN0 + ib] += seq.a[prN0 + ia];
11: } /*iPrefixSum*/
```

Przedstawiona na listingu G.9 procedura `iPrefixSum`, działająca na CPU, realizuje zewnętrzną pętlę algorytmu sumowania, wykonywaną $s = \lceil \log_2 n \rceil$ razy. Parametry `N0` i `N` procedury określają miejsce początku i długość ciągu w buforze magazynowym przywiązanym do punktu dowiązania `0` w celu `GL_SHADER_STORAGE_BUFFER`. Ich wartości są przypisywane zmiennym jednolitym `prN0` i `prN`, których położenia zostały odczytane po złączeniu programu i zapamiętane w zmiennych `locN0` i `locN`. Zmienna jednolita `prStep` otrzymuje w kolejnych przebiegach pętli wartości $0, 1, \ldots, s-1$. W linii 7 jest obliczana potrzebna liczba wątków, równa $\lfloor n/2 \rfloor$, czyli długość globalnej grupy roboczej.

Makrodefinicja `COMPUTE` po wywołaniu procedury `glDispatchCompute` wywołuje procedurę `glMemoryBarrier`. Jej zadaniem jest wstrzymać wykonywanie programu na CPU aż do zakończenia obliczeń i zapisania w pamięci wyników przez wszystkie wątki szadera, aby można było przystąpić do następnego kroku sumowania lub do dalszych obliczeń korzystających z gotowych sum prefiksowych.

[4] Taki wariant algorytmu jest realizowany przez szader z listingu 29.30 w liniach 71–81.

[5] wprowadzony w specyfikacji OpenGL 4.3, dopuszczalny tylko dla buforów magazynowych

Listing G.9. Podprogram na CPU wywołujący procedurę z listingu G.8

C

```
void iPrefixSum ( GLuint *uvofs, GLuint N0, GLuint N )
{
  unsigned int k, m, d;

  glUniform1ui ( locN0, N0 );       /* uniform prN0 = N0; */
  glUinform1ui ( locN, N );         /* uniform prN = N; */
  d = N/2;
  for ( k = 0, m = N-1;  m > 0;  k++, m >>= 1 ) {
    glUniform1ui ( locprStep, k );  /* uniform prStep = k; */
    COMPUTE ( d, 1, 1 )
  }
  ExitIfGLError ( "iPrefixSum" );
} /*iPrefixSum*/
```

G.3. Sortowanie

Ale zemsta, choć leniwa,
Nagnała cię w nasze sieci;

Adam Mickiewicz: *Pani Twardowska*

Zaprogramujemy tzw. **sieć sortującą**, której masywnie równoległa implementacja sortuje ciąg o długości n w czasie rzędu $\log^2 n$. Implementacja sieci sortującej na GPU okazuje się znacznie prostsza niż dokładny opis i dowód poprawności tego algorytmu; opiszę *jak* on działa, ale po wyjaśnienie, *dlaczego* on działa (czyli po podstawy teoretyczne), jestem zmuszony odesłać Czytelników do rozdziału 27 książki [54].

Wykonawcą algorytmu na najniższym poziomie jest komparator, czyli procedura porównująca dane obiekty na dwóch wskazanych miejscach w tablicy i przestawiająca je, jeśli drugi obiekt powinien poprzedzać pierwszy. Rodzaj obiektów i relacja, zgodnie z którą należy je uporządkować, są „zaszyte" w komparatorze. Dla ustalenia uwagi w opisie algorytmu przyjmiemy, że w tablicy jest ciąg liczb, który ma być posortowany niemalejąco. W każdym kroku algorytmu $\lfloor n/2 \rfloor$ wątków komparatora może jednocześnie zbadać i tam, gdzie trzeba, przestawić $\lfloor n/2 \rfloor$ par obiektów w tablicy. Jeśli $n > 2$, to sztuka polega na wykonaniu właściwej (jak najmniejszej) liczby kroków i wskazaniu, w każdym kroku, odpowiednich par dla poszczególnych komparatorów.

Listing G.10 przedstawia implementację sieci sortującej na CPU — napisałem ją po to, aby ułatwić sobie uruchomienie implementacji docelowej i zrobić rysunek G.3, a zamieściłem ją tu, aby lepiej objaśnić działanie algorytmu. Dlatego ta implementacja może posortować tylko ciąg liczb całkowitych. Procedura `CompSwap` w liniach 1–9 jest dostosowanym do tego komparatorem. Sortowanie składa się z $s = \lceil \log_2 n \rceil$ etapów. W kolejnych etapach sortowanie odbywa się we fragmentach tablicy, których długościami są liczby $2, 4, 8, \ldots$, tj. kolejne całkowite potęgi dwójki[6]. Na początku etapu połowy każdego takiego fragmentu są posortowane; zadaniem etapu jest „scalenie" tych połówek, czyli takie przestawienie obiek-

[6] Jeśli liczba n nie jest potęgą dwójki, to ostatni fragment tablicy może być krótszy.

tów, aby uporządkować fragment. W i-tym etapie scalenie fragmentów o długości 2^{i-1}, dające posortowane fragmenty o długości 2^i, wymaga wykonania i kroków algorytmu.

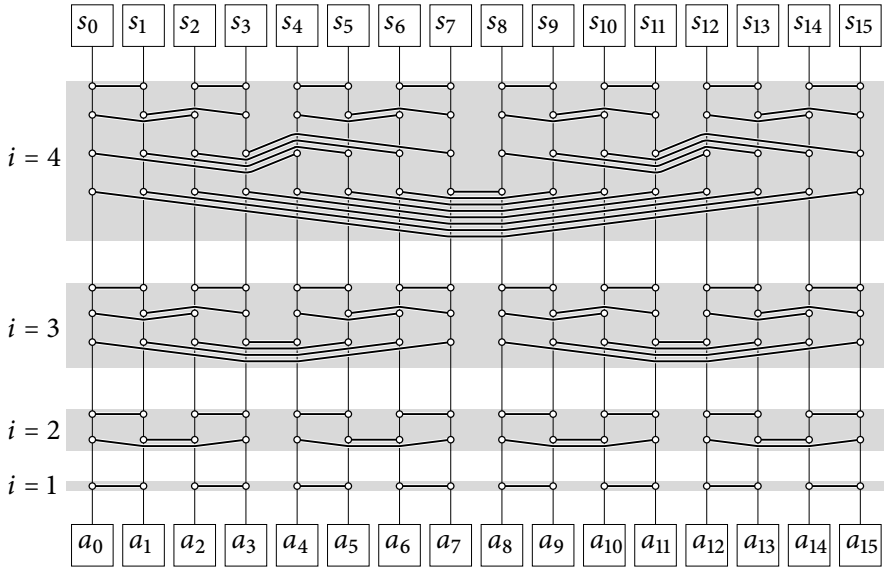
Listing G.10. Implementacja sekwencyjna sieci sortującej

C

```
1: void CompSwap ( unsigned int n, int *data, unsigned int i, unsigned int j )
2: {
3:   int x;
4:
5:   if ( j < n ) {
6:     if ( data[i] > data[j] )
7:       { x = data[i];  data[i] = data[j];  data[j] = x; }
8:   }
9: } /*CompSwap*/
10:
11: void NetSort ( unsigned int n, int *data )
12: {
13:   unsigned int steps, nn, h, h2, h4, i, j, k, kk, l;
14:
15:   if ( n < 2 )
16:     return;
17:   for ( nn = n-1, steps = 0;  nn;  nn >>= 1, steps ++ )
18:     ;
19:   nn = 1 << steps;
20:   for ( i = 0, h2 = 1, h = 2, k = nn >> 1;
21:         i < steps;
22:         i++, h2 = h, h <<= 1, k >>= 1 ) {
23:     for ( j = 0; j < k; j++ ) {
24:       for ( l = 0; l < h2; l++ )
25:         CompSwap ( n, data, j*h+l, j*h+h-1-l );
26:     }
27:     for ( h2 = h >> 1, h4 = h2 >> 1, kk = k << 1;
28:           h2 > 1;
29:           h2 = h4, kk <<= 1, h4 >>= 1 ) {
30:       for ( j = 0; j < kk; j++ ) {
31:         for ( l = 0; l < h4; l++ )
32:           CompSwap ( n, data, j*h2+l, j*h2+l+h4 );
33:       }
34:     }
35:   }
36: } /*NetSort*/
```

Instrukcje procedury `NetSort` w liniach 17–19 wyznaczają i zapamiętują w zmiennej `steps` liczbę s etapów sortowania, a zmienna `nn` otrzymuje wartość 2^s, tzn. najmniejszą całkowitą potęgę liczby 2 nie mniejszą niż n.

Kolejne etapy wykonywane są w pętli w liniach 20–35. Pierwszy krok każdego z tych etapów różni się od pozostałych, dlatego jest on wykonywany przez osobną pętlę w liniach 23–26. Ale choć CPU wywoła komparator dla każdej pary po kolei, zarówno w tym, jak i w każdym



Rysunek G.3. Sieć sortująca dla $n = 16$ — poziome odcinki i zygzaki symbolizują komparatory

następnym kroku (linie 27–34) wszystkie działania wykonywane przez komparator (komparatory) *mogą* się odbywać jednocześnie.

Przed i-tym etapem obie połowy fragmentu o długości 2^i tablicy są posortowane. W pierwszym kroku etapu komparatory porównują pierwszy obiekt z pierwszej połowy z ostatnim obiektem drugiej, drugi z przedostatnim itd. (rys. G.3). Okazuje się, że wskutek dokonanej wymiany obiektów między połowami powstają w tych połowach fragmentu tzw. **ciągi bitoniczne**, tj. dające się podzielić na dwie części monotoniczne — niemalejącą i nierosnącą — w tej lub w odwrotnej kolejności[7]. Co więcej, wszystkie obiekty w pierwszej połowie fragmentu są mniejsze lub równe obiektom w drugiej połowie. Kolejne kroki i-tego etapu mają na celu posortowanie tych połówek.

Sortowanie ciągu bitonicznego jest nazywane **czyszczeniem**. Zaczyna się ono w drugim kroku i-tego etapu, w którym ciąg bitoniczny w każdym kolejnym fragmencie tablicy o długości 2^{i-1} komparatory zamieniają na dwa dwukrotnie krótsze ciągi bitoniczne, przy czym żaden element pierwszego z nich nie jest większy niż którykolwiek element drugiego ciągu. Podobnie w kolejnych krokach czyszczenia długości ciągów bitonicznych maleją dwukrotnie, a elementy mniejsze zostają przestawione przed większe, aż powstaną ciągi o długości 1 — i w ten sposób zawartości kolejnych fragmentów tablicy o długości 2^i zostają posortowane.

Wartość nadawana zmiennej `k` przed wykonaniem i-tego etapu jest liczbą fragmentów o długości 2^i, przypisywanej zmiennej `h`. Wartości zmiennych `h2` i `h4` to odpowiednio długości ciągów bitonicznych przed i po kolejnym kroku czyszczenia.

Listing G.11 przedstawia szader obliczeniowy, którego wątki (instancje) wyznaczają zadania dla komparatorów. Właściwy komparator jest procedurą, którą najlepiej jest umieścić

[7] Ciąg monotoniczny też jest ciągiem bitonicznym, którego jedna z części monotonicznych jest pusta.

Listing G.11. Procedura `main` szadera sieci sortującej

GLSL

```
1: #version 450 core
2:
3: layout(local_size_x=1) in;
4:
5: uniform uint n, h;
6: uniform bool reverse;
7:
8: void CompSwap ( uint i, uint j );
9:
10: void main ( void )
11: {
12:   uint iid, i, j, l, h2;
13:
14:   iid = uint ( gl_GlobalInvocationID.x );
15:   h2 = h >> 1;  l = iid % h2;  iid /= h2;  i = iid*h+l;
16:   if ( (j = reverse ? (iid+1)*h-l-1 : i+h2) < n )
17:     CompSwap ( i, j );
18: } /*main*/
```

w innym szaderze. Dzięki temu tu nie ma żadnych zależności od rodzaju sortowanych obiektów ani żadnej konkretnej relacji ustalającej porządek w ich zbiorze, co w zasadzie umożliwia użycie tej samej procedury sortowania do obiektów różnych typów — wystarczy złączyć ten szader z szaderami zawierającymi różne komparatory, aby zbudować wszystkie programy do sortowania potrzebne w aplikacji.

Zadaniem procedury `main` jest obliczenie (na podstawie numeru instancji podanego w zmiennej `gl_GlobalInvocationID`) właściwej pary indeksów do sortowanej tablicy i wywołanie procedury komparatora. Zmienna jednolita `reverse` ma wartość `true`, jeśli szader jest wywołany w pierwszym kroku i-tego etapu; wtedy wartość zmiennej `h` jest równa 2^i, czyli jest długością fragmentu z dwoma podciągami niemalejącymi, które mają być scalone. Jeśli `reverse == false`, to wywołanie szadera nastąpiło w kroku czyszczenia, a wartość zmiennej `h` jest długością fragmentu tablicy zawierającego początkowy ciąg bitoniczny (to jest wartość zmiennej `h2` w procedurach na listingach G.10 i G.12).

W parze liczb podawanych jako parametry komparatora zawsze druga liczba jest większa; jeśli jest ona większa niż indeks ostatniego elementu ciągu, to komparator nie jest wywoływany, bo obiekt określony przez pierwszą liczbę (jeśli ona nie jest też za duża) ma zostać na swoim miejscu. Warunek badany w linii 16 zapewnia, że oba parametry podawane w wywołaniu komparatora są liczbami z zakresu $0, \ldots, n-1$. Przykładowy komparator, odpowiedni do sortowania liczb całkowitych, jest pokazany na listingu G.5.

Pliki o nazwach `sortnet.comp.glsl` i `sortnetcs.comp.glsl` zawierają procedurę `main` szadera sortowania i komparator; poza tym przykładowa procedura kompilacji szaderów sortowania na listingu G.12 nie wymaga objaśnień. Parametry procedury sortowania, `GPUNetSort`, opisują długość i początek ciągu oraz identyfikator bufora magazynowego zawierającego ten ciąg. W liniach 44–47 procedura przywiązuje ten bufor do celu, wybiera

Listing G.12. Procedury kompilacji szaderów i sortowania

C

