O OpenGL — wprowadzenie

O.1 Informacje ogólne

Dawno, dawno temu firma Silicon Graphics opracowała dla produkowanego przez siebie sprzętu bibliotekę graficzną zwaną IRIS GL. Związany z nią interfejs programisty był zależny od sprzętu i od systemu operacyjnego. Rozwój kolejnych wersji sprzętu (w tym uproszczonego, w którym procedury rysowania musiały być realizowane programowo) wymusił kolejne zmiany w kierunku niezależności sprzętowej. Uniezależnienie sposobu programowania grafiki od sprzętu i środowiska, w jakim działa program, zostało zrealizowane pod nazwą OpenGL. Umożliwiło to opracowanie implementacji współpracujących z zupełnie różnymi systemami okienkowymi, np. XWindow, Apple, OS/2 Presentation Manager i inne. Pierwsza specyfikacja standardu OpenGL, 1.0, została opublikowana w 1992r. O jakości tego projektu świadczy mała liczba rewizji (do tej pory 10); kolejne wersje, oznaczone numerami 1.1–1.5, pojawiły się w latach 1992–2003. Specyfikacje o numerach 2.0, i 2.1 ukazały się w latach 2004 i 2006. Specyfikacja 3.0 jest datowana na rok 2008, obecnie najnowsza jest specyfikacja 3.2 (sierpień 2009).

Prawdopodobnie największa część sprzętu spotykanego obecnie jest zgodna ze specyfikacją 2.0 lub 2.1. Dlatego ten opis jest zgodny z tymi specyfikacjami. Niestety, przejście do specyfikacji o głównym numerze 3 wiąże się z uznaniem większości opisanych tu procedur za przestarzałe (jak wiadomo, lepsze jest najgorszym wrogiem dobrego, oczywiście wszystko jest usprawiedliwiane dążeniem do usprawnienia wielu rzeczy) i w przyszłości to się niestety zdezaktualizuje (należy się liczyć z tym, że opisane tu procedury zostaną ze standardu usunięte).

Opublikowanie standardu umożliwiło powstanie niezależnych implementacji. Firmy inne niż Silicon Graphics produkują sprzęt („karty graficzne”), który kosztuje niewiele i udostępnia OpenGL-a na pecetach. Producenci sprzętu dostarczają odpowiednie sterowniki pracujące w najpopularniejszych systemach operacyjnych. Oprócz tego istnieją implementacje niezależne. Jedna z nich, zwana Mesa, której autorem jest Brian Paul, była czysto programowa (dzięki czemu jej używanie nie wymaga posiadania specjanego sprzętu), ale mogła współpracować z odpowiednim sprzętem, jeśli taki był zainstalowany (dzięki czemu ten sprzęt się nie marnował i grafika była wyświetlana szybko). W trakcie prac nad implementacją XFree86 systemu XWindow Mesa została zintegrowana z tym systemem, dzięki czemu OpenGL jest powszechnie dostępny także na zasadach Wolnego Oprogramowania. Dobrą wiadomością jest też fakt, że wielu producentów kart graficznych do pecetów dostarcza sterowniki do swojego sprzętu dla systemu XFree86 (gorszą wiadomością jest to, że producenci nie publikują kodu źródłowego tych sterowników, co wywołuje irytację osób traktujących wolność oprogramowania jak najważniejszą filozofię życiową).

OpenGL ma m.in. następujące możliwości:

- Wykonywanie rysunków kreskowych, z cieniowaniem głębokości (ang. *depth cueing*), obcinaniem głębokości (*depth culling*) i antyaliasingiem,
- Wyświetlanie scen zbudowanych z wielokątów z uwzględnieniem widoczności i oświetlenia, nakładaniem tekstury i efektami specjalnymi, takimi jak mgła, głębia ostrości, jest też pewne wspomaganie wyznaczania cieni,
- Ułatwienia w programowaniu animacji — można skorzystać z podwójnego buforowania obrazu, a także otrzymać rozmycie obiektów w ruchu (ang. *motion blur*),
- Obsługa list obrazowych, wspomaganie wyszukiwania obiektów w scenie,
- Programowanie bezpośrednio sprzętu graficznego; obecnie porządne karty graficzne zawierają procesory o mocy kilkakrotnie większej niż główny procesor komputera; jest to osiągane przez zwielokrotnienie procesora, który wykonuje obliczenia masywnie równolegle (np. w trybie SIMD — ang. *single instruction, multiple data*), co jest naturalne podczas teksturowania. Ten kierunek rozwoju standardu obecnie dominuje.
- Praca w sieci, w trybie klient/serwer.

Specyfikacja 2.0 i późniejsze określają język programowania sprzętu (zwany GLSL i podobny do języka C); napisane w nim procedury[1)] są wykonywane przez procesor graficzny i służą do nietypowego przetwarzania danych geometrycznych i tekstur nakładanych na wyświetlane figury. Niniejszy opis OpenGL-a traktuje jednak o podstawach. Najważniejsze, to zacząć pisać programy; apetyt przychodzi (albo przechodzi) w miarę jedzenia.

[1)] które już się niestety doczekały „polskiej” nazwy „szadery”!?