```
static GLuint shader_id[2], program_id;
static GLuint uloc[4];    /* położenia zmiennych jednolitych */

void LoadSortingShaders ( void )
{
  static const char *filename[] =
    { "sortnet.comp.glsl", "sortnetcs.comp.glsl" };
  static const char *uname[] = { "n", "n0", "reverse", "h" };
  int i;

  shader_id[0] = CompileShaderFiles ( GL_COMPUTE_SHADER, 1, &filename[0] );
  shader_id[1] = CompileShaderFiles ( GL_COMPUTE_SHADER, 1, &filename[1] );
  program_id = LinkShaderProgram ( 2, shader_id, "sort" );
  for ( i = 0; i < 4; i++ )
    uloc[i] = glGetUniformLocation ( program_id, uname[i] );
  ExitIfGLError ( "LoadSortingShaders" );
} /*LoadSortingShaders*/

static void _GPUNetSort ( GLuint nseq, GLuint n, GLuint rloc, GLuint hloc )
{
  GLuint steps, nn, h, h2, gsize, i;

  if ( n < 2 )
    return;
  for ( nn = n-1, steps = 0;  nn;  nn >>= 1, steps ++ )
    ;
  nn = 1 << steps;  gsize = nn/2;
  for ( i = 0, h2 = 1, h = 2;  i < steps;  i++, h2 = h, h += h ) {
    glUniform1ui ( rloc, GL_TRUE );   /* uniform reverse = true; */
    glUniform1ui ( hloc, h );
    COMPUTE ( gsize, nseq, 1 )
    glUniform1ui ( rloc, GL_FALSE );  /* uniform reverse = false; */
    for ( ;  h2 > 1;  h2 >>= 1 ) {
      glUniform1ui ( hloc, h2 );   /* uniform h = h2; */
      COMPUTE ( gsize, nseq, 1 )
    }
  }
} /*_GPUNetSort*/

void GPUNetSort ( GLuint n, GLuint n0, GLuint dbuf )
{
  if ( n < 2 )
    return;
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, dbuf );
  glUseProgram ( program_id );
```

```
46:   glUniform1ui ( uloc[0], n );
47:   glUniform1ui ( uloc[1], n0 );
48:   _GPUNetSort ( 1, n, uloc[2], uloc[3] );
49:   ExitIfGLError ( "GPUNetSort" );
50: } /*GPUNetSort*/
```

program szaderów i nadaje wartości zmiennym jednolitym `n` i `n0`, po czym wywołuje procedurę `_GPUNetSort`, która realizuje właściwy algorytm; w zastosowaniach opisanych dalej użyjemy tej procedury, której parametry określają liczbę sortowanych ciągów (tu mamy jeden) oraz długość ciągu i odczytane z programu szaderów położenia zmiennych jednolitych `reverse` i `h`; w różnych zastosowaniach użyjemy różnych programów zawierających pełniące tę samą rolę zmienne jednolite o tych nazwach.

Zamiast pętli w liniach 23–26 i 30–33 na listingu G.10, w liniach 31 i 35 są wywołania programu szaderów (przez makrodefinicję `COMPUTE`). Liczba potrzebnych wątków (podczas sortowania jednego ciągu) jest równa `nn/2` — jest to największa całkowita potęga liczby 2 mniejsza niż n. Wywołana następnie procedura `glMemoryBarrier` czeka, aż wszystkie komparatory zakończą pracę.

G.4. Przetwarzanie macierzy rzadkich

Macierz rzadka $m \times n$ jest to (duża) macierz, która ma większość współczynników równych 0. Gdy liczba mn jest znacznie większa niż liczba współczynników niezerowych, warto[8] używać reprezentacji macierzy zajmujących jak najmniej miejsca w pamięci i wykonywać działania na nich bez marnowania czasu na mnożenie przez 0 i dodawanie zer.

Z pewnymi macierzami rzadkimi można sobie poradzić dosyć łatwo. Na przykład w podrozdziale B.3 jest użyta najbardziej naturalna reprezentacja macierzy trójdiagonalnych. Trochę większym wyzwaniem są **macierze o nieregularnej strukturze**, których niezerowe współczynniki są rozmieszczone w zupełnie dowolnych miejscach. W grafice komputerowej takie macierze pojawiają się m.in. w obliczeniach globalnego oświetlenia (rozdz. 29) i w przetwarzaniu siatek (rozdz. 31). Tu przyjrzymy się reprezentacji takich macierzy zwanej CSR (*compressed sparse rows*).

Dane opisujące macierz rzadką są umieszczone w trzech tablicach, które nazwę `r`, `c` i `a`. W tablicy `r`, o długości $m+1$, jest niemalejący ciąg liczb całkowitych $r_0, \ldots, r_m$, przy czym $r_0 = 0$, a różnice $r_{i+1} - r_i$ są liczbami niezerowych współczynników w kolejnych wierszach[9]. W szczególności r_m jest liczbą niezerowych współczynników w całej macierzy.

Pozostałe dwie tablice mają długość $N = r_m$. Liczby całkowite w tablicy `c` są numerami kolumn, w których występują niezerowe współczynniki. Tak więc w i-tym wierszu macierzy jest $r_{i+1} - r_i$ współczynników znajdujących się w kolumnach, których numery są podane w tablicy `c` na miejscach $r_i, r_i+1, \ldots, r_{i+1}-1$. Same współczynniki są przechowywane w tablicy `a` na miejscach o tych samych indeksach.

[8] a czasami po prostu trzeba

[9] Wiersze i kolumny macierzy, a także współrzędne wektorów, numerujemy, zaczynając od 0.

Przykład. Macierz

$$A = \begin{bmatrix} 1 & 2 & 0 & 0 & 3 & 0 & 4 & 0 \\ 0 & 5 & 6 & 7 & 0 & 0 & 0 & 8 \\ 9 & 0 & 0 & 10 & 0 & 0 & 0 & 0 \end{bmatrix}$$

ma wymiary 3×8 i tylko 10 niezerowych współczynników. Reprezentują ją tablice

```
r = { 0, 4, 8, 10};
c = {  0,  1,  4,  6,  1,  2,  3,  7,  0,  3};
a = {1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0};
```

Listing G.13 przedstawia definicję struktury, w której są zebrane niezbędne informacje o macierzy rzadkiej. Pola `m`, `n` i `nnz` służą do przechowania liczb wierszy, kolumn i niezerowych współczynników macierzy. Pole `lmax` służy do zapamiętania maksymalnej liczby niezerowych współczyników w wierszu. W tablicy `buf` są identyfikatory dwóch buforów magazynowych. W pierwszym z nich są umieszczone opisane wyżej tablice `r` i `c`, a w drugim tablica `a` z niezerowymi współczynnikami.

Listing G.13. Opakowanie reprezentacji macierzy rzadkiej

C

```
typedef struct {
    int    m, n, nnz, lmax;
    GLuint buf[2];
  } GPUSparseMatrix;
```

G.4.1. Mnożenie macierzy rzadkiej przez wektor

Przypuśćmy, że trzeba obliczyć wektor $\boldsymbol{y} \in \mathbb{R}^m$, który jest iloczynem macierzy rzadkiej A i wektora $\boldsymbol{x} = (x_0, \ldots, x_{n-1}) \in \mathbb{R}^n$. Jak wiemy, i-ta współrzędna wektora $\boldsymbol{y}$ jest równa

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j, \tag{G.1}$$

przy czym dla macierzy rzadkiej zwykle tylko niewiele składników sumy w tym wzorze nie jest zerem. Opisane niżej algorytmy mnożenia, zrealizowane w postaci procedur w C i szaderów obliczeniowych, obliczają i sumują tylko te składniki, w których $a_{ij} \neq 0$.

W wielu zastosowaniach występuje mnożenie macierzy rzadkiej przez *macierz* o więcej niż jednej kolumnie i dlatego warto mieć procedurę realizującą takie działanie. W szczególności wiersze macierzy X o wymiarach $n \times d$ i macierzy $Y = AX$ mogą być interpretowane jako punkty w przestrzeni d-wymiarowej. Dotyczy to na przykład rozpatrywanych w podrozdziale 31.11 macierzy, za pomocą których możemy obliczać położenia wierzchołków zagęszczonej siatki na podstawie wierzchołków siatki danej.

Przedstawię dwa algorytmy mnożenia macierzy rzadkiej przez wektor. Pierwszy z nich jest znacznie prostszy, nie korzysta z pamięci dodatkowej (tylko z reprezentacji macierzy oraz tablic z macierzami X i Y), a przy tym często działa znacznie szybciej niż algorytm opisany dalej, o teoretycznie mniejszym rzędzie złożoności czasowej. Szader realizujący ten algorytm jest pokazany na listingu G.14; nie pokazuję procedury kompilującej ten szader i odczytującej położenia zmiennych jednolitych, uznając to za niepotrzebne.

Listing G.14. Szader pierwszego algorytmu mnożenia macierzy rzadkiej przez wektor

GLSL

```
1: #version 450 core
2:
3: layout(local_size_x=1) in;
4:
5: layout(std430,binding=0) buffer RowsCols { uint  rc[]; } rc;
6: layout(std430,binding=1) buffer Coeff    { float a[]; } a;
7: layout(std430,binding=2) buffer Xvec     { float x[]; } x;
8: layout(std430,binding=3) buffer Yvec     { float y[]; } y;
9:
10: uniform uint m, dim;
11:
12: #define r(i) rc.rc[i]
13: #define c(i) rc.rc[m+1+(i)]
14:
15: void main ( void )
16: {
17:   uint  xi, yi, j, k, l;
18:   float s;
19:
20:   xi = gl_GlobalInvocationID.x;
21:   yi = gl_GlobalInvocationID.y;
22:   k = xi*dim + yi;
23:   for ( j = r(xi), s = 0.0;  j < r(xi+1);  j++ ) {
24:     l = c(j);
25:     s += a.a[j] * x.x[l*dim + yi];
26:   }
27:   y.y[k] = s;
28: } /*main*/
```

Deklaracje w liniach 5–8 opisują punkty dowiązania buforów z tablicami `r` i `c`, `a`, z macierzą X i miejscem na macierz Y. Wartościami zmiennych jednolitych `m` i `dim` są liczba m wierszy macierzy A i liczba d kolumn macierzy X (i Y). Globalna grupa robocza ma wymiary $m \times d \times 1$; zadaniem każdego wątku jest obliczenie jednego współczynnika macierzy Y, w wierszu i kolumnie określonych przez indeksy wątku w grupie roboczej. Zastosowany jest tu sekwencyjny algorytm sumowania. Sposób zatrudnienia tego szadera przez procedurę z listingu G.15 nie wymaga objaśnień.

Listing G.15. Pierwsza procedura mnożenia macierzy rzadkiej przez wektor

C

```
1: static GLuint program_id;
2: static GLuint uloc[2];  /* położenia odczytane po skompilowaniu programu */
3:
4: void GPUSMultSparseMatrixVectorf ( GLuint ybuf,
5:                          GPUSparseMatrix *a, GLuint dim, GLuint xbuf )
6: {
7:   glUseProgram ( program_id );
8:   glUniform1ui ( uloc[0], a->m );  /* uniform m = a->m; */
9:   glUniform1ui ( uloc[1], dim );   /* uniform dim = dim; */
10:  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, a->buf[0] );
11:  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, a->buf[1] );
12:  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 2, xbuf );
13:  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 3, ybuf );
14:  COMPUTE ( a->m, dim, 1 )
15:  ExitIfGLError ( "GPUSMultSparseMatrixVectorf" );
16: } /*GPUSMultSparseMatrixVectorf*/
```

Drugi algorytm powstał z chęci użycia algorytmu sumowania parami, który jak wiemy można zrównoleglić i który daje lepsze oszacowania skutków błędów zaokrągleń (zobacz podrozdz. G.1). Obie części implementacji są przedstawione na listingach G.16 i G.17.

Procedura `GPUMultSparseMatrixVectorf` ma takie same parametry jak procedura `GPUSMultSparseMatrixVectorf`: identyfikator bufora, w którym ma się znaleźć wynik, strukturę z opisem macierzy, liczbę d kolumn macierzy X i Y i identyfikator bufora z macierzą X.

Pole `lmax` służy do zapamiętania maksymalnej liczby niezerowych współczynników w wierszu macierzy A; jest to maksymalna liczba składników we wzorze (G.1), na jej podstawie ustalana jest liczba kroków sumowania parami. Jeśli pole to ma wartość 0, to procedura znajduje i zapisuje w nim maksymalną liczbę niezerowych współczynników, aby pominąć ten krok obliczeń w następnym wywołaniu[10].

W liniach 12–26 następują przygotowania, w tym nadawanie wartości zmiennym jednolitym i przywiązywanie buforów o podanych identyfikatorach do odpowiednich punktów dowiązania. Dwa bufory pomocnicze są tworzone w linii 20, po czym w liniach 21–26 są przywiązywane do odpowiednich punktów i są im nadawane odpowiednie wielkości.

Obliczenie składa się z kilku etapów, których numery są kolejno przypisywane zmiennej jednolitej `stage`. Globalna grupa robocza składa się z grup lokalnych o wymiarach $1 \times 1 \times 1$ i w poszczególnych etapach jest jedno- lub dwuwymiarowa. Makrodefinicja `EXECSTAGE` w liniach 4–5, wprowadzona dla skrócenia i uczytelnienia kodu, wywołuje (za pośrednictwem makrodefinicji `COMPUTE` z listingu 9.1) program szaderów w celu wykonania kolejnego kroku

[10] Mnożenie macierzy przez wektor jest krokiem wielu iteracyjnych metod rozwiązywania układów równań liniowych, w których dana macierz jest mnożona kolejno przez różne wektory. Jeśli macierz się nie zmienia, to i liczbę $l_{\max}$ wystarczy znaleźć tylko raz.

Listing G.16. Druga procedura mnożenia macierzy rzadkiej przez wektor

C

```
static GLuint program_id;
static GLuint uloc[5];

#define EXECSTAGE(STAGE,SIZEX,SIZEY,SIZEZ) \
  { glUniform1i ( uloc[0], STAGE );  COMPUTE ( SIZEX, SIZEY, SIZEZ ) }

void GPUMultSparseMatrixVectorf ( GLuint ybuf,
                        GPUSparseMatrix *a, GLuint dim, GLuint xbuf )
{
  GLuint auxb[2], t;

  glUseProgram ( program_id );
  glUniform1ui ( uloc[1], a->m );
  glUniform1ui ( uloc[2], a->nnz );
  glUniform1ui ( uloc[3], dim );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, a->buf[0] );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, a->buf[1] );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 2, xbuf );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 3, ybuf );
  glGenBuffers ( 2, auxb );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 5, auxb[1] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER, a->nnz*dim*sizeof(GLfloat),
                 NULL, GL_DYNAMIC_DRAW );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 4, auxb[0] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER, a->m*sizeof(GLuint),
                 NULL, GL_DYNAMIC_DRAW );
  EXECSTAGE ( 0, a->m, 1, 1 )
  if ( !a->lmax ) {
    glUniform1i ( uloc[0], 1 );  /* stage = 1 */
    for ( t = a->m; t > 1; t = (t+1)/2 ) {
      glUniform1ui ( uloc[4], t );
      COMPUTE ( t/2, 1, 1 )
    }
    glGetBufferSubData ( GL_SHADER_STORAGE_BUFFER, 0,
                         sizeof(GLuint), &a->lmax );
    EXECSTAGE ( 0, a->m, 1, 1 )
  }
  EXECSTAGE ( 2, a->nnz, 1, 1 )
  glUniform1i ( uloc[0], 3 );  /* stage = 3 */
  for ( t = a->lmax; t > 1; t = (t+1)/2 ) {
    glUniform1ui ( uloc[4], t );
    COMPUTE ( a->m, (t/2), 1 )
  }
  EXECSTAGE ( 4, a->m, 1, 1 )
  glUseProgram ( 0 );
```