O.1.1 Biblioteki procedur

Programista aplikacji ma możliwość użycia następujących bibliotek:

- GL — procedury „niskiego poziomu", które mogą być realizowane przez sprzęt. Figury geometryczne przetwarzane przez te procedury, to punkty, odcinki i wielokąty wypukłe. Nazwy wszystkich procedur z tej biblioteki zaczynają się od liter `gl`. Plik nagłówkowy procedur z tej biblioteki można włączyć do programu pisząc `#include <GL/gl.h>`.
- GLU — procedury „wyższego poziomu", w tym dające możliwości określania częściej spotykanych brył (sześcian, kula, stożek, krzywe i powierzchnie Béziera i NURBS). Nazwy procedur zaczynają się od `glu`, a plik nagłówkowy jest włączany do programu poleceniem `#include <GL/glu.h>`.
- GLX, AGL, PGL, WGL — biblioteki współpracujące z systemem okien, specyficzne dla różnych systemów. Procedury w tych bibliotekach mają nazwy zaczynające się od `glx`, `pgl`, `wgl`. Aby umieścić w programie odpowiednie deklaracje dla biblioteki GLX, współpracującej z systemem XWindow, należy napisać

```
#include <X11/Xlib.h>
#include <GL/glx.h>
```

- GLUT — biblioteki całkowicie ukrywające szczegóły współpracy programu z systemem okienkowym (za cenę pewnego ograniczenia możliwości). Poszczególne wersje współpracują z procedurami z biblioteki GLX, PGL lub WGL, ale udostępniają taki sam interfejs programisty. Procedury realizujące ten interfejs mają na początku nazwy przedrostek `glut`, a ich nagłówki umieszczamy w programie pisząc `#include <GL/glut.h>`. Nie musimy wtedy pisać dyrektyw `#include` dla plików `gl.h` ani `glu.h`, bo włączenie pliku `glut.h` spowoduje włączenie pozostałych dwóch. Nie powinniśmy również włączać plików `glx.h` lub pozostałych, bo używamy GLUTa aby mieć program niezależny od środowiska, w którym ma działać[2)].

O.1.2 Reguły nazewnictwa procedur

Nazwy procedur są zwięzłe, ale raczej znaczące. Przedrostek `gl`, `glu`, `glx`, `glut` zależy od biblioteki, natomiast przyrostek określa typ argumentu lub argumentów. Wiele procedur ma kilka wariantów, które wykonują to samo zadanie, ale różnią

[2)] Oryginalny projekt GLUT nie jest już rozwijany, niemniej będziemy korzystać z tej biblioteki. Istnieje niezależna implementacja, zwana FreeGLUT, której można używać zamiast GLUTa.

się typem i sposobem przekazywania argumentów. Przyrostki nazw procedur i odpowiadające im typy argumentów są następujące:

b	`GLbyte`	l. całkowita 8-bitowa,
ub	`GLubyte`, `GLboolean`	l. 8-bitowa bez znaku,
s	`GLshort`	l. całkowita 16-bitowa,
us	`GLushort`	l. 16-bitowa bez znaku,
i	`GLint`, `GLsizei`	l. całkowita 32-bitowa,
ui	`GLuint`, `GLenum`, `GLbitfield`	l. 32-bitowa bez znaku,
f	`GLfloat`, `GLclampf`	l. zmiennopozycyjna 32-bitowa,
d	`GLdouble`, `GLclampd`	l. zmiennopozycyjna 64-bitowa.

Typ `GLenum` jest zbiorem obiektów zdefiniowanych przez podanie listy identyfikatorów. Typy `GLclampf` i `GLclampd` są zbiorami liczb zmiennopozycyjnych z przedziału [0, 1].

Przed oznaczeniem typu bywa cyfra, która oznacza liczbę argumentów. Na samym końcu nazwy procedury może też być litera `v`, której obecność oznacza, że argumenty mają być umieszczone w tablicy, a parametr wywołania procedury jest wskaźnikiem (adresem) tej tablicy.

Powyższe reguły nazwenictwa możemy prześledzić na przykładze procedur wyprowadzania punktu i koloru:

```
glVertex2i ( 1, 2 );
glVertex3f ( 1.0, 2.71, 3.14 );
glColor3f ( r, g, b );
glColor3fv ( kolor );
```

Procedura `glVertex*` wprowadza do systemu rysującego punkt; procedura z końcówką nazwy `2i` otrzymuje dwie współrzędne, które są liczbami całkowitymi; końcówka `3f` oznacza wymaganie podania trzech liczb zmiennopozycyjnych o pojedynczej precyzji (32-bitowych). Sposób przetwarzania punktów wprowadzonych za pomocą każdej z tych procedur jest zależny tylko od kontekstu wywołania, a nie od tego, która z nich była użyta. Punkty wprowadzane przez `glVertex*` mogą mieć 2, 3 lub 4 współrzędne, ale to są współrzędne jednorodne. Podanie mniej niż czterech współrzędnych powoduje przyjęcie domyślnej wartości ostatniej współrzędnej (wagowej) równej 1. Podanie tylko dwóch współrzędnych daje punkt o trzeciej współrzędnej równej 0.

Procedura `glColor*` może być wywołana w celu określenia koloru, za pomocą trzech liczb z przedziału [0, 1] (typu `GLclampf`). Można je podać jako osobne parametry procedury `glColor3f`, albo umieścić w tablicy, np.

```
GLclampf kolor[3] = { 0.5, 1.0, 0.2 };
```

i wywołać procedurę `glColor3fv` (tak jak w podanym wcześniej przykładzie).

O.1.3 OpenGL jako automat stanów

Istnieje pewien zestaw „wewnętrznych” zmiennych przetwarzanych przez biblioteki, które określają stan (tryb pracy), w jakim system się znajduje. Zmienne te mają określone wartości domyślne, które można zmieniać przez wywołanie procedur `glEnable ( ... );` i `glDisable ( ... );`, a także poznawać ich wartości — jest tu cała menażeria procedur o nazwach `glGet...`, `glIsEnabled`, `glPush...`, `glPop...`. W kategoriach stanu automatu można też rozpatrywać zmienne określające kolor, zródła światła, sposób rzutowania itd.