```
46:   glDeleteBuffers ( 2, auxb );
47:   ExitIfGLError ( "GPUMultSparseMatrixVectorf" );
48: } /*GPUMultSparseMatrixVectorf*/
```

lub etapu obliczeń i czeka na jego wyniki. Pierwszy etap (w którym zmienna `stage` ma wartość `0`) polega na obliczeniu różnic $r_{i+1} - r_i$ i zapamiętaniu ich w tablicy `lgt.l` (w pierwszym buforze pomocniczym).

Jeśli pole `a->lmax` ma wartość `0`, to następują (w liniach 28–37) dwa dodatkowe etapy mające znaleźć maksymalną różnicę. Pierwszy z nich realizuje algorytm opisany w podrozdziale G.1 i składający się z $\lceil \log_2 m \rceil$ kroków. Największa różnica jest odczytywana z bufora w liniach 34–35. Ponieważ jej znalezienie wiąże się z zepsuciem zawartości bufora, w linii 36 różnice są obliczane ponownie.

Uwaga: Największa liczba niezerowych współczynników w wierszu jest odczytywana z bufora `auxb[0]` (tj. z bloku magazynowego `RowL`) — to dlatego ten bufor został przywiązany do celu `GL_SHADER_STORAGE_BUFFER` jako ostatni (w linii 24).

Etap realizowany w linii 38 wykonuje mnożenie współczynników macierzy przez odpowiednie wiersze macierzy X; iloczyny są zapamiętywane w drugim buforze pomocniczym (w tablicy `b.b`)[11]. Zauważmy (listing G.17, linie 33–34), że współczynnik a_{ij} macierzy, pamiętany w miejscu `i` tablicy `a.a`, musi być pomnożony przez j-ty wiersz macierzy X, którgo numer j jest brany z tablicy `c`.

Kolejny etap, realizowany w liniach 39–43 procedury w C i w liniach 37–42 szadera, ma obliczyć sumy składników poszczególnych współczynników macierzy Y. Wszystkie wiersze tej macierzy są obliczane jednocześnie za pomocą opisanego w podrozdziale G.1 sumowania parami. W tym etapie grupa robocza jest dwuwymiarowa; indeks x wątku w grupie jest numerem sumy (czyli wiersza macierzy Y), a indeks y określa pierwszy składnik w danym kroku sumowania.

Drugi składnik w danym kroku ma numer $k = j + \lceil t/2 \rceil$, obliczany w linii 38, przy czym numer ten musi być mniejszy niż liczba składników i-tej sumy i mniejszy niż liczba t (wartość zmiennej `t`) określająca maksymalną liczbę składników sumowanych w danym kroku algorytmu sumowania parami. Ten warunek jest sprawdzany w linii 39; zauważmy, że jeśli liczby składników poszczególnych sum są różne, to pewne wątki „próżnują”, ale w kolejnych krokach (których jest $\lceil \log_2 l_{\max} \rceil$) drugi wymiar grupy roboczej maleje do jedynki i próżnujących wątków jest coraz mniej.

Ostatni etap (linia 44 na listingu G.16 i linie 45–46 na listingu G.17) ma skopiować obliczone sumy z bufora pomocniczego do bufora, w którym ma się znaleźć wynik. Miejsce w buforze pomocniczym, w którym poprzedni etap obliczeń zostawił i-ty wiersz macierzy Y (którego indeks i jest wartością zmiennej `i`), ma numer `r(i)`, chyba że następna liczba w tablicy `r` jest taka sama (czyli różnica $r_{i+1} - r_i$ pamiętana w `lgt.l[i]` jest zerem). Jest tak wtedy, gdy w i-tym wierszu macierzy A są tylko zerowe współczynniki, a wtedy i-ty wiersz macierzy Y też składa się z samych zer.

[11] Mnożenie współczynników wiersza macierzy X jest tu wykonywane sekwencyjnie. Można by to zrównoleglić, ale dla zadań „dużych” (w porównaniu z liczbą procesorów w GPU) to nie musi przyspieszyć obliczeń.

Listing G.17. Szader drugiego algorytmu mnożenia macierzy rzadkiej przez wektor

GLSL

```
#version 450

layout(local_size_x=1) in;

layout(std430,binding=0) buffer RowsCols { uint  rc[]; } rc;
layout(std430,binding=1) buffer Coeff    { float a[]; } a;
layout(std430,binding=2) buffer Xvec     { float x[]; } x;
layout(std430,binding=3) buffer Yvec     { float y[]; } y;
layout(std430,binding=4) buffer RowL     { uint  l[]; } lgt;
layout(std430,binding=5) buffer Prod     { float b[]; } b;

uniform int  stage;
uniform uint m, nnz, dim, t;

#define r(i) rc.rc[i]
#define c(i) rc.rc[m+1+(i)]

void main ( void )
{
  uint i, j, k, l, u, v;

  i = gl_GlobalInvocationID.x;
  switch ( stage ) {
case 0:
    lgt.l[i] = r(i+1) - r(i);
    return;
case 1:
    if ( (j = i + (t+1)/2) < t )
      if ( lgt.l[j] > lgt.l[i] )
        lgt.l[i] = lgt.l[j];
    return;
case 2:
    for ( l = 0, u = i*dim, v = c(i)*dim;  l < dim;  l++)
      b.b[u++] = a.a[i] * x.x[v++];
    return;
case 3:
    j = gl_GlobalInvocationID.y;
    k = j + (t+1)/2;
    if ( k < lgt.l[i] && k < t ) {
      for ( l = 0, u = (r(i)+j)*dim, v = (r(i)+k)*dim;  l < dim;  l++ )
        b.b[u++] += b.b[v++];
    }
    return;
case 4:
    for ( l = 0, u = i*dim, v = r(i)*dim;  l < dim;  l++ )
```

```
46:       y.y[u++] = b.b[v++];
47:     return;
48: default:
49:     return;
50:   }
51: } /*main*/
```

Drugi z przedstawionych wyżej algorytmów jest znacznie bardziej skomplikowany i potrzebuje sporo pamięci dodatkowej. Warto więc sprawdzić, czy jest lepszy. Choć obliczenia są w nim podzielone na więcej wątków, które mogą być wykonywane równolegle, warunkiem faktycznego przyspieszenia obliczeń byłoby istnienie odpowiednio dużej liczby procesorów GPU. W rzeczywistości wielokrotne uruchamianie kolejnych etapów obliczeń i czekanie na ich dokończenie przed następnymi etapami zabiera sporo czasu. Jeśli liczba wierszy macierzy jest zbliżona do liczby procesorów lub większa, a przy tym maksymalna liczba niezerowych współczynników w wierszu nie jest duża, to pierwszy algorytm „zatrudnia" procesory GPU w prawie jednakowym stopniu, a to oznacza, że wykorzystuje GPU bardzo efektywnie.

Algorytm sumowania parami daje zazwyczaj dokładniejsze wyniki niż algorytm sekwencyjny, ale dla zwiększenia dokładności tego ostatniego wystarczy zadeklarować zmienną `s` (listing G.14, linia 18) typu `double`[12], co zmniejszy błędy zaokrągleń sumowania.

Jeśli macierz rzadka $n \times n$ jest symetryczna, to może być przechowywana w mniejszej ilości pamięci — ponieważ $a_{ij} = a_{ji}$ dla każdego i, j, w zasadzie wystarczy trzymać w tablicach tylko niezerowe współczynniki na i pod diagonalą (dla $i \geqslant j$). Nie jest jednak łatwe dostosowanie do takiej oszczędnej reprezentacji równoległego algorytmu mnożenia macierzy przez wektor, dlatego rzadkie macierze symetryczne powinny być w pamięci GPU reprezentowane tak jak niesymetryczne.

G.4.2. Transponowanie macierzy rzadkiej

Dla macierzy rzadkiej A znajdziemy reprezentację jej transpozycji, A^T, a raczej napiszemy szader i procedurę w C, które to będą robić. Dla macierzy $m \times n$, która ma N niezerowych współczynników, zostanie to wykonane w $O(\log^2 N + \log m)$ krokach.

Listing G.18 przedstawia procedurę dokonującą transpozycji przy użyciu szadera z listingu G.19. Identyfikator programu szaderów i położenia zmiennych jednolitych w tym programie są pamiętane w zmiennych `program_id` i `uloc`.

Po znalezieniu macierzy A^T macierz A może być w aplikacji niepotrzebna. W takim przypadku tablica `a`, w której są przechowywane współczynniki macierzy A, może stać się częścią reprezentacji macierzy A^T — będą w niej te same liczby ustawione w innej kolejności. Jeśli potrzebne są obie macierze, to zostanie utworzony bufor z nową tablicą o tej samej długości. Zawsze będzie tworzony nowy bufor z tablicami `r` i `c` dla macierzy A^T, bo jego długość, $n + 1 + N$, jest na ogół inna niż suma $m + 1 + N$ długości tablic `r` i `c` dla macierzy A.

[12] Trzeba wtedy dopisać konwersję między typami `double` a `float`, aby kompilator uznał kod szadera za bezbłędny.

Listing G.18. Procedura znajdowania transpozycji macierzy rzadkiej

C

```
static GLuint program_id;
static GLuint uloc[7];

#define EXECSTAGE(STAGE,SIZEX,SIZEY,SIZEZ) .... /* listing G.16 */

char GPUTransposeSparsef ( GPUSparseMatrix *at, GPUSparseMatrix *a,
                           char keep_a )
{
  GLuint atb[3];
  GLuint m, n, nnz;

  glUseProgram ( program_id );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, a->buf[0] );
  glUniform1ui ( uloc[4], m = at->n = a->m );
  glUniform1ui ( uloc[5], n = at->m = a->n );
  glUniform1ui ( uloc[6], nnz = at->nnz = a->nnz );
  if ( keep_a ) {
    glGenBuffers ( 3, atb );
    glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, a->buf[1] );
    glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 3, at->buf[1] = atb[2] );
    glBufferData ( GL_SHADER_STORAGE_BUFFER, nnz*sizeof(GLfloat),
                   NULL, GL_DYNAMIC_DRAW );
    EXECSTAGE ( 0, nnz, 1, 1 );
  }
  else {
    glGenBuffers ( 2, atb );
    glBindBufferBase ( GL_SHADER_STORAGE_BLOCK, 3, at->buf[1] = a->buf[1] );
  }
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 2, at->buf[0] = atb[0] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER, (n+1+nnz)*sizeof(GLuint),
                 NULL, GL_DYNAMIC_DRAW );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 4, atb[1] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER, nnz*sizeof(GLuint),
                 NULL, GL_DYNAMIC_DRAW );
  EXECSTAGE ( 1, nnz, 1, 1 );
  _GPUNetSort ( 1, nnz, uloc[2], uloc[3] );  /* listing G.12 */
  EXECSTAGE ( 3, n+1, 1, 1 );
  glDeleteBuffers ( 1, &atb[1] );
  if ( !keep_a ) {
    glDeleteBuffers ( 1, &a->buf[0] );
    memset ( a, 0, sizeof(GPUSparseMatrix) );
  }
  return true;
} /*GPUTransposeSparsef*/
```

Listing G.19. Szader znajdowania transpozycji macierzy rzadkiej

GLSL

```
#version 430

layout(local_size_x=1) in;

layout(std430,binding=0) buffer RC     { uint  rc[]; } rc;
layout(std430,binding=1) buffer Coeff  { float a[];  } aa;
layout(std430,binding=2) buffer RCT    { uint  rc[]; } rct;
layout(std430,binding=3) buffer CoeffT { float a[];  } at;
layout(std430,binding=4) buffer auxb   { uint  a[];  } aux;

uniform uint m, n, nnz;
uniform uint stage, step, h;
uniform bool reverse;

#define r(I) rc.rc[I]
#define c(I) rc.rc[m+1+(I)]
#define rt(I) rct.rc[I]
#define ct(I) rct.rc[n+1+(I)]
#define pairi(I) ct(I)
#define pairj(I) aux.a[I]

void CompSwap ( uint i, uint j )
{
  uint  b;
  float x;

  if ( pairj(j) < pairj(i) ||
       pairj(j) == pairj(i) && pairi(j) < pairi(i) ) {
    b = pairi(i);  pairi(i) = pairi(j);  pairi(j) = b;
    b = pairj(i);  pairj(i) = pairj(j);  pairj(j) = b;
    x = at.a[i];   at.a[i] = at.a[j];    at.a[j] = x;
  }
} /*CompSwap*/

void main ( void )
{
  uint i, j, k, l, h2;

  i = gl_GlobalInvocationID.x;
  switch ( stage ) {
case 0:
    at.a[i] = aa.a[i];
    return;
case 1:
    pairj(i) = c(i);
```