O.2 Podstawowe procedury rysowania

Rysowanie czegokolwiek zaczyna się od wywołania procedury `glBegin`, która ma jeden parametr typu `GLenum`. Parametr ten musi mieć jedną z dziesięciu opisanych niżej wartości:

`GL_POINTS` — kolejne punkty są traktowane indywidualnie, na przykład rzuty tych puntków są rysowane w postaci kropek.

`GL_LINES` — każde kolejne dwa punkty są końcami odcinków.

`GL_LINE_STRIP` — kolejne punkty są wierzchołkami łamanej otwartej.

`GL_LINE_LOOP` — kolejne punkty są wierzchołkami łamanej zamkniętej.

`GL_TRIANGLES` — kolejne trójki punktów są wierzchołkami trójkątów.

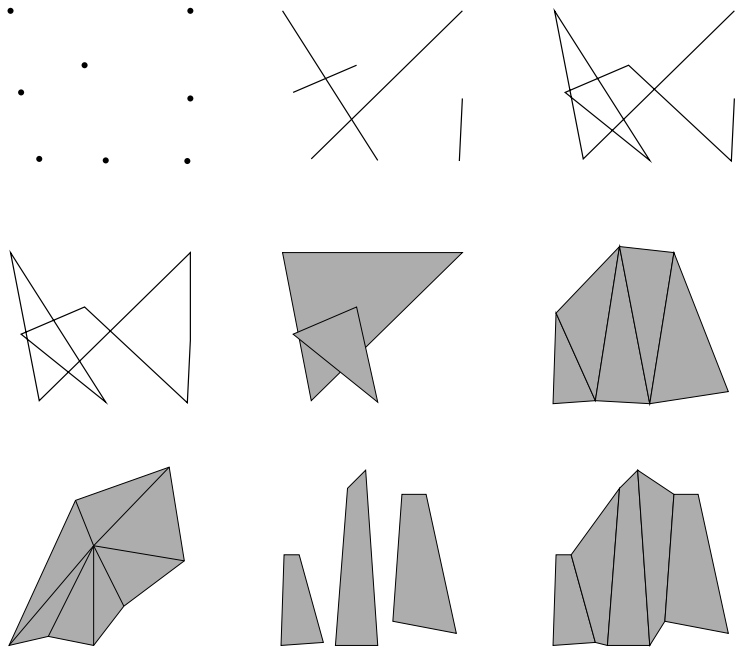
`GL_TRIANGLE_STRIP` — tak zwana taśma trójkątowa; po wprowadzeniu dwóch pierwszych punktów, każdy kolejny punkt powoduje wygenerowanie trójkąta, którego wierzchołkami są: ten punkt i dwa ostatnie wprowadzone wcześniej.

`GL_TRIANGLE_FAN` — wprowadzane punkty spowodują wygenerowanie trójkątów. Jednym z wierzchołków wszystkich tych trójkątów jest pierwszy punkt, dwa pozostałe to ostatni i przedostatni wprowadzony punkt. Liczba trójkątów, podobnie jak poprzednio, jest o 2 mniejsza od liczby punktów.

`GL_QUADS` — czworokąty, wyznaczone przez kolejne czwórki punktów. Uwaga: każda taka czwórka punktów musi być współpłaszczyznowa.

`GL_QUAD_STRIP` — taśma z czworokątów. Każda kolejna para punktów, z wyjątkiem pierwszej, powoduje wygenerowanie czworokąta, którego wierzchołkami są ostatnie cztery wprowadzone punkty.

`GL_POLYGON` — wielokąt wypukły. Rysując czworokąty jak i wielokąt należy zadbać o to, aby wszystkie wierzchołki leżały w jednej płaszczyźnie.



Rysunek O.1. Interpretacja wierzchołków wprowadzanych w różnych trybach.

Koniec wprowadzania punktów w danym trybie sygnalizuje się przez wywołanie procedury `glEnd ();`. Przykład:

```
glBegin (GL_POLYGON);
  glVertex2f ( x1, y1 );
  ...
  glVertex2f ( xn, yn );
glEnd ();
```

Między wywołaniami procedur `glBegin` i `glEnd` dopuszczalne jest wywoływanie następujących procedur z biblioteki GL (gwiazdka w nazwie powinna być zastąpiona odpowiednią końcówką, zgodnie z regułami w p. O.1.2):

`glVertex*(...);`	—	punkt o podanych współrzędnych,
`glColor*(...);`	—	kolor,
`glIndex*(...);`	—	kolor w trybie z paletą,
`glNormal*(...);`	—	wektor normalny,
`glTexCoord*(...);`	—	współrzędne tekstury,
`glEdgeFlag*(...);`	—	określenie, czy krawędź leży na brzegu wielokąta,
`glMaterial*(...);`	—	określenie właściwości materiału,
`glArrayElement(...);`	—	współrzędne punktu, kolor itd. z zarejestrowanej wcześniej tablicy,
`glEvalCoord(...);`		
`glEvalPoint*(...);`	—	punkt na krzywej lub powierzchni Béziera,
`glCallList();`		
`glCallLists();`	—	wyprowadzenie zawartości listy obrazowej.

Lista obrazowa jest strukturą danych, w której są przechowywane polecenia równoważne wywołaniom procedur OpenGL-a z odpowiednimi parametrami. Lista, której zawartość jest wyprowadzana między wywołaniami `glBegin` i `glEnd` musi zawierać tylko polecenia dozwolone w tym momencie.

Jeśli wyprowadzając punkt, podamy mniej niż cztery współrzędne, (np. dwie lub trzy), to domyślna wartość trzeciej współrzędnej jest równa 0, a czwartej (wagowej) 1. Możemy więc podawać współrzędne kartezjańskie lub jednorodne.

Zasada wyświetlania punktów (wierzchołków) wyprowadzanych za pomocą `glVertex*` jest taka, że własności obiektu w punkcie, który wprowadzamy wywołując `glVertex*`, na przykład kolor, należy określić *wcześniej*. Zanim cokolwiek wyświetlimy, należy określić odpowiednie rzutowanie, o czym będzie mowa dalej.

O.2.1 Wyświetlanie obiektów

Dla ustalenia uwagi, opis dotyczy rysowania z użyciem bufora głębokości. Aby go uaktywnić, podczas inicjalizacji wywołujemy

```
glEnable ( GL_DEPTH_TEST );.
```

Przed wyświetlaniem należy ustawić odpowiednie przekształcenia określające rzut przestrzeni trójwymiarowej na ekran, co jest opisane dalej. Rysowanie powinno zacząć się od wyczyszczenia tła i inicjalizacji bufora głębokości. W tym celu należy wywołać

```
glClear ( GL_COLOR_BUFFER_BIT |
          GL_DEPTH_BUFFER_BIT );
```


Następnie wyświetlamy obiekty — to na najniższym poziomie jest wykonywane przez procedury biblioteki GL wywoływane między `glBegin` i `glEnd` (to bywa ukryte w procedurach „wyższego poziomu”, np. rysujących sześcian, sferę itp.). Na zakończenie rysowania obrazka należy wywołać `glFlush ();`, albo, jeśli trzeba, `glFinish ();`. Pierwsza z tych procedur powoduje rozpoczęcie wykonania wszystkich komend GL-a, które mogą czekać w kolejce. Zakończenie tej procedury nie oznacza, że obrazek w oknie jest gotowy. Druga procedura czeka na potwierdzenie zakończenia ostatniej komendy, co może trwać, zwłaszcza w sieci, ale bywa konieczne, np. wtedy, gdy chcemy odczytać obraz z ekranu (w celu zapisania go do pliku, albo utworzenia z niego tekstury).

O.3 Przekształcenia

O.3.1 Macierze przekształceń i ich stosy

OpenGL przetwarza macierze 4×4 i 4×1. Reprezentują one przekształcenia rzutowe lub afiniczne przestrzeni trójwymiarowej i punkty, za pomocą współrzędnych jednorodnych. Macierz 4×4 jest przechowywana w tablicy jednowymiarowej o 16 elementach; tablica ta zawiera kolejne kolumny. Istnieją trzy wewnętrzne stosy, na których są przechowywane macierze przekształceń spełniających ustalone role w działaniu systemu. Operacje na macierzach dotyczą stosu wybranego przez wywołanie procedury `glMatrixMode`, z jednym argumentem, który może być równy

- `GL_MODELVIEW` — przekształcenia opisują przejście między układem, w którym są podawane współrzędne punktów i wektorów, a układem, w którym następuje rzutowanie perspektywiczne lub równoległe. W każdej implementacji OpenGL-a stos ten ma pojemność co najmniej 32.
- `GL_PROJECTION` — przekształcenie reprezentowane przez macierz na tym stosie opisuje rzut perspektywiczny lub równoległy. Ten stos ma pojemność co najmniej 2. Tyle wystarczy, bo w intencji twórców standardu ten stos jest potrzebny tylko w sytuacji awaryjnej — w razie błędu można zachować przekształcenie używane do rzutowania sceny, określić inne, odpowiednie dla potrzeb wyświetlenia komunikatu o błędach, a następnie przywrócić pierwotne przekształcenie podczas likwidacji komunikatu.
- `GL_TEXTURE` — przekształcenia na tym stosie opisują odwzorowanie tekstury. Jego pojemność nie jest mniejsza od 2.

Procedury obsługi macierzy, działające na wierzchołku wybranego stosu:

`void glLoadIdentity ();` — przypisanie macierzy jednostkowej,

`void glLoadMatrix* ( m );` — przypisanie macierzy podanej jako parametr,

`void glMultMatrix* ( m );` — mnożenie przez macierz podaną jako parametr,

`void glTranslate* ( x, y, z );` — mnożenie przez macierz przesunięcia,

`void glRotate* ( a, x, y, z );` — mnożenie przez macierz obrotu o kąt a wokół osi o kierunku wektora $[x, y, z]^T$,

`void glScale* ( x, y, z );` — mnożenie przez macierz skalowania.

W mnożeniu macierzy argument podany w wywołaniu procedury jest z *prawej* strony. Przekształcenia są przez to składane w kolejności odwrotnej do wykonywania obliczeń. Skutek tego jest taki sam, jak w PostScripcie. Jeśli więc mamy procedurę, która rysuje (tj. wprowadza do systemu) jakieś obiekty określone w pewnym układzie współrzędnych, to aby umieścić je w globalnym układzie, należy wywołanie tej procedury poprzedzić wywołaniem procedur opisanych wyżej, które odpowiednio przekształcą te obiekty.

Operacje stosowe są wykonywane przez następujące procedury:

`void glPushMatrix ();` — umieszcza na stosie dodatkową kopię macierzy, która dotychczas była na wierzchołku,

`void glPopMatrix ();` — usuwa element (macierz) z wierzchołka stosu.

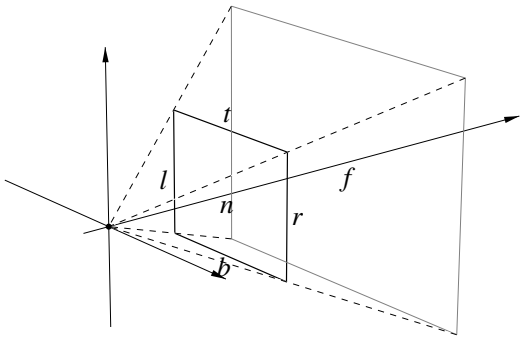
O.3.2 Rzutowanie

Odwzorowanie przestrzeni trójwymiarowej na ekran w OpenGL-u składa się z trzech (a właściwie czterech) kroków. Pierwszym jest przejście do układu współrzędnych (jednorodnych) obserwatora; polega ono na pomnożeniu macierzy współrzędnych jednorodnych punktu przez macierz znajdującą się na wierzchołku stosu `GL_MODELVIEW`.