```
46:     for ( j = 0, k = m;  k-j > 1; ) {
47:       l = j + (k-j)/2;
48:       i < r(l) ? (k = l) : (j = l);
49:     }
50:     pairi(i) = j;
51:     return;
52: case 2:
53:     h2 = h >> 1;  l = i % h2;  i /= h2;  j = i*h+l;
54:     if ( (k = reverse ? (i+1)*h-l-1 : j+h2) < nnz )
55:       CompSwap ( j, k );
56:     return;
57: case 3:
58:     if ( i > pairj(nnz-1) )
59:       rt(n) = nnz;
60:     else if ( i <= pairj(0) )
61:       rt(i) = 0;
62:     else {  /* i < n */
63:       for ( j = 1, k = nnz;  k-j > 1; ) {
64:         l = j + (k-j)/2;
65:         i < pairj(l) || i == pairj(l-1) ? (k = l) : (j = l);
66:       }
67:       rt(i) = j;
68:     }
69:     return;
70: default:
71:     return;
72:   }
73: } /*main*/
```

Pierwszy parametr procedury `GPUTransposeSparsef` to wskaźnik opakowania wyniku, tj. transpozycji macierzy, której opis jest wskazywany przez drugi parametr. Jeśli trzeci parametr ma wartość zerową (`false`), to reprezentacja macierzy danej zostanie zlikwidowana, a bufor, w którym są jej współczynniki, stanie się częścią macierzy wynikowej.

W liniach 12–16 procedura wybiera program szaderów, przywiązuje bufor z tablicami `r` i `c` do celu `GL_SHADER_STORAGE_BUFFER` i nadaje zmiennym jednolitym `m`, `n` i `nnz` wartości m, n i N.

Jeśli reprezentacja macierzy danej ma być zachowana, to w linii 18 są tworzone trzy bufory, z których dwa staną się częścią reprezentacji wyniku, a trzeci jest pamięcią dodatkową dla algorytmu. Etap 0 (listing G.19, linia 42) ma skopiować tablicę `a` ze współczynnikami macierzy danej do bufora z tablicą `a` znajdowanej transpozycji. Jeśli reprezentacja macierzy danej nie jest dalej potrzebna, to w linii 26 są tworzone tylko dwa bufory, w pierwszym z nich będą tablice `r` i `c` wyniku, a drugi jest pamięcią dodatkową.

W liniach 29–34 nowo utworzone bufory są przywiązywane do odpowiednich punktów dowiązania w celu `GL_SHADER_STORAGE_BUFFER` i są im nadawane potrzebne wielkości — bufor roboczy musi pomieścić N liczb całkowitych.

W etapie 1 powstają trójki (i, j, a_{ij}) opisujące niezerowe współczynniki macierzy A; każdy wątek szadera przetwarza jeden współczynnik. Indeks kolumny, j, który stanie się numerem wiersza macierzy A^T, jest wpisywany do bufora roboczego (listing G.19, linia 45). Indeks wiersza i stanie się indeksem kolumny; jest on wpisywany do docelowej tablicy `c` (listing G.19, linie 50, 18, 19), przy czym dla każdego współczynnika numer wiersza macierzy A, w którym on się znajduje, trzeba znaleźć metodą wyszukiwania binarnego — to jest wykonywane w pętli w liniach 46–49.

Etap 2 jest sortowaniem trójek w kolejności niemalejących indeksów j, trójki z tym samym indeksem j są sortowane w kolejności rosnących indeksów i. Do sortowania jest użyta procedura w C z listingu G.12. Instrukcje w linii 53 obliczają indeksy pary trójek, po czym jest wywoływana procedura `CompSwap`, czyli komparator, który porządkuje tę parę.

Ostatni etap, 3, ma wypełnić tablicę `r` reprezentacji macierzy A^T. W tablicy roboczej są, uporządkowane niemalejąco, numery wierszy kolejnych współczynników tej macierzy. Pierwszy element tablicy `r` jest równy 0, ostatni element (o indeksie n) jest równy N. Jeśli początkowe wiersze mają tylko zerowe współczynniki, to na początku tablicy będzie odpowiednio więcej zer, podobnie, jeśli macierz A^T ma zerowe końcowe wiersze, to odpowiadające im elementy tablicy `r` otrzymują wartość N — to robią instrukcje w liniach 61 i 59. Dla każdego z pozostałych wierszy metodą wyszukiwania binarnego jest znajdowany (w liniach 63–66) indeks w tablicy `a` pierwszego niezerowego współczynnika, albo, jeśli wiersz jest zerowy, indeks w tablicy `a` pierwszego niezerowego współczynnika w najbliższym wierszu niezerowym.

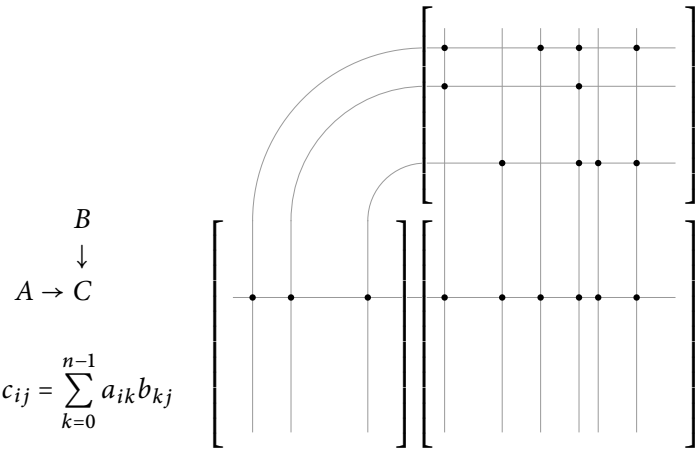
Bufor roboczy jest likwidowany przez instrukcję w linii 38 na listingu G.18. Jeśli reprezentacja macierzy A nie ma być zachowana, to w liniach 40–41 jest też sprzątany bufor z jej tablicami `r` i `c` i opakowanie macierzy A jest czyszczone.

G.4.3. Mnożenie macierzy rzadkich

Zrealizujemy na GPU mnożenie macierzy rzadkich reprezentowanych w sposób przedstawiony na początku tego podrozdziału. Procedura mnożenia korzysta z opisanych wcześniej w tym dodatku algorytmów sortowania i obliczania sum prefiksowych.

Spodziewając się, że macierz C, która jest iloczynem macierzy rzadkich A i B, też jest rzadka i chcąc ją reprezentować w taki sam sposób, trzeba wskazać miejsca, w których pojawią się jej niezerowe współczynniki. Aby to zrobić, zobaczmy schemat na rysunku G.4. Przedstawia on niezerowe współczynniki w pewnym (i-tym) wierszu macierzy A; i-ty wiersz iloczynu jest sumą wierszy macierzy B pomnożonych przez współczynniki z i-tego wiersza macierzy A. Macierz C może mieć niezerowe współczynniki w tych kolumnach, w których występują niezerowe współczynniki w wyróżnionych wierszach macierzy B. Zatem, aby obliczyć i-ty wiersz macierzy C, trzeba przez niezerowe współczynniki w i-tym wierszu macierzy A pomnożyć niezerowe współczynniki w odpowiednich wierszach macierzy B, a następnie obliczyć, dla każdego j, współczynnik c_{ij} jako sumę tych iloczynów, w których występują współczynniki macierzy B z j-tej kolumny[13].

[13] Taka suma iloczynów może być równa 0, ale tym się nie zajmiemy, chyba że Czytelnik, w ramach ćwiczenia, rozbuduje procedurę o „czyszczenie" otrzymanego wyniku z zer. Ma to większy sens dla macierzy o współczynnikach całkowitych niż zmiennopozycyjnych. Dalej, pisząc o niezerowych współczynnikach macierzy C, mam na myśli sumy niezerowych iloczynów.



Rysunek G.4. Schemat Falka dla iloczynu macierzy rzadkich

Listing G.20 przedstawia procedurę `GPUMultSparseMatricesf`, która oblicza iloczyn macierzy rzadkich $m \times n$ i $n \times l$, korzystając z szadera obliczeniowego zamieszczonego na listingu G.21. Po skompilowaniu i złączeniu programu szaderów odczytane z niego położenia zmiennych jednolitych `stage`, `prN0`, `prN`, `prStep`, `ma`, `nnza`, `mb`, `nprod`, `nnzc`, `h`, `reverse` i `tablgt` są (w tej kolejności) zapamiętane w tablicy `uvloc`.

Parametry procedury mnożenia to wskaźniki opakowań macierzy: wynikowej (tj. iloczynu $C = AB$) oraz czynników A i B.

Instrukcje w liniach 15–21 przywiązują bufory z macierzami A i B do odpowiednich punktów dowiązania i nadają wartości zmiennym jednolitym `ma` (liczba wierszy macierzy A i C), `nnza` (liczba niezerowych współczynników macierzy A) i `mb` (liczba wierszy macierzy B i jednocześnie liczba kolumn macierzy A). W linii 22 procedura rezerwuje pięć buforów; pierwsze trzy z nich będą użyte jako pamięć robocza, a w pozostałych dwóch będzie umieszczony wynik (i identyfikatory tych buforów zostaną przypisane wskaźnikom w tablicy `c->buf`).

Pierwszy etap algorytmu (linie 26–29) ma znaleźć liczbę wszystkich niezerowych iloczynów $a_{ik}b_{kj}$. Niezerowy współczynnik a_{ik} ma być pomnożony przez niezerowe współczynniki macierzy B w k-tym wierszu. Do bufora roboczego (o długości `nnza+1`), przywiązanego do punktu 4, wątki szadera wpisują liczby współczynników w odpowiednich wierszach macierzy B (listing G.21, linie 55–57). Liczby te są wpisywane „o jedno miejsce dalej", a na początek ciągu liczb w buforze trafi 0. W linii 27 następuje obliczenie ciągu sum prefiksowych. Ostatnia liczba w tym ciągu, oznaczę ją literą P, jest całkowitą liczbą iloczynów, która trafi do zmiennej `nprod` w procedurze `GPUMultSparseMatricesf` i do zmiennej jednolitej `nprod` szadera. Pary sąsiednich liczb w tym ciągu wyznaczają początki i końce obliczonych później podciągów iloczynów współczynników, otrzymanych dla kolejnych współczynników macierzy A.

Istnieją takie niezerowe macierze rzadkie, których iloczyn jest macierzą zerową. W szczególności może się okazać, że liczba P niezerowych iloczynów jest zerem, a wtedy procedura kończy działanie, wypełniając zerami opakowanie macierzy C (linie 31–33).

Listing G.20. Procedura mnożenia macierzy rzadkich

C

```
static GLuint program_id;
static GLuint uvloc[12];

#define EXECSTAGE(STAGE,SIZEX,SIZEY,SIZEZ)  .... /* listing G.16 */

static void iPrefixSum ( int N0, int N ) { .... } /* listing G.9 */

char GPUMultSparseMatricesf ( GPUSparseMatrix *c,
                              GPUSparseMatrix *a, GPUSparseMatrix *b )
{
  GLuint auxb[5];
  GLuint nprod, _nnzc, maxnt, tablgt, i;

  glUseProgram ( program_id );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, a->buf[0] );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, a->buf[1] );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 2, b->buf[0] );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 3, b->buf[1] );
  glUniform1ui ( uloc[4], a->m );
  glUniform1ui ( uloc[5], a->nnz );
  glUniform1ui ( uloc[6], a->n );
  glGenBuffers ( 5, auxb );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 4, auxb[0] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER,
                 (a->nnz+1)*sizeof(GLuint), NULL, GL_DYNAMIC_DRAW );
  EXECSTAGE ( 1, a->nnz, 1, 1 )
  iPrefixSum ( 1, a->nnz );
  glGetBufferSubData ( GL_SHADER_STORAGE_BUFFER,
                       a->nnz*sizeof(GLuint), sizeof(GLuint), &nprod );
  if ( !nprod ) {
    glDeleteBuffers ( 5, auxb );
    memset ( c, 0, sizeof(GPUSparseMatrix) );
    return false;
  }
  glUniform1ui ( uloc[7], nprod );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 6, auxb[1] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER,
                 2*nprod*sizeof(GLuint), NULL, GL_DYNAMIC_DRAW );
  glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 5, auxb[2] );
  glBufferData ( GL_SHADER_STORAGE_BUFFER,
                 nprod*sizeof(GLfloat), NULL, GL_DYNAMIC_DRAW );
  EXECSTAGE ( 2, nprod, 1, 1 )
  glUniform1ui ( uloc[11], tablgt = nprod > a->m ? nprod+1 : a->m+1 );
  glDeleteBuffers ( 1, &auxb[0] );
  glGenBuffers ( 1, &auxb[0] );
```