Drugi krok to rzutowanie. Otrzymana w pierwszym kroku macierz współrzędnych jednorodnych jest mnożona przez macierz znajdującą się na wierzchołku stosu `GL_PROJECTION`, a następnie pierwsze trzy współczynniki iloczynu są dzielone przez czwarty (a więc jest to przejście od współrzędnych jednorodnych do kartezjańskich). Obrazem bryły widzenia po tym kroku jest kostka jednostkowa.

Krok trzeci to odwzorowanie kostki na ekran. Do pierwszych dwóch współrzędnych, pomnożonych odpowiednio przez szerokość i wysokość klatki (w pikselach) są dodawane współrzędne piksela w *dolnym lewym* rogu klatki. Ostatni krok, za który jest odpowiedzialny system okien, to odwzorowanie klatki na ekran, zależne od położenia okna, w którym ma być wyświetlony obraz. Może się to wiązać ze zmianą zwrotu osi y, np. w systemie GLUT. Zauważmy, że klatka nie musi wypełniać całego okna.

Jeśli mamy współrzędne piksela na przykład wskazanego przez kursor, podane przez GLUTa (albo system okien, z którym program pracuje bez pośrednictwa GLUTa, np. XWindow), to należy przeliczyć współrzędną y, z układu określonego przez system okien do układu OpenGL-a. Wystarczy użyć wzoru $y' = h - y - 1$ (h jest wysokością okna w pikselach). Ten sam wzór służy również do konwersji w drugą stronę.



Rysunek O.2. Parametry ostrosłupa widzenia w OpenGL-u

Przykład poniżej przedstawia procedurę `reshape`, przystosowaną do współpracy z aplikacją GLUTa. Procedura ta będzie wywoływana po utworzeniu okna i po każdej zmianie jego wielkości (spowodowanej przez użytkownika, który może sobie okno rozciągać i zmniejszać myszą); jej parametrami są wymiary (wysokość i szerokość) okna w pikselach. Procedura ta umieszcza na stosie `GL_PROJECTION` macierz rzutowania perspektywicznego, skonstruowaną przez procedurę `glFrustum`. Przekształcenie to odwzorowuje bryłę widzenia na sześcian jednostkowy. Wywołanie macierzy `glViewport` określa przekształcenie odpowiedniej ściany tego sześcianu na wskazany prostokąt na ekranie.

```
void reshape ( int w, int h )
{
```

```
  glViewport ( 0, 0, w, h );
  glMatrixMode ( GL_PROJECTION );
  glLoadIdentity ( );
  glFrustum ( -1.0, 1.0, -1.0, 1.0, 1.5, 20.0 );
  glMatrixMode ( GL_MODELVIEW );
  glLoadIdentity ( );
} /*reshape*/
```

Procedura `glViewport` określa trzeci krok rzutowania, tj. przekształcenie kostki jednostkowej w okno. Jej dwa pierwsze parametry określają współrzędne lewego dolnego narożnika w pikselach, w układzie, którego początek znajduje się w lewym dolnym rogu okna. Kolejne dwa parametry to odpowiednio szerokość i wysokość okna w pikselach.

Pierwsze wywołanie procedury `glMatrixMode`, zgodnie z wcześniejszą informacją, wybiera do dalszych działań na przekształceniach stos macierzy rzutowania. Procedura `glLoadIdentity` inicjalizuje macierz na wierzchołku tego stosu; wywołana następnie procedura `glFrustum` oblicza współczynniki macierzy R przekształcenia rzutowego, które opisuje rzutowanie perspektywiczne, i zastępuje macierz na tym stosie przez iloczyn jej i macierzy R.

Parametry procedury `glFrustum` określają kształt i wielkość ostrosłupa widzenia. Znaczenie kolejnych parametrów, l, r, b, t, n, f jest na rysunku. Zwróćmy uwagę, że ostrosłup ten nie musi być symetryczny, a poza tym wymiary jego podstawy nie są skorelowane z wielkością okna, co może prowadzić do zniekształceń (nierównomiernego skalowania obrazu w pionie i poziomie). Dlatego trzeba samemu zadbać o uniknięcie takich zniekształceń; powinno być $(r-l):(t-b) = w:h$, gdzie w, h to wartości parametrów `w` i `h` procedury `glViewport` (w przykładowej procedurze `reshape` podanej wyżej tak nie jest). Dla osób ciekawych, macierz generowana przez procedurę `glFrustum` ma postać

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

Łatwiejsza w użyciu jest procedura `gluPerspective`, która ma 4 parametry: `fovy`, `aspect`, `n` i `f`. Dwa ostatnie są takie jak n i f w `glFrustum`. Parametr `fovy` jest kątem (w stopniach; w OpenGL-u wszystkie kąty mierzy się, niestety, w stopniach) między płaszczyznami górnej i dolnej ściany ostrosłupa, który jest symetryczny. Parametr `aspect` odpowiada proporcjom wymiarów klatki na ekranie; jeśli piksele są kwadratowe (tj. o jednakowej wysokości i szerokości), to `aspect` powinien być równy `w/h`.

Domyślne położenie obserwatora to punkt $[0,0,0]^T$, patrzy on w kierunku osi z, w stronę punktu $[0,0,-1]$ i oś y układu globalnego ma na obrazie kierunek pionowy. Jeśli chcemy umieścić obserwatora w innym punkcie, to możemy wywołać procedurę `glLookAt`. Ma ona 9 parametrów; pierwsze trzy, to współrzędne x, y, z punktu położenia obserwatora. Następne trzy to współrzędne punktu, który znajduje się przed obserwatorem i którego rzut leży na środku obrazu. Ostatnie trzy parametry to współrzędne wektora określającego kierunek „do góry”.

Procedurę `glLookAt`, która wywołuje procedury określające odpowiednie przesunięcia i obroty, wywołuje się na początku procesu ustawiania *obiektów*, który zaczyna się od wywołania `glMatrixMode ( GL_MODELVIEW );` i zaraz potem `glLoadIdentity ( );`.

Aby określić **rzutowanie równoległe**, można wywołać procedurę `glOrtho` lub `gluOrtho2D`. Pierwsza z tych procedur ma 6 parametrów, o podobnym znaczeniu jak `glFrustum`. Bryła widoczności jest prostopadłościanem, o ścianach równoległych do płaszczyzn układu, którego wierzchołkami są punkty $[l,b,n]^T$ i $[r,t,f]^T$. Procedura `gluOrtho2D` ma tylko 4 parametry — domyślnie przyjęte są wartości $n=-1$ i $f=+1$. Macierz rzutowania tworzona przez procedurę `glOrtho` ma postać

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Gdybyśmy chcieli określić rzutowanie tak, aby współrzędne x i y punktów podawane w czasie rysowania były współrzędnymi w oknie, z punktem $[0,0]^T$ w górnym lewym rogu i z osią y skierowaną do dołu, to procedura `reshape` powinna mieć postać

```
void reshape ( int w, int h )
{
  glViewport ( 0, 0, w, h );
  glMatrixMode ( GL_PROJECTION );
  glLoadIdentity ( );
  gluOrtho2D ( 0.0, w, h, 0.0 );
  glMatrixMode ( GL_MODELVIEW );
  glLoadIdentity ();
} /*reshape*/
```

Rzutowanie jest zwykle wykonywane przez sprzęt, ale zdarza się potrzeba obliczenia współrzędnych obrazu danego punktu w przestrzeni, albo przeciwobrazu

punktu na ekranie. Umożliwiają to procedury

```
int gluProject ( x, y, z, mm, pm, vp, wx, wy, wz );
```

oraz

```
int gluUnProject ( wx, wy, wz, mm, pm, vp, x, y, z );
```

Parametry `x`, `y` i `z` określają współrzędne kartezjańskie punktu w przestrzeni. Parametry `wx`, `wy` i `wz` współrzędne „w oknie". Dla punktu położonego między płaszczyznami obcinającymi z przodu i z tyłu jest $0 \leqslant$ `wz` $\leqslant 1$. W wywołaniu `gluUnProject` parametr `wz` jest konieczny, aby wynik był jednoznacznie określony — pamiętamy, że to jest czynność odwrotna do rzutowania, które nie jest przekształceniem różnowartościowym.

Parametry `mm` i `pm` to odpowiednio macierz przekształcenia sceny i rzutowania. Współczynniki tych macierzy można „wyciągnąć" z systemu wywołując

```
glGetDoublev ( GL_MODELVIEW_MATRIX, mm );
glGetDoublev ( GL_PROJECTION_MATRIX, pm );
```

(`mm` i `pm` powinny tu być tablicami liczb typu `double`, o długości 16) natomiast parametr `vp` jest tablicą, która zawiera wymiary okna w pikselach. Można je uzyskać przez wywołanie

```
glGetIntegerv ( GL_VIEWPORT, vp );
```

z parametrem `vp`, który jest tablicą czterech liczb całkowitych — procedura wpisuje do niej parametry ostatniego wywołania procedury `glViewport`.

O.4 Działanie GLUTa

Zadaniem biblioteki GLUT jest ukrycie przed aplikacją wszystkich szczegółów interfejsu programowego systemu okien. W tym celu GLUT definiuje własny interfejs, zaprojektowany w duchu obiektowym. Korzystanie z GLUTa daje tę korzyść, że aplikacja może być przeniesiona do innego systemu (np. z Unixa do OS/2) bez żadnej zmiany kodu źródłowego, a ponadto GLUT jest niezwykle prosty w użyciu. Za tę przyjemność płacimy brakiem dostępu do obiektów zdefiniowanych w systemie, np. XWindow, takich jak boksy dialogowe i wihajstry (guziki, suwaki itp.). Można je utworzyć i obsługiwać wyłącznie za pomocą GLUTa i bibliotek GL i GLU, co jest bardziej pracochłonne.

O.4.1 Schemat aplikacji GLUTa

Aplikacja GLUTa składa się z kilku procedur bezpośrednio współpracujących z systemem. Podczas inicjalizacji (na ogół w procedurze `main`) należy określić

pewne szczegóły korzystania z systemu i zarejestrować procedury obsługi zdarzeń, które będą następnie wywoływane przez system.