```
46:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 4, auxb[0] );
47:   glBufferData ( GL_SHADER_STORAGE_BUFFER,
48:                  2*tablgt*sizeof(GLuint), NULL, GL_DYNAMIC_DRAW );
49:   EXECSTAGE ( 3, a->m, 1, 1 )
50:   EXECSTAGE ( 4, a->m, 1, 1 );
51:   glUniform1i ( uloc[0], 5 );  /* uniform stage = 5; */
52:   for ( i = a->m; i > 1; i = (i+1)/2 ) {
53:     glUniform1ui ( uloc[2], i );
54:     COMPUTE ( i/2, 1, 1 );
55:   }
56:   glGetBufferSubData ( GL_SHADER_STORAGE_BUFFER,
57:                        tablgt*sizeof(GLuint), sizeof(GLuint), &maxnt );
58:   _GPUNetSort ( a->m, maxnt, uloc[10], uloc[9] );
59:   EXECSTAGE ( 7, nprod, 1, 1 )
60:   iPrefixSum ( 1, nprod );
61:   glGetBufferSubData ( GL_SHADER_STORAGE_BUFFER,
62:                        nprod*sizeof(GLuint), sizeof(GLuint), &_nnzc );
63:   glUniform1ui ( uloc[8], _nnzc );
64:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 1, auxb[4] );
65:   glBufferData ( GL_SHADER_STORAGE_BUFFER,
66:                  _nnzc*sizeof(GLfloat), NULL, GL_DYNAMIC_DRAW );
67:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, auxb[3] );
68:   glBufferData ( GL_SHADER_STORAGE_BUFFER,
69:                  (a->m+1+_nnzc)*sizeof(GLuint), NULL, GL_DYNAMIC_DRAW );
70:   EXECSTAGE ( 8, nprod+1, 1, 1 )
71:   EXECSTAGE ( 9, _nnzc, 1, 1 );
72:   EXECSTAGE ( 10, a->m, 1, 1 )
73:   c->m = a->m;  c->n = b->n;  c->nnz = _nnzc;  c->lmax = 0;
74:   c->buf[0] = auxb[3];  c->buf[1] = auxb[4];
75:   glUseProgram ( 0 );
76:   glDeleteBuffers ( 3, auxb );
77:   ExitIfGLError ( "GPUMultSparseMatricesf" );
78:   return true;
79: } /*GPUMultSparseMatricesf*/
```

Drugi etap mnożenia realizują instrukcje w liniach 35–42. Dla każdego iloczynu $a_{ik}b_{kj}$ trzeba zapamiętać w buforach roboczych trzy liczby: indeksy i, j oraz sam iloczyn. W liniach 36–41 rezerwowana jest pamięć na $2P$ liczb typu `GLuint` w jednym oraz P liczb typu `GLfloat` w drugim buforze. Bufory robocze są przywiązywane do punktów dowiązania 6 i 5; ich nazwy lokalne w treści szadera to `aux2` i `aux1`, ale dostęp do pierwszego z nich odbywa się za pomocą makrodefinicji `pairi` i `pairj` (linie 22, 23), bo w tym buforze będą przechowywane pary liczb (i, j).

Wywołanie (przez makro `EXECSTAGE`) procedury `glDispatchCompute` w linii 42 powoduje wykonanie, dla każdego iloczynu do obliczenia, instrukcji szadera w liniach 60–71. Pętla w liniach 60–63 znajduje numer p niezerowego współczynnika a_{ik}, który jest pierwszym czynnikiem, za pomocą wyszukiwania binarnego w ciągu sum prefiksowych obliczo-

nych w pierwszym etapie. Pętla w liniach 64–67, również metodą wyszukiwania binarnego, znajduje indeks i, tj. numer wiersza zawierającego współczynnik a_{ik} będący przedmiotem zainteresowania danego wątku szadera i zapamiętuje go (w linii 68) w buforze. Wartość przypisana zmiennej q w linii 69 jest numerem współczynnika b_{kj} w tablicy niezerowych współczynników macierzy B, a w linii 70 jest zapamiętany w buforze indeks j kolumny z tym współczynnikiem. W linii 71 współczynniki a_{ik} i b_{kj} są mnożone. Z powodu konieczności wyszukiwania indeksów koszt tego etapu, identyczny dla wszystkich iloczynów, jest rzędu sumy logarytmów liczby niezerowych współczynników macierzy A i liczby jej wierszy.

Po zapamiętaniu w buforach pomocniczych par (i, j) oraz iloczynów $a_{ik}b_{kj}$ dostęp do tablic z reprezentacjami macierzy A i B przestaje być potrzebny. W związku z tym, oraz dążeniem do zmieszczenia się w limicie ośmiu buforów magazynowych, do których szader obliczeniowy może mieć dostęp (zobacz p. 11.5.1), bufory, w których będzie umieszczony końcowy wynik (czyli reprezentacja macierzy C) zostaną przywiązane do punktów 0 i 1, dzięki którym wcześniej szader miał dostęp do macierzy A. Do zrobienia pozostało obliczenie sum właściwych iloczynów i znalezienie, dla każdej sumy, numeru wiersza i kolumny, na przecięciu których ta suma jest współczynnikiem macierzy C.

W liniach 44–48 następuje realokacja pierwszego bufora pomocniczego, ponieważ dalej trzeba będzie zmieścić w nim dwie tablice o długościach $T = \max(P, m) + 1$. Dostęp do tych tablic w treści szadera odbywa się za pomocą makrodefinicji `tab1` i `tab2`. Wartością zmiennej jednolitej `tablgt` jest liczba T, przypisywana w linii 43.

Kolejne cztery etapy mają posortować trójki $(i, j, a_{ik}b_{kj})$, aby iloczyny, które trzeba zsumować, znalazły się w tablicy obok siebie. Trójki już są uporządkowane względem indeksów i, zatem trzeba dla każdego i wyodrębnić odpowiedni podciąg trójek i uporządkować go względem j. Etap 3 (listing G.21, linie 74–84) znajduje metodą wyszukiwania binarnego i wpisuje do pierwszej tablicy $m+1$ liczb będących numerami miejsc, od których zaczynają się podciągi z danym indeksem i; ostatnia liczba jest równa P. W ten sposób różnice kolejnych liczb w tablicy są długościami odpowiednich podciągów do posortowania. Etap 4 oblicza te różnice, a etap 5 (listing G.20, linie 51–57, listing G.21, linie 90–93) znajduje największą z nich i przypisuje ją zmiennej `maxnt`.

Szósty etap jest sortowaniem podciągów trójek za pomocą algorytmu sieci sortującej realizowanego przez procedurę `_GPUNetSort` z listingu G.12; liczba sortowanych podciągów jest liczbą wierszy macierzy A i C. Liczba kroków sortowania jest określona przez największą długość podciągu. Globalna grupa robocza jest w tym etapie dwuwymiarowa. Jej pierwszy wymiar jest równy $\lceil \log_2 l \rceil - 1$, gdzie l jest największą długością sortowanego podciągu, a drugi wymiar jest liczbą podciągów (czyli liczbą wierszy macierzy C).

Współrzędne x i y wątku w grupie roboczej określają numer podciągu i numer komparatora zatrudnionego do posortowania tego podciągu. Numery te są parametrami procedury `SortIt` (listing G.21, linie 29–42), która na ich podstawie oblicza położenia trójek w tablicach i wykonuje zadanie komparatora. Warto zwrócić uwagę, że choć długości podciągów mogą się znacznie różnić (mogą być nawet podciągi puste), określenie liczby etapów sortowania na podstawie największej długości daje poprawny wynik: podciągi najkrótsze zostaną posortowane w początkowych etapach, a dalej komparatory, po porównaniu, niczego już nie

przestawią. Trzeba tylko sprawdzać, czy indeks drugiego elementu pary odpowiada elementowi tego samego podciągu, co jest robione w linii 36.

Listing G.21. Szader obliczeniowy mnożenia macierzy rzadkich

GLSL

```
1: #version 450 core
2:
3: layout(local_size_x=1) in;
4:
5: layout(std430,binding=0) buffer RCA    { uint  rc[]; } rca;
6: layout(std430,binding=1) buffer CoeffA { float a[]; }  aa;
7: layout(std430,binding=2) buffer RCB    { uint  rc[]; } rcb;
8: layout(std430,binding=3) buffer CoeffB { float a[]; }  ab;
9: layout(std430,binding=4) buffer Auxb0  { uint  a[]; }  seq;
10: layout(std430,binding=5) buffer Auxb1  { float a[]; }  aux1;
11: layout(std430,binding=6) buffer Auxb2  { uint  a[]; }  aux2;
12:
13: uniform int  stage;
14: uniform uint prN0, prN, prStep;
15: uniform uint ma, nnza, mb, nprod, nnzc, h, tablgt;
16: uniform bool reverse;
17:
18: #define ra(I) rca.rc[I]
19: #define ca(I) rca.rc[ma+1+(I)]
20: #define rb(I) rcb.rc[I]
21: #define cb(I) rcb.rc[mb+1+(I)]
22: #define pairi(I) aux2.a[2*(I)]
23: #define pairj(I) aux2.a[2*(I)+1]
24: #define tab1(I) seq.a[I]
25: #define tab2(I) seq.a[tablgt+(I)]
26:
27: void iPrefixSum ( uint i ) { .... } /* procedura z listingu G.8 */
28:
29: void SortIt ( uint ns, uint np )
30: {
31:   uint  i, j, l, h2;
32:   float x;
33:
34:   h2 = h >> 1;  l = np % h2;  np /= h2;
35:   i = tab1(ns)+np*h+l;  j = reverse ? tab1(ns)+(np+1)*h-l-1 : i + h2;
36:   if ( j < tab1(ns+1) ) {
37:     if ( pairj(i) > pairj(j) ) {
38:       l = pairj(i);   pairj(i) = pairj(j);    pairj(j) = l;
39:       x = aux1.a[i];  aux1.a[i] = aux1.a[j];  aux1.a[j] = x;
40:     }
41:   }
42: } /*SortIt*/
```

```
43:
44: void main ( void )
45: {
46:   uint  i, j, k, l, m, p, q;
47:   float s;
48:
49:   i = gl_GlobalInvocationID.x;
50:   switch ( stage ) {
51: case 0:
52:     iPrefixSum ( i );
53:     return;
54: case 1:
55:     if ( i == 0 ) seq.a[i] = 0;
56:     j = ca(i);
57:     seq.a[i+1] = rb(j+1)-rb(j);
58:     return;
59: case 2:
60:     for ( p = 0, k = nnza;  k-p > 1; ) {
61:       l = p + (k-p)/2;
62:       if ( i >= seq.a[l] ) p = l; else k = l;
63:     }
64:     for ( j = 0, k = ma;  k-j > 1; ) {
65:       l = j + (k-j)/2;
66:       if ( p >= ra(l) ) j = l; else k = l;
67:     }
68:     pairi(i) = j;
69:     q = rb(ca(p))+i-seq.a[p];
70:     pairj(i) = cb(q);
71:     aux1.a[i] = aa.a[p]*ab.a[q];
72:     return;
73: case 3:
74:     if ( i == 0 ) {
75:       tab1(0) = 0;
76:       tab1(ma) = nprod;
77:     }
78:     else {
79:       for ( j = 0, k = nprod;  k-j > 1; ) {
80:         l = j + (k-j)/2;
81:         if ( pairi(l) < i ) j = l; else k = l;
82:       }
83:       tab1(i) = k;
84:     }
85:     return;
86: case 4:
87:     tab2(i) = tab1(i+1)-tab1(i);
88:     return;
89: case 5:
```