Szkielet programu — aplikacji GLUTa jest następujący:

```
...  /* różne dyrektywy #include */
#include <GL/glut.h>

...  /* różne procedury */
void reshape ( int w, int h) { ...  }
void display ( ) { ...  }
void mouse ( int button, int state, int x, int y )
{ ...  }
void motion ( int x, int y ) { ...  }
void keyboard ( unsigned char key, int x, int y )
{ ...  }
void idle ( ) { ...  }
int main ( int argc, char **argv)
{
  glutInit ( & argc, argv );
  glutInitDisplayMode ( ...  );
  glutInitWindowSize ( w, h );
  glutInitWindowPosition ( x, y );
  glutCreateWindow ( argv[0] );
  init ( );   /* tu inicjalizacja danych programu */
  glutDisplayFunc ( display );
  glutReshapeFunc ( reshape );
  glutKeyboardFunc ( keyboard );
  glutMouseFunc ( mouse );
  glutMotionFunc ( mousemove );
  glutIdleFunc ( idle );
  glutMainLoop ( );
  exit ( 0 );
} /*main*/
```

Procedury użyte w powyższym programie wykonują zadania opisane niżej, w kolejności wywoływania.

```
glutInit ( int *argc, char **argv );
```

Procedura `glutInit` dokonuje inicjalizacji biblioteki. Jako parametry są przekazywane parametry wywołania programu, wśród których mogą być opcje dla systemu okien, określające np. terminal, na którym program ma wyświetlać obrazki.

```
glutInitDisplayMode ( unsigned int mode );
```

Ta procedura określa sposób działania GL-a w tej aplikacji, w tym wykorzystywane zasoby. Parametr jest polem bitowym, np.

```
GLUT_SINGLE | GLUT_RGBA | GLUT_DEPTH |
GLUT_STENCIL | GLUT_ACCUM, albo
GLUT_DOUBLE | GLUT_INDEX ...
```

Poszczególne wyrazy są maskami bitowymi. `GLUT_SINGLE` oznacza używanie jednego bufora obrazów, `GLUT_DOUBLE` — dwóch, które są potrzebne do płynnej animacji. `GLUT_DEPTH` deklaruje chęć używania bufora głębokości (do rysowania z uwzględnieniem widoczności). `GLUT_STENCIL` oznacza bufor maski, do wyłączania rysowania w pewnych obszarach okna. `GLUT_ACCUM` oznacza bufor akumulacji, przydatny w antyaliasingu.

`glutInitWindowSize` i `glutInitWindowPosition`

Procedury określają początkowe wymiary i położenie okna (w pikselach, w układzie, w którym (0, 0) jest górnym lewym narożnikiem ekranu). Użytkownik może je zmieniać gdy program już działa.

```
int glutCreateWindow ( char *tytuł );
```

Procedura `glutCreateWindow` tworzy okno (ale nie wyświetla go od razu). Parametr jest napisem, który system okien umieści na ramce, w przykładzie jest to nazwa programu z linii komend.

```
glutDisplayFunc ( void (*func)(void) );
```

Procedura dokonuje rejestracji w GLUCie procedury, która będzie wywoływana za każdym razem, gdy nastąpi konieczność odtworzenia (narysowania) zawartości okna. Może to nastąpić po odsłonięciu fragmentu okna (bo inne okno zostało przesunięte lub zlikwidowane), a także na wniosek aplikacji, który jest zgłaszany przez wywołanie `glutPostRedisplay ();`.

```
glutReshapeFunc ( void (*func)( int w, int h ) );
```

Procedura rejestruje procedurę, która jest odpowiedzialna za przeliczenie macierzy rzutowania, stosownie do wymiarów okna, przekazywanych jako parametry. Przykłady takich procedur były podane wcześniej.

`glutMouseFunc`, `glutMotionFunc` i `glutKeyboard`

Powyższe procedury rejestrują procedury obsługi komunikatów o zdarzeniach spowodowanych przez użytkownika. Są to odpowiednio naciśnięcie lub zwolnienie guzika, przesunięcie myszy i naciśnięcie klawisza na klawiaturze. Obsługując taki komunikat, aplikacja może zmienić dane, a następnie wywołać `glutPostRedisplay ();` w celu spowodowania narysowania nowego obrazka w oknie. Nie powinno tu być bezpośrednich wywołań procedur rysujących.

Jest też procedura `glutIdleFunc` rejestruje procedurę, która będzie wywoływana za każdym razem, gdy komputer nie ma nic innego do roboty. Procedura taka powinna wykonywać krótkie obliczenie (które może być fragmentem długiego obliczenia) i wrócić; obliczenie będzie kontynuowane po następnym wywołaniu.

Procedury rejestrujące mogą być użyte do zmiany lub „wyłączenia" procedury obsługi komunikatu w trakcie działania programu. W tym celu należy wywołać taką procedurę, podając jako parametr nową procedurę lub wskaźnik pusty (`NULL`).

```
glutMainLoop ( void );
```

To jest procedura obsługi pętli komunikatów, która nigdy nie zwraca sterowania (czyli instrukcja `exit ( 0 );` w przykładzie jest wyrazem pewnej przesady). W tej procedurze następuje translacja komunikatów otrzymywanych z systemu (np. XWindow lub innego) na wywołania odpowiednich procedur zarejestrowanych w GLUCie. Zatrzymanie programu następuje przez wywołanie procedury `exit` w ramach obsługi komunikatu, który oznacza, że użytkownik wydał polecenie zatrzymania programu (np. przez naciśnięcie klawisza `<Esc>`).

O.4.2 Przegląd procedur GLUTa

Choć możliwości tworzenia menu udostępniane przez bibliotekę GLUT wydają się skromne, jednak mamy możliwość tworzenia wielu okien, z których każde może mieć inną zawartość, a także podokien, czyli prostokątnych części okien, w których możemy rysować cokolwiek. Dzięki temu do utworzenia wihajstrów obsługujących dialog z użytkownikiem możemy użyć wszystkich możliwości OpenGLa. Opis poniżej jest w zasadzie przewodnikiem po pliku nagłówkowym `glut.h`. Dlatego nie ma w nim zbyt dokładnego przedstawienia list parametrów.

Aby utworzyć **okno**, należy wywołać procedurę `glutCreateWindow`. Jej wartością jest liczba całkowita, która jest identyfikatorem okna w GLUCie (identyfikatory tworzone przez system XWindow lub inny są przed aplikacją GLUTa ukryte). Początkowe wymiary i położenie okna określa się wywołując *wcześniej* procedury `glutInitWindowSize` i `glutInitWindowPosition`.