```
90:     if ( (j = i+(prN+1)/2) < prN ) {
91:       if ( tab2(i) < tab2(j) )
92:         tab2(i) = tab2(j);
93:     }
94:     return;
95: case 6:
96:     SortIt ( i, gl_GlobalInvocationID.y );
97:     return;
98: case 7:
99:     if ( i == 0 )
100:      { tab1(0) = 0;  tab1(1) = 1; }
101:    else
102:      tab1(i+1) = pairi(i-1) != pairi(i) || pairj(i-1) != pairj(i) ? 1 : 0;
103:    return;
104: case 8:
105:    if ( i == nprod )
106:      tab2(nnzc) = nprod;
107:    else if ( tab1(i+1) > tab1(i) ) {
108:      ca(tab1(i)) = pairj(i);
109:      tab2(tab1(i)) = i;
110:    }
111:    return;
112: case 9:
113:    if ( tab2(i+1) > tab2(i) ) {
114:      for ( j = tab2(i), s = 0.0;  j < tab2(i+1);  j++ )
115:        s += aux1.a[j];
116:      aa.a[i] = s;
117:      tab1(i) = pairi(tab2(i));
118:    }
119:    return;
120: case 10:
121:    if ( i == 0 ) {
122:      ra(0) = 0;
123:      ra(ma) = nnzc;
124:    }
125:    else {
126:      for ( p = 0, k = nnzc;  k-p > 1; ) {
127:        l = p + (k-p)/2;
128:        if ( i > tab1(l-1) ) p = l; else k = l;
129:      }
130:      ra(i) = p;
131:    }
132:    return;
133: default:
134:    return;
135:  }
136: } /*main*/
```

W etapie siódmym (linia 59 na listingu G.20 oraz 99–102 na listingu G.21) do pierwszej tablicy są wpisywane jedynki i zera: jedynka odpowiada iloczynowi, który jest pierwszym składnikiem sumy do obliczenia (czyli jest pierwszym iloczynem z daną parą indeksów (i, j)), a zero odpowiada każdemu kolejnemu składnikowi. Wspomniane zera i jedynki są wpisywane „o jedno miejsce dalej", a na początek tablicy trafia 0. Po obliczeniu ciągu sum prefiksowych na końcu ciągu (w miejscu o numerze P) otrzymamy liczbę wpisanych jedynek, czyli liczbę N niezerowych współczynników macierzy C. Liczba ta jest odczytywana z bufora w liniach 61–62, a w linii 63 jest przypisywana zmiennej jednolitej `nnzc`.

W liniach 64–69 bufory, w których ma się znaleźć wynik, są przywiązywane do punktów dowiązania 0 i 1 i następuje rezerwacja bloków pamięci GPU o odpowiedniej długości.

W etapie ósmym (listing G.20, linia 70 i listing G.21, linie 105–110) dla każdego niezerowego współczynnika macierzy C do tablicy `c` będącej częścią reprezentacji tej macierzy jest wpisywany numer kolumny, w której jest ten współczynnik. Ponadto do drugiej tablicy pomocniczej jest wpisywany ciąg liczb, które są indeksami początków podciągów iloczynów do zsumowania. Warunek badany przez szader w linii 107 jest spełniony, gdy numer wątku jest numerem pierwszego iloczynu w sumowanym podciągu. Instrukcja przypisania w linii 106 „zakańcza" wpisywany do tablicy ciąg liczbą P na pozycji N. W rezultacie w tablicy `tab2` mamy ciąg liczb $t_0 = 0, t_1, \ldots, t_{N-1}, t_N = P$, taki że i-ty współczynnik macierzy C jest sumą iloczynów znajdujących się w tablicy `aux1` od miejsca t_i do $t_{i+1} - 1$.

W etapie dziewiątym (linia 71 i 113–118) następuje sumowanie składników w celu obliczenia współczynników c_{ij} macierzy C. Obliczona suma jest w linii 116 zapisywana w tablicy `a` reprezentacji macierzy C, a w linii 117 w tablicy pomocniczej jest zapamiętywany numer wiersza, w którym występuje obliczony współczynnik.

W etapie dziesiątym jest wypełniana tablica `r` reprezentacji macierzy C; są w niej zapisywane indeksy początków miejsc w tablicach `c` i `a`, od których zaczynają się numery kolumn i współczynniki w danym wierszu macierzy C. Indeksy te są znajdowane metodą wyszukiwania binarnego w tablicy `tab1`, w której (właśnie w tym celu) w poprzednim etapie zostały zapisane numery wierszy obliczonych współczynników. Działa to poprawnie także wtedy, gdy pewne wiersze macierzy C są zerowe. Wątek zerowy, oprócz zera na początku, wpisuje do tablicy `r` na pozycji m, która jest liczbą wierszy macierzy C, liczbę N niezerowych współczynników tej macierzy.

Tyle obliczeń na GPU. W liniach 73–76 procedura przypisuje liczbę niezerowych współczynników wyniku i identyfikatory zawierających go buforów zmiennym wskazywanym przez parametry, po czym sprząta po sobie.

Słowniki

H.1. Słownik TLS-ów i CzLS-ów

AABB — *axis-aligned bounding box*, prostopadłościan otaczający obiekt, jego część, zespół obiektów lub całą scenę, o krawędziach równoległych do osi (jakiegoś ustalonego) układu współrzędnych, zobacz OBB.

AFR — *alternating frame rendering*, naprzemienne wykonywanie klatek, technika tworzenia animacji na komputerze wyposażonym w więcej niż jedną GPU. Kolejne klatki są wykonywane przez inne GPU, co pozwala na zwiększenie liczby klatek na sekundę albo na zwiększenie stopnia skomplikowania rysowanych scen lub algorytmów rysowania.

AMD — Advanced Micro Devices, jeden z dwóch wiodących producentów CPU i jeden z dwóch wiodących producentów GPU.

ANSI — American National Standards Institute, organizacja, która opracowała kanoniczny standard języka C. Staram się go trzymać.

API — *application programming interface*, po włosku słowo *api* oznacza pszczoły.

ARB — OpenGL Architecture Review Board, komitet, który w latach 1992–2006 odpowiadał za rozwój standardu OpenGL, później wszedł w skład Khronos Group.

ATD — abstrakcyjny typ danych, czyli opis możliwych działań na obiekcie i ich skutków, umożliwiający programiście skupienie się na sposobie używania obiektu, bez rozpraszania uwagi na szczegóły jego implementacji (które mogą być różne). Przykładami ATD są stosy, kolejki i słowniki, a także konteksty OpenGL-a — maszyny stanów zdefiniowane w specyfikacji.

AZDO — *approaching zero driver overhead*, co proponuję spolszczyć na QZNS, czyli ku zerowemu narzutowi sterownika. Filozofia rozwoju standardu Vulkan na podstawie doświadczeń z OpenGL-em, zgodnie z którą dla optymalnego wykorzystania GPU cała odpowiedzialność za konfigurację obiektów używanych w procesie wykonywania obrazów, a także synchronizację współdziałania CPU i GPU, spada na autora aplikacji.

BFA — *brute force approach*, metoda brutalnej siły, czyli zamiana zadania trudniejszego (np. narysowania gładkiej powierzchni) na dużo zadań łatwych (np. narysowania mnóstwa trójkącików), które można rozwiązywać równolegle, gdy się ma odpowiedni sprzęt.

BFS — *breadth-first search*, przeszukiwanie grafu wszerz.

BLAS — *basic linear algebra subroutines*, podstawowe podprogramy algebry liniowej realizujące działania elementarne, takie jak mnożenie wektorów przez liczby, dodawanie wektorów, obliczanie iloczynu skalarnego, mnożenie macierzy przez wektor, mnożenie dwóch macierzy itd. Podprogramy te są zwykle zoptymalizowane do pracy na sprzęcie, na którym mają działać i są wykorzystywane m.in. przez bibliotekę LAPACK. Ich odpowiedniki działające na GPU są w bibliotece CUBLAS, dostępnej dla procedur napisanych w języku CUDA.

Także *bottom level acceleration structure*, struktura danych używana w śledzeniu promieni za pomocą GPU, zobacz TLAS.

blit — *block image transfer*, blokowe przesłanie obrazu, działanie wykonywane przez procedurę `glBlitFramebuffer`.

BRDF — *bidirectional reflectance distribution function*, dwukierunkowa funkcja rozkładu odbicia światła, składnik BSDF.

BSDF — *bidirectional scattering distribution function*, dwukierunkowa funkcja odbicia i załamania światła, centralny element modelu oświetlenia powierzchni. Jej argumentami są kierunki padania i odbicia lub załamania światła, a jej wartość jest ilorazem radiancji światła odbitego lub załamanego i irradiancji światła padającego na powierzchnię.

BTDF — *bidirectional transmission distribution function*, dwukierunkowa funkcja rozkładu załamania światła, drugi składnik BSDF.

BVH — *bounding volume hierarchy*, hierarchia otoczek części skomplikowanej sceny używana do przyspieszania rozwiązywania zadań takich jak znajdowanie przecięć promieni z obiektami lub wykrywanie kolizji obiektów. Często otoczkami są kule lub AABB.

CAD — *computer aided design*, projektowanie wspomagane komputerem.

CAGD — *computer aided geometric design*, modelowanie geometryczne, dział matematyki stosowanej zajmujący się teoretycznymi podstawami CAD.

CCD — *charge-coupled device*, współczesna namiastka szklanej płyty lub taśmy z octanu celulozy, pokrytej emulsją z bromkiem srebra.

CCW — *counterclockwise*, przeciwnie do ruchu wskazówek zegara.

CIE — Commission Internationale de l'Éclairage, Międzynarodowa Komisja Oświetleniowa, założona w roku 1913 organizacja zajmująca się opracowaniem standardów dla radiometrii, fotometrii i kolorymetrii.

CMY, **CMYK** — *Cyan, Magenta, Yellow,* (*blacK*), współrzędne używane w subtraktywnym modelu mieszania barw.

CPU — *central processing unit*, „główny" procesor, albo „jednostka centralna", stąd uznałem, że CPU jest rodzaju żeńskiego.

CRT — *cathode ray tube*, w zasadzie lampa kineskopowa, ale wciąż jeszcze tym skrótem w różnych pakietach oprogramowania (i bibliotekach procedur) określa się monitor.

CSR — *compressed sparse rows*, reprezentacja macierzy rzadkich opisana w podrozdziale G.4 i użyta w implementacji metody bilansu energetycznego w rozdziale 29 i w zagęszczaniu siatek w rozdziale 31. Jest też skrót CSC (*compressed sparse columns*), który oznacza podobną reprezentację, z zamienionymi rolami wierszy i kolumn.

CUDA — *Compute Unified Device Architecture*, język programowania GPU opracowany przez firmę NVIDIA. Jest on podobniejszy do C niż GLSL i w zastosowaniach niezwiązanych z potokiem przetwarzania grafiki wydaje się mieć większą siłę wyrazu. Zamiast szaderów obliczeniowych napisanych w GLSL-u można napisać odpowiedni program w języku CUDA, ale jego zastosowanie ograniczone jest do komputerów wyposażonych w GPU z procesorami firmy NVIDIA.

CzLS — czteroliterowy skrót, na przykład CzLS (nie mylić z TLS).

DFS — *depth-first search*, przeszukiwanie grafu w głąb.

DMA — *direct memory access*, układy wejścia/wyjścia umożliwiające przesyłanie danych m.in. między pamięcią operacyjną CPU a pamięcią GPU znacznie szybciej niż może to czynić CPU.

DOF — *degrees of freedom*, stopnie swobody, niezależne parametry artykulacji łańcucha kinematycznego[1].

Także *depth of field*, głębia ostrości obiektywu, którą można symulować przy użyciu bufora akumulacji.

DPI — *dots per inch*, jednostka rozdzielczości obrazu, liczba pikseli na cal.

DRI — *direct rendering interface*, zaimplementowana w bibliotece GLX możliwość przekazywania danych i poleceń między CPU a GPU z pominięciem protokołu komunikacyjnego systemu X Window, w celu przyspieszenia tworzenia grafiki.

DSA — *direct state access*, wprowadzony w specyfikacji OpenGL 4.5 dostęp do obiektów w pamięci GPU (buforów, tekstur i buforów ramki) bez przywiązywania ich do odpowiednich celów, realizowany za pomocą procedur mających słowo `Named` w nazwie.

EBO — *element buffer object*, to samo co IBO.

FBO — *framebuffer object*, obiekt bufora ramki.

FIFO — *first in, first out*, kolejka.

FPS — *frames per second*, liczba wyświetlanych klatek na sekundę. Także *first person shooting*, co można rozumieć jako strzelanie w pierwszej osobie lub strzelanie do pierwszej osoby. Ja tego nie robię.

[1] W łańcuchach otwartych wszystkie parametry artykulacji są niezależne.

FSF — *Free Software Foundation*, organizacja, która opracowała licencje GPL i LGPL.

GAF — *geometric attenuation factor*, funkcja opisująca wzajemne zasłanianie mikrościanek w opartych na prawach fizyki modelach oświetlenia powierzchni.

GCC — *GNU compiler collection*, pakiet kompilatorów różnych języków programowania, w tym języka C.

GDI — *graphics device interface*, biblioteka grafiki dwuwymiarowej, której można używać w natywnych aplikacjach systemu Windows do rysowania wihajstrów.

GGX — mimo starań nie udało mi się odnaleźć słów, z których powstał ten TLS. Bywa on używany jako określenie implementacji dwukierunkowej funkcji odbicia światła (BRDF) lub jej czynników: funkcji rozkładu kierunków wektora normalnego (NDF) lub funkcji zasłaniania mikrościanek (GAF).

GIGO — *garbage in, garbage out*, niezależnie, czy używamy kolejki (FIFO), czy stosu (LIFO).

GIMP — *GNU Image Manipulation Program*, program do obróbki obrazów rastrowych, przydaje się do przygotowywania tekstur i ilustracji.

GLEW — *OpenGL Extension Wrangler*, jedna z bibliotek udostępniających aplikacji adresy procedur OpenGL-a.

GLSL — *OpenGL shading language*, bohater tej książki.

GLU — *OpenGL utilities*, biblioteka pomocnicza, większość jej procedur jest dostosowana do starego OpenGL-a.

GLUT — *OpenGL Utility Toolkit*, historycznie pierwsza biblioteka z API dla interakcyjnych aplikacji OpenGL-a, umożliwiająca uniezależnienie aplikacji od systemu operacyjnego i systemu okien.

GNU — *GNU's not Unix*, TLS, w którym rekurencja służy kokieterii.

GPL — *GNU Public License*, opracowana przez FSF licencja, na zasadach której są rozpowszechniane liczne programy.

GPU — *graphics processing unit*, procesor grafiki. Inaczej „jednostka" lub „karta" graficzna, według mnie jest rodzaju takiego jak CPU.

GUI — *graphical user interface*, zestaw wyświetlanych na ekranie wihajstrów, które służą do interakcji użytkownika z programem. Zobacz też WIMP.

GWS — *global workgroup size*, wielkość globalnej grupy roboczej.

HDR — *high dynamic range*, szeroki zakres dynamiczny, reprezentacja obrazu, w której składowe r, g, b pikseli są reprezentowane przez liczby zmiennopozycyjne. Wyświetlenie takiego obrazu na ekranie musi być poprzedzone przekształceniem do postaci LDR, ale reprezentacja HDR umożliwia prowadzenie obliczeń z dużą dokładnością, a składowe pikseli nie muszą należeć do przedziału $[0,1]$.

HID — *human input device*, dowolne urządzenie umożliwiające wprowadzanie danych przez człowieka, na przykład klawiatura, mysz, dżojstik.

HLSL — *high level shading language*, język programowania GPU, który w standardzie DirectX firmy Microsoft pełni rolę analogiczną do GLSL-a.

HSL, **HSV** — *Hue, Saturation, Lightness, Value* (odcień, nasycenie, światłość, wartość), „malarskie" współrzędne barw, wymyślone jako wygodniejsze niż RGB dla użytkowników programów graficznych.

IBL — *image-based lighting*, oświetlenie przez obraz, tj. przy użyciu obrazu świata otaczającego rysowany obiekt — światłem odbitym lub wysyłanym przez obiekty dookoła.

IBO — *index buffer object*, bufor z indeksami do tablicy wierzchołków, umożliwia wygodne rysowanie łamanych, taśm trójkątowych lub wachlarzy określonych przez ciąg wierzchołków, które mogą się powtarzać.

IEEE — Institute of Electrical and Electronics Engineers, organizacja, która opracowała m.in. standard arytmetyki zmiennopozycyjnej IEEE 754, obejmujący reprezentacje liczb *oraz* najważniejsze własności działań na nich. CPU realizują ten standard w pełni, natomiast GPU zazwyczaj tylko w ograniczonym zakresie, o czym trzeba wiedzieć.

IFS — *iterated function system*, układ iterowanych przekształceń, jeden z modeli matematycznych używanych do otrzymywania obrazów figur fraktalowych.

JPEG — Joint Photographic Experts Group, komitet, który opracował algorytmy kompresji odpowiednie dla fotografii. Także format zapisu plików z obrazami, korzystający z tych algorytmów, które mogą być też stosowane do kompresji obrazów w zapisanych innych formatach, na przykład TIFF.

JPG — TLS CzLS-u JPEG używany jako rozszerzenie nazw plików w formacie JPEG.

KISS — *keep it simple, stupid!*, najważniejsza maksyma, która powinna zawsze przyświecać każdemu programiście. W praktyce, niestety, nie każdemu, nie zawsze, albo w ogóle nie przyświeca. Mam wrażenie, że twórcy standardu Vulkan o niej nie słyszeli.

KHR — Khronos Group, konsorcjum sprawujące od roku 2006 opiekę nad standardem OpenGL.

LCD — *liquid crystal display*, wyświetlacz ciekłokrystaliczny.

LDR — *low dynamic range*, wąski zakres dynamiczny, reprezentacja obrazu z pikselami reprezentowanymi przy użyciu 8 bitów na każdą ze składowych r, g, b koloru (bity te reprezentują liczniki ułamków o mianowniku 255, składowe należą więc do przedziału $[0,1]$). Taka reprezentacja nadaje się do bezpośredniego wyświetlenia na ekranie, ale jest niewystarczająca dla niektórych metod tworzenia obrazów. Zobacz HDR.

LGPL — *Lesser GNU Public License*, nieco inna licencja niż GPL.

LIFO — *last in, first out*, stos.

LLVM — *low level virtual machine*, niskopoziomowa maszyna wirtualna, obecnie jest to nazwa własna oderwana od słów, których jest skrótem. To jest kompilator dla wielu języków

programowania (początkowo C/C++), wytwarzający kod pośredni, który może być poddany optymalizacji przed przetworzeniem go na kod docelowy. Projekt ten był podstawą do opracowania reprezentacji SPIR-V szaderów.

LOD — *level of detail*, poziom szczegółowości modelu dostosowany do jego wielkości na obrazie. Nie należy rysować wentyli w kołach samochodu, jeśli obraz całego tego samochodu w narysowanym krajobrazie ma średnicę kilkunastu pikseli (co innego, jeśli obraz wentyla ma kilkanaście pikseli).

LSB — *least-significant bits*, bity na mniej znaczących pozycjach w reprezentacji liczby.

LTE — *light transfer equation*, równanie transportu światła, nazywane też równaniem bilansu energetycznego, jest ono matematycznym modelem globalnego oświetlenia sceny.

LWS — *local workgroup size*, wielkość lokalnej grupy roboczej, zadeklarowana w treści szadera obliczeniowego i dostępna w zmiennej wbudowanej `gl_WorkGroupSize`.

LZNK — liniowe zadanie najmniejszych kwadratów. Dla danej macierzy $A \in \mathbb{R}^{m \times n}$ i wektora $\boldsymbol{b} \in \mathbb{R}^m$ polega ono na znalezieniu takiego wektora $\boldsymbol{x} \in \mathbb{R}^n$, że długość wektora $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$ (równoważnie: suma kwadratów jego współrzędnych) jest najmniejsza. Jeśli układ równań $A\boldsymbol{x} = \boldsymbol{b}$ jest niesprzeczny, to każde jego rozwiązanie jest rozwiązaniem LZNK. Jeśli kolumny macierzy A są liniowo niezależne, to jest to tzw. **regularne** LZNK, które ma jednoznaczne rozwiązanie. Jeśli wiersze macierzy A są liniowo niezależne, to układ $A\boldsymbol{x} = \boldsymbol{b}$ jest niesprzeczny, ale dla $m < n$ ma nieskończenie wiele rozwiązań — wtedy stawia się tzw. **dualne** LZNK, które polega na znalezieniu rozwiązania najkrótszego lub położonego najbliżej danego wektora $\hat{\boldsymbol{x}} \in \mathbb{R}^n$. Jeśli i wiersze i kolumny macierzy A są liniowo zależne, to LZNK jest **nieregularne**; w zbiorze wektorów $\boldsymbol{x}$, takich że wektor $\boldsymbol{b} - A\boldsymbol{x}$ jest najkrótszy, trzeba znaleźć wektor najkrótszy lub najbliższy danego wektora $\hat{\boldsymbol{x}}$.

MIMD — *multiple instruction, multiple data*, komputer wieloprocesorowy, w którym każdy procesor może wykonywać w tym samym czasie inną instrukcję, zobacz SIMD.

MIP — *multum in parvo*, wiele w niewielu, określenie techniki teksturowania (mipmapingu, *MIP-mapping*) użytej w rozdziale 19.

MRT — *multiple render target*, jednoczesne wykonywanie wielu obrazów, na przykład na różnych warstwach jednego załącznika bufora ramki (wybieranych przez nadanie odpowiednich wartości zmiennej `gl_Layer` przez szader geometrii, zobacz rozdz. 26) lub na różnych załącznikach koloru bufora ramki (wybieranych przez kwalifikatory `layout(location=`*i*`)` zmiennych wyjściowych szadera fragmentów, rozdz. 27) albo w różnych klatkach (wybieranych za pomocą zmiennej `gl_Viewport` szadera geometrii, zobacz rozdz. 29).

MSAA — *multisampled antialiasing*, antyaliasing przez wielopróbkowanie. W tej technice szacuje się obszar piksela zajęty przez fragment powierzchni, zliczając (wybrane w pikselu) punkty należące do obrazu tego fragmentu, ale jego kolor (uwzględniający oświetlenie i teksturę) oblicza się tylko dla jednego (lub niewielu) punktów, porównaj z SSAA.

MSB — *most-significant bits*, bity na bardziej znaczących pozycjach w reprezentacji liczby.

MVP — *model–view–projection*, ciąg przekształceń opisujących kolejno przejścia od układu współrzędnych obiektu (modelu) do układu świata, obserwatora i kostki standardowej, definiujący rzutowanie obiektu trójwymiarowego na płaszczyznę obrazu. Macierz opisująca złożenie tych przekształceń jest równa PVM (zobacz rozdz. 6).

NaN — *not a number*, nie-liczba, ciąg bitów zapisany w zmiennej typu `float` lub `double` niereprezentujący żadnej liczby rzeczywistej ani nieskończoności.

NDC — *normalized device coordinates*, układ współrzędnych kostki standardowej; w tym układzie współrzędne kartezjańskie punktów bryły widzenia leżą w przedziale $[-1, 1]$.

NDF — *normal distribution function*, funkcja opisująca rozkład kierunków wektorów normalnych mikrościanek chropowatej powierzchni.

OBB — *oriented bounding box*, prostopadłościan otaczający obiekt, jego część lub zespół obiektów, którego krawędzie mogą być dowolnie obrócone względem osi układu współrzędnych, zobacz AABB.

ODW — ostatnia działająca wersja, czyli program, który działał, zanim postanowiliśmy go ulepszyć. Trzeba było zrobić kopię zapasową.

OOP — *object oriented programming*, programowanie obiektowe, czyli takie, w którym procedury (zwane metodami) są traktowane jak integralna część przetwarzanych przez nie danych. Język C++ ma dostosowaną do tego składnię, ale w C też tak można.

Oops — ups, to akurat nie jest skrót.

PBO — *pixel buffer object*, bufor z tablicą pikseli, a właściwie dowolnych danych reprezentowanych przez pojedyncze liczby, pary lub czwórki liczb, przetwarzanych przez szadery jako obraz (*image*).

PBR — *physically based rendering*, obrazowanie oparte na prawach fizyki. W węższym sensie jest to używanie zaawansowanych lokalnych i globalnych modeli oświetlenia, a w szerszym stosowanie w konstrukcji scen do narysowania odpowiednich modeli matematycznych, na przykład w animacji rozwiązywanie równań ruchu zgodnych z zasadami mechaniki.

PCF — *percentage-closer filtering*, technika antyaliasigu obrazu cienia opisana w p. 22.6.2.

PRAM — *parallel random access machine*, model matematyczny komputera z wieloma procesorami, badany w teorii złożoności obliczeniowej. Dość dobrą realizacją takiego modelu jest komputer z wielordzeniową CPU, a znacznie gorszą (z uwagi na to, że procesory nie działają całkowicie niezależnie) jest GPU.

POLA — *principle of least astonishment*, zasada minimalizacji zaskoczeń. Mianowicie, należy ich oszczędzać użytkownikom aplikacji.

PWN — dawniej Państwowe Wydawnictwo Naukowe, obecnie (od 1991 r.) Wydawnictwo Naukowe PWN.

QED — *quantum electrodynamics*, elektrodynamika kwantowa, czyli fizyczna teoria światła, oraz *quod erat demonstrandum*, czego należało dowieść. □

QZNS — zobacz AZDO.

RAM — *random access memory*, pamięć o dostępie bezpośrednim. Również *random access machine*, model matematyczny komputera składającego się z jednordzeniowego procesora z pamięcią RAM, badany w teorii złożoności obliczeniowej. Po angielsku *ram* to także baran.

RGB — *red, green, blue*, współrzędne w przestrzeni koloru.

RGBA — *red, green, blue, alpha*, współrzędne w przestrzeni koloru i składowa alfa, pomocnicza w tworzeniu obrazu.

RMS — *root mean square*, odchylenie standardowe, parametr rozkładu kierunków wektorów normalnych mikrościanek, z których składa się chropowata powierzchnia.

RRZ — równanie różniczkowe zwyczajne, opisujące na przykład ruch cząsteczki w rozdziale 24.

SIMD — *single instruction, multiple data*, komputer wieloprocesorowy, w którym wszystkie procesory w danej chwili wykonują tę samą instrukcję na różnych danych albo czekają. GPU jest takim komputerem. Zobacz MIMD.

SMF — *Simple Model Format*, format plików tekstowych przeznaczony do opisu modeli obiektów trójwymiarowych.

SPD — *spectral power distribution*, widmo rozkładu mocy, określona w przedziale długości fal światła widzialnego funkcja opisująca strumień energetyczny światła.

SPIR — *Standard Portable Intermediate Representation*, binarny format częściowo skompilowanych programów dla GPU, można go używać do rozpowszechniania szaderów bez udostępniania ich kodów źródłowych.

SSAA — *supersampled antialiasing*, antyaliasing przez nadpróbkowanie, kolor fragmentu oblicza się na podstawie kolorów wszystkich wybranych punktów w obszarze piksela. To jest bardziej czasochłonne niż wielopróbkowanie, zobacz MSAA.

SSAO — *screen space ambient occlusion*, metoda obliczania oświetlenia stosowana podczas opóźnionego cieniowania.

SSBO — *shader storage buffer object*, obiekt bufora magazynowego.

SVD — *singular value decomposition*, rozkład macierzy względem wartości szczególnych. Jednym z jego najważniejszych zastosowań jest rozwiązywanie nieregularnych LZNK.

TBO — *texture buffer object*, obiekt bufora tekstury, a właściwie bufor magazynowy udostępniony szaderom jako tekstura jednowymiarowa.

TIFF — *tagged image file format*, format plików do zapisu obrazów rastrowych, niesamowicie elastyczny.

TIGA — *Texas Instruments Graphics Architecture*, standard grafiki zbudowany w latach dziewięćdziesiątych XX wieku wokół procesorów TMS 34010 i TMS 34020, które były

pierwszymi możliwymi do zainstalowania w komputerach osobistych całkowicie programowalnymi GPU (choć wtedy ten TLS jeszcze nie istniał). Standard ten okazał się ślepą uliczką w rozwoju technologii, ale był w swoim czasie inspirujący.

TLAS — *top level acceleration structure*, struktura danych używana w śledzeniu promieni (realizowanym za pomocą obecnie najnowszych generacji GPU firmy NVIDIA), dostępnym w rozszerzeniu standardu Vulkan (niestety, nie OpenGL, nie tylko ja ubolewam nad tym zaniedbaniem). Określone przez aplikację zbiory trójkątów, reprezentowane przez struktury BLAS, są (w fazie preprocesingu) organizowane w TLAS, która umożliwia szybkie odnajdowanie trójkątów przeciętych przez promienie.

TLS — trzyliterowy skrót, na przykład TLS (nie mylić z CzLS).

UBO — *uniform buffer object*, bufor z blokiem zmiennych jednolitych.

ulp — *unit in the last position*, jednostka „rozdzielczości" dwójkowej reprezentacji liczby, czyli wartość bezwzględna przyrostu wartości x zmiennej danego typu spowodowanego zmianą najmniej znaczącego bitu tej zmiennej. Dla zmiennych stałopozycyjnych (całkowitych, np. `int`) jednostka ta nie zależy od wartości zmiennej i w zasadzie jest równa 1, chyba że zmienna reprezentuje licznik jakiegoś ułamka. Na przykład liczby ośmiobitowe w buforze obrazu lub teksturze reprezentują liczby rzeczywiste z przedziału $[0,1]$ — wtedy $\operatorname{ulp} x = \frac{1}{255}$.

Dla liczby zmiennopozycyjnej **znormalizowanej** $x = (-1)^s 2^{c-b}(1+m)$, składającej się z bitu znaku s, cechy c i (reprezentowanej przez t bitów i będącej ułamkiem z przedziału $[0,1)$) mantysy m jest $\operatorname{ulp} x = 2^{c-b-t}$. Jeśli $c = 0$, to mamy **liczbę nieznormalizowaną** $x = (-1)^s 2^{1-b} m$ i wtedy $\operatorname{ulp} x = 2^{1-b-t}$. Mantysy liczb pojedynczej precyzji (`float`) mają 23 bity, a mantysy liczb podwójnej precyzji (`double`) mają ich 52. Stałe b tych reprezentacji to odpowiednio 127 i 1023.

Ups — uch, pomyliłem się (zobacz Oops).

USB — *universal serial bus*, magistrala danych dla urządzeń wejścia/wyjścia, takich jak klawiatury, myszy, drukarki, pendrajwy, dyski zewnętrzne, dżojstiki, wentylatory, lampki, odkurzacze do klawiatury itp.

VAO — *vertex array object*, obiekt tablicy wierzchołków (zobacz hasło **lista** na s. 1188).

VBO — *vertex buffer object*, bufor przechowujący atrybuty wierzchołków, rejestrowany w VAO.

VESA — Video Electronics Standards Association, organizacja dbająca o to, aby monitory różnych producentów można było podłączać do komputerów różnych producentów.

WIMP — *windows, icons, menus, pointers*, dawno używany CzLS, w krótkim czasie wyparty przez TLS GUI, bo jak ktoś zauważył, każdy wolałby być *gui* niż *wimp*.

WMIM — Wydział Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego.

WWW — *world-wide web*, w praktyce wysypisko wszelkich wiadomości, w którym potrzebne (i rzetelne) informacje bywają trudne do znalezienia.

H.2. Słownik wyrazów wieloznacznych

bodziec — oddziaływujący na zmysły sygnał, na przykład świetlny lub dźwiękowy. Dawniej kij pasterski.

bufor — pierwotnie zderzak wagonu lub lokomotywy, ułatwiający ich sprzęgnięcie. W komputerach to najpierw był magazyn danych przekazywanych między urządzeniami działającymi z różną szybkością, na przykład procesorem i dyskiem lub drukarką. Obecnie słowo to oznacza dowolny magazyn danych w pamięci RAM CPU, a także GPU, ale nie tylko. Bufor ramki (*framebuffer*) to struktura danych, w której ramach odbywa się tworzenie obrazu, w *nomen omen* buforach (obrazu, głębokości i maski) zarejestrowanych jako załączniki bufora ramki.

funkcja — w matematyce jest to podzbiór iloczynu kartezjańskiego $A \times B$, czyli zbioru par (a, b) złożonych z elementów dowolnych zbiorów A i B. Piszemy $f\colon A \to B$, aby określić **dziedzinę** A i **zbiór wartości** B funkcji f. Każdy element zbioru A (**argument funkcji**) jest pierwszym elementem *jednej* pary tego podzbioru. To samo znaczenie co funkcja mają słowa **przekształcenie**, **przyporządkowanie**, **odwzorowanie** i kilka innych, używanych trochę rzadziej. Jeśli para (a, b) należy do (czyli jest elementem) funkcji f, to mówimy, że b jest **wartością funkcji** f w punkcie a, lub **obrazem punktu** a w przekształceniu f.

W informatyce „funkcja" to oficjalna nazwa dowolnego podprogramu w językach C i GLSL, ale w tej książce słowa tego używam tylko w odniesieniu do podprogramów obliczających pewien wynik i podających go przez swoją nazwę, przy czym wynik ten zależy tylko od parametrów tego podprogramu, który ponadto nie ma żadnych efektów ubocznych (w zasadzie tylko taki podprogram wypada uznać za realizację funkcji w sensie matematycznym). Pozostałe podprogramy określam słowem **procedura**.

iloczyn — działanie zwane też **mnożeniem**, albo wynik tego działania, określonego w sposób dostosowany do jego argumentów. **Iloczyn liczb** całkowitych lub rzeczywistych, jaki jest, każdy widzi. **Iloczyny dwóch macierzy**, AB, **macierzy i wektora**, $A\boldsymbol{x}$ oraz **liczby i macierzy**, aA, są opisane w podrozdziale 5.1. **Iloczyn funkcji** o tej samej dziedzinie jest funkcją $(fg)(x) = f(x)g(x)$. **Iloczyn tensorowy** funkcji o dziedzinach D_1 i D_2 jest funkcją określoną w zbiorze $D_1 \times D_2$ (**iloczynie kartezjańskim** dziedzin tych funkcji), $(f \otimes g)(u, v) = f(u)g(v)$, zobacz podrozdział 15.1. Szczególnym przypadkiem iloczynu tensorowego jest opisany w podrozdziale 17.1 **iloczyn sferyczny** dwóch płaskich krzywych, który jest konstrukcją płata powierzchni parametrycznej. Opisy **iloczynu skalarnego** $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ i **iloczynu wektorowego** $\boldsymbol{a} \wedge \boldsymbol{b}$ wektorów w przestrzeni $\mathbb{R}^3$ są podane w podrozdziale 5.5, a **iloczyn wektorowy** $\boldsymbol{a} \wedge \boldsymbol{b} \wedge \boldsymbol{c}$ w $\mathbb{R}^4$ jest przedstawiony w podrozdziale 15.2. **Iloczyn mieszany** wektorów $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{R}^3$ jest liczbą $\langle \boldsymbol{a} \wedge \boldsymbol{b}, \boldsymbol{c} \rangle = \det[\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}]$. W podrozdziale 28.3 jest użyty przykład **iloczynu skalarnego** $\langle f, g \rangle$ **funkcji** f i g określonych na sferze jednostkowej. Definicja **iloczynu liczb zespolonych** jest przypomniana w p. F.1.1, wreszcie w podrozdziale A.4 są rozpatrywane **iloczyny kwaternionów**, $q_1 \cdot q_2$.

lista — struktura danych, w której każdy element oprócz pewnej informacji przechowuje identyfikator (wskaźnik lub indeks) następnego elementu.

Lista obrazowa (*display list*) to w starym OpenGL-u reprezentacja ciągu wykonanych wywołań procedur OpenGL-a z odpowiednimi parametrami, umożliwiająca powtarzanie go bez ponownego obliczania wartości parametrów. Listy obrazowe i procedury ich obsługi w nowym OpenGL-u zostały zdeprecjonowane, a szkoda.

Listą obrazową dostępną w nowym OpenGL-u jest obiekt tablicy wierzchołków (VAO), w którym jest zapamiętany ciąg wywołań procedur określających miejsca, z których etap pobierania wierzchołków ma odczytać atrybuty wierzchołków podczas rysowania. Odtworzenie tego ciągu następuje po każdym wywołaniu procedury `glBindVertexArray`.

Lista parametrów to to, co w językach C i GLSL odróżnia identyfikator podprogramu (który ją ma) od identyfikatora zmiennej (która jej nie ma).

metoda — sposób robienia czegoś, na przykład metoda śledzenia promieni lub metoda bilansu energetycznego są sposobami obliczania oświetlenia.

W programowaniu obiektowym metoda to podprogram zintegrowany z pewną strukturą danych.

model — w matematyce stosowanej jest to wzór lub algorytm opisujący (zawsze w uproszczeniu) dowolne zjawisko, na przykład odbicie światła od powierzchni.

Słowo „model" oznacza również rysowany przedmiot lub jego reprezentację w pamięci komputera.

obiekt — figura geometryczna lub jej reprezentacja. Obiekt to też struktura danych wyposażona w **metody**, czyli podprogramy oficjalnie uznawane za nieodłączne części takiej struktury. W nomenklaturze OpenGL-a to także używany w określonej roli bufor lub inna struktura danych w pamięci GPU, na przykład FBO, IBO, TBO, SSBO, UBO, VAO, VBO (zobacz podrozdz. H.1).

obraz — wartość funkcji f w danym punkcie (obraz punktu) lub zbiór wartości funkcji we wszystkich punktach pewnego zbioru (obraz zbioru).

W grafice komputerowej obraz jest wyświetlonym na ekranie lub wydrukowanym powodem stosowania grafiki, ale także reprezentacją takiego obrazu w postaci tablicy pikseli (w odpowiednim buforze), a nawet dowolnym zbiorem danych przechowywanych w buforze obrazu (*image buffer*).

orientacja — relacja równoważności określona między uporządkowanymi liniowo niezależnymi n-tkami wektorów w n-wymiarowej rzeczywistej przestrzeni liniowej (a także między układami współrzędnych kartezjańskich). Dwie takie n-tki są **zorientowane zgodnie**, jeśli wyznacznik macierzy przekształcenia przeprowadzającego jedną z nich na drugą jest dodatni, i **przeciwnie**, jeśli jest ujemny.

Orientacja danej n-tki wektorów (lub danego układu współrzędnych) jest to należenie do jednej z dwóch klas abstrakcji powyższej relacji. W przestrzeni trójwymiarowej mówi się o orientacji **prawoskrętnej** i **lewoskrętnej** (zobacz podrozdz. 5.6). W płaszczyźnie wyróżnia się orientację **zgodną z ruchem wskazówek zegara** i orientację **przeciwną do ruchu wskazówek zegara**. W przestrzeni jednowymiarowej orientacja jest **zwrotem** wektora.

Słowem „orientacja" potocznie określa się też obrót, który razem z odpowiednim przesunięciem nadaje obiektowi bieżące położenie w przestrzeni.

osobliwość — najogólniej, jest to miejsce nieciągłości jakiejś funkcji związanej z reprezentacją krzywej lub powierzchni, które może być widoczne na obrazie. Na przykład osobliwością jest węzeł krzywej sklejanej: w węźle o krotności r krzywej stopnia n może być nieciągła pochodna rzędu $n - r + 1$ parametryzacji, jeśli więc $r = n$, to możliwa jest nieciągłość pochodnej widoczna jako punkt załamania, a jeśli $r = n - 1$, to nieciągłość podchodnej drugiego rzędu powoduje skokową zmianę krzywizny krzywej.

Inny przykład to punkt, w którym pochodne cząstkowe parametryzacji powierzchni są liniowo zależne. Wyznacznik macierzy tzw. pierwszej formy podstawowej jest w takim punkcie równy 0, a ponieważ we wzorach opisujących krzywizny powierzchni występuje on w mianowniku, krzywizny mogą być nieograniczone (co oznacza brak ich ciągłości), tak jak w wierzchołku stożka. Nawet jeśli powierzchnia jest gładka (jak dno i pokrywka czajnika), to iloczyn wektorowy liniowo zależnych pochodnych cząstkowych jest wektorem zerowym i wektor normalny trzeba znaleźć w inny sposób.

Osobliwość macierzy kwadratowej to liniowa zależność jej kolumn, macierz osobliwa nie ma odwrotności, a przekształcenie przez nią opisane nie ma przekształcenia odwrotnego.

parametr — dla funkcji wielu zmiennych jest to zmienna (argument) o ustalonej wartości podczas badania własności funkcji (np. czy jest rosnąca) ze względu na pozostałe zmienne.

Parametr krzywej lub powierzchni jest to argument funkcji zwanej **parametryzacją** tej krzywej lub powierzchni.

Parametr artykulacji określa wzajemne położenie członów łańcucha kinematycznego związanych w parę kinematyczną, jeśli na przykład jest ona zawiasem, to jest to kąt obrotu wokół jego osi.

Parametr formalny podprogramu jest to zmienna zadeklarowana w nagłówku, otrzymuje wartość początkową w chwili wywołania. Jest to wartość parametru aktualnego, czyli odpowiedniego wyrażenia podanego w wywołaniu tego podprogramu.

płat — powierzchnia lub fragment powierzchni dany za pomocą parametryzacji, tj. funkcji wektorowej określonej w obszarze płaskim, na przykład płat Béziera lub B-sklejany.

W OpenGL-u płat jest to prymityw, którego obróbka w potoku przetwarzania grafiki ma etap rozdrabniania; w jego wyniku powstają odcinki lub trójkąty, które mogą przybliżać gładką krzywą lub powierzchnię.

W opisanej w rozdziale 29 implementacji metody bilansu energetycznego słowo płat oznacza zestaw trójkątów dzielony na elementy i makroelementy w celu dokonania dyskretyzacji równania bilansu energetycznego.

prymityw — figura geometryczna zdefiniowana lub przetwarzana „w całości", niezłożona z prostszych obiektów.

W poszczególnych etapach potoku przetwarzania grafiki co innego może być prymitywem: dla etapu pobierania wierzchołków prymitywem może być zbiór punktów, zbiór odcinków, łamana, zbiór trójkątów, taśma trójkątowa, wachlarz trójkątów lub płat. Dla szadera geometrii lub obcinania prymitywami są tylko pojedyncze punkty, odcinki i trójkąty, ale wyjściem szadera geometrii (z którego powstają prymitywy dla etapu obcinania) są zbiory punktów, łamane lub taśmy trójkątowe.

przestrzeń — w matematyce to dowolny zbiór, najczęściej słowo to jest używane w geometrii (a także, nie bójmy się tego słowa, topologii) i może oznaczać przestrzeń jednowymiarową (np. prostą), dwuwymiarową (np. płaszczyznę) lub przestrzeń o większym wymiarze.

Przestrzeń kolorów to w istocie układ współrzędnych używanych do opisu światła. Faktyczny zbiór kolorów (przestrzeń w sensie matematycznym, będąca obrazem zbioru widm promieniowania elektromagnetycznego w pewnym przekształceniu liniowym) jest jeden, a w nim są określone różne układy współrzędnych nazywane przestrzeniami kolorów. To jest powszechnie przyjęty błąd pojęciowy.

W informatyce jest mowa o **przestrzeniach nazw**, czyli wykazach identyfikatorów, w których są przechowywane nazwy podprogramów standardowych w danym języku lub nazwy zmiennych, podprogramów i innych obiektów zadeklarowanych w programie.

punkt — w matematyce element dowolnego zbioru, czasami pojęcie to bywa zawężone do elementów przestrzeni geometrycznej.

Punkt dowiązania to element tablicy w **indeksowanym celu** OpenGL-a. W ten sposób jeden cel (np. `GL_UNIFORM_BUFFER` lub `GL_TEXTURE_2D`) umożliwia dostęp szadera do wielu obiektów (bloków zmiennych jednolitych lub tekstur).

rzut — w matematyce dowolna funkcja $f\colon A \to A$, taka że $f\big(f(a)\big) = f(a)$ dla każdego $a \in A$; jest też w tym znaczeniu używane określenie **przekształcenie idempotentne**. Dla takiej funkcji, jeśli $b = f(a)$, to mówimy też, że punkt b jest rzutem punktu a.

W grafice najczęściej rzut oznacza przekształcenie przestrzeni (bryły widzenia), w której są określone obiekty do narysowania, na obszar tworzonego obrazu. Może być rzut równoległy, perspektywiczny lub nieliniowy (np. panoramiczny).

tekstura — funkcja opisująca dowolną własność, która wpływa na wygląd punktów rysowanej powierzchni na obrazie.

W OpenGL-u tekstury są reprezentowane przez tablice tzw. **tekseli**, przechowywujących wartości takiej funkcji w skończenie wielu punktach.

topologia — w matematyce jest to rodzina tzw. **podzbiorów otwartych** dowolnego zbioru, zwanego **przestrzenią topologiczną**, spełniająca aksjomaty podane w podręcznikach.

Dla powierzchni (zbioru punktów) zbudowanej z wielokątów informacja o ich połączeniach wzdłuż wspólnych krawędzi określa topologię w sensie podanym wyżej, co usprawiedliwia używanie słowa „topologia” dla tej informacji.

wektor — obiekt, który można mnożyć przez skalary (tj. liczby) i dodawać do innych obiektów tego samego rodzaju. Najczęściej słowo to oznacza macierz kolumnową składającą się ze współrzędnych punktu, ale to może być też dowolna macierz, przekształcenie liniowe lub funkcja o określonej dziedzinie.

Łacińskie słowo VECTOR, zanim Galileusz wprowadził je do matematyki, oznaczało tragarza.

węzeł — wyróżniony punkt w dziedzinie funkcji skalarnej lub wektorowej (parametryzacji krzywej). **Węzły interpolacyjne** są to punkty u_i, dla których funkcja ma przyjmować zadane wartości. **Węzły funkcji** lub **krzywej sklejanej** to punkty u_i rozgraniczające przedziały, w których ta funkcja lub krzywa jest opisana przy użyciu różnych wielomianów. **Węzły kwadratury** to punkty, w których są obliczane wartości funkcji w celu numerycznego obliczenia całki z tej funkcji.

wierzchołek — w geometrii jest to punkt będący wspólnym końcem odcinków rozpatrywanej łamanej (która *może* być brzegiem wielokąta, wtedy jest to wierzchołek wielokąta, a także bryły wielościennej, której ten wielokąt jest ścianą).

Wierzchołek stożka jest punktem wspólnym wszystkich jego tworzących. Na powierzchni stożka jest to jedyna osobliwość, czyli punkt, w którym kierunek wektora normalnego nie jest określony.

W teorii grafów wierzchołek jest obiektem — elementem (dowolnie określonego) zbioru wierzchołków grafu. Obiekt ten może mieć dowolne atrybuty, jeśli na przykład jest punktem w przestrzeni, to ma określone położenie. Wierzchołek grafu opisującego łańcuch kinematyczny jest członem łańcucha.

W OpenGL-u wierzchołek jest obiektem o atrybutach opisanych przez liczby, przy czym najczęściej jest to punkt w przestrzeni. Wierzchołki są przekazywane między kolejnymi etapami potoku przetwarzania grafiki, ale dopiero ostatni szader części przedniej potoku w programie szaderów, jeśli ma spowodować rysowanie, musi przekazać (do etapu obcinania) wierzchołki o ustalonym położeniu w przestrzeni (w układzie współrzędnych kostki standardowej).

Jest jeszcze wierzchołek góry lodowej — to jest część OpenGL-a opisana w tej książce.

Tak zwana Metoda Bromby polega na tym,
że czytamy hasła jedno po drugim
w kolejności alfabetycznej i sprawdzamy,
czy to prawda, że są na podanej stronie.

Maciej Wojtyszko: *Bromba i inni.*

Skorowidz

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 III zawiera wiadomości uzupełniające:

- opis prostego interfejsu użytkownika w środowisku X Window,
- opis i implementację algorytmu zagęszczania siatek reprezentujących powierzchnie gładkie,
- aplikację rysującą taką powierzchnię,
- łańcuch kinematyczny umożliwiający animowanie odkształceń powierzchni,
- opis i implementację sposobu wprawiania animacji w ruch,
- opis zastosowania kwaternionów do reprezentowania i animowania obrotów w przestrzeni,
- opis krzywych i powierzchni B-sklejanych, umożliwiających modelowanie obiektów bardziej skomplikowanych niż płaty Béziera,
- podstawy kolorymetrii,
- sposób użycia dżojstika w aplikacjach X Window,
- sposoby nieliniowego rzutowania przestrzeni na płaskie obrazy,
- sposoby rysowania obiektów fraktalowych: zbioru Mandelbrota, piramidy Sierpińskiego i gąbki Mengera,
- algorytmy masywnie równoległych obliczeń na karcie graficznej niezwiązanych bezpośrednio z grafiką: sumowania długich ciągów, obliczania sum prefiksowych, sortowania i przetwarzania nieregularnych macierzy rzadkich.

Cz. I–III
ISBN 978-83-971793-0-1
9 788397 179301

Cz. III
ISBN 978-83-971793-3-2
9 788397 179332