Aplikacja, która tworzy tylko jedno okno, może zignorować wartość funkcji `glutCreateWindow`. Inne aplikacje powinny ją zapamiętać. Jeśli utworzymy drugie okno, to potrzebujemy móc określić na przykład w którym oknie

chcemy rysować. W danej chwili tylko jedno okno jest aktywne; możemy wywołać procedurę `glutGetWindow` aby otrzymać jego identyfikator. Okno jest aktywne natychmiast po utworzeniu i właśnie aktywnego okna dotyczą wywołania procedur `glutMouseFunc` itd., rejestrujące procedury obsługi komunikatów okna. Okno jest też aktywne w chwili wywołania jego procedury obsługi komunikatu. Jeśli chcemy spowodować odrysowanie zawartości tylko tego okna, to po prostu wywołujemy procedurę `glutPostWindowRedisplay` (wywołanie `glutPostRedisplay` powoduje odrysowanie wszystkich okien). Jeśli chcemy odrysowania innego okna, to powinniśmy wcześniej je uaktywnić, wywołując `glutSetWindow (` *identyfikator_okna* `)`. Podobnie trzeba postąpić, aby w trakcie działania programu zmienić lub zlikwidować procedurę obsługi komunikatu (aby zlikwidować należy przekazać zamiast procedury wskaźnik `NULL`).

Podokno jest prostokątnym fragmentem okna, w którym można rysować niezależnie od tego okna i innych jego podokien. Aby utworzyć podokno, wywołujemy procedurę

```
glutCreateSubWindow ( win, x, y, w, h );
```

Wartością tej procedury jest identyfikator podokna. Identyfikatory okien i podokien tworzą wspólną przestrzeń, tj. identyfikatory wszystkich okien i podokien są różne. W ten sposób procedury `glutSetWindow` i procedury rejestracji procedur obsługi komunikatów działają tak samo na oknach jak i podoknach.

Pierwszym parametrem procedury `glutCreateSubWindow` jest identyfikator okna (lub podokna), którego to jest część. Cztery pozostałe parametry określają położenie i wymiary podokna, względem górnego lewego rogu okna.

Do zlikwidowania okna lub podokna służy procedura `glutDestroyWindow`. Aby zmienić położenie lub wymiary okna lub podokna, należy uczynić je aktywnym (przez wywołanie `glutSetWindow`), a następnie wywołać procedurę `glutPositionWindow` lub `glutReshapeWindow`. Jeśli okno jest podzielone na kilka podokien, to możemy zmieniać ich wymiary w procedurze obsługi komunikatu o zmianie wielkości okna głównego; użytkownik zmienia wielkość okna za pomocą myszy, a procedura ta oblicza wielkości i położenia podokien tak, aby dostosować je do zmienionego okna głównego.

Są jeszcze następujące procedury „zarządzania oknami”:

`glutSetWindowTitle` — ustawia tytuł okna na ramce utworzonej przez system (to chyba nie dotyczy podokien).

`glutSetIconTitle` — okno może być wyświetlone w postaci ikony; procedura określa podpis tej ikony na taką okoliczność.

`glutIconifyWindow` — wyświetla ikonę symbolizującą okno.

`glutHideWindow` i `glutShowWindow` — likwidują i przywracają obraz okna na ekranie.

`glutFullScreen` — po wywołaniu tej procedury aktywne okno zajmuje cały ekran (nie w każdej implementacji GLUTa to jest dostępne).

`glutPopWindow`, `glutPushWindow` — zmieniają kolejność wyświetlania okien, co ma wpływ na to, które jest widoczne, jeśli się nakładają.

Poza tym jest funkcja `glutGet`, która udostępnia różne informacje. Ma ona jeden parametr, któremu możemy nadać następujące wartości:

`GLUT_WINDOW_X`, `GLUT_WINDOW_Y` — wartością funkcji `glutGet` jest odpowiednia współrzędna górnego lewego narożnika okna w układzie okna nadrzędnego (w przypadku okna głównego — na ekranie),

`GLUT_WINDOW_WIDTH`, `GLUT_WINDOW_HEIGHT` — szerokość lub wysokość okna,

`GLUT_WINDOW_PARENT` — identyfikator okna, którego to jest podokno,

`GLUT_WINDOW_NUM_CHILDREN` — liczba podokien,

`GLUT_WINDOW_DOUBLEBUFFER` — informacja, czy jest podwójny bufor obrazu,

`GLUT_BUFFER_SIZE` — liczba bitów reprezentujących kolor piksela,

`GLUT_WINDOW_RGBA` — informacja, czy wartość piksela jest bezpośrednią reprezentacją koloru (jeśli 0, to jest tryb z paletą),

`GLUT_DEPTH_SIZE` — liczba bitów piksela w buforze głębokości,

`GLUT_HAS_KEYBOARD`, `GLUT_HAS_MOUSE`, `GLUT_HAS_SPACEBALL` itp. — informacja o obecności różnych urządzeń

i wiele innych, o które na razie mniejsza.

O.4.3 Współpraca okien z OpenGL-em

Przyznam, że nie jestem pewien, czy GLUT tworzy osobny kontekst OpenGL-a dla każdego okna i podokna (chyba tak; eksperymentatorzy, którzy to wyjaśnią są mile widziani), ale nie jest to takie ważne, jeśli program jest napisany tak, aby wyświetlanie zawartości każdego okna było niezależne od tego, co robiliśmy z OpenGL-em poprzednio. Z moich skromnych doświadczeń wynika, że taki styl pisania programów opłaca się, nawet jeśli wiąże się to ze spadkiem sprawności programu, który za każdym razem ustawia te same parametry. Spadek ten jest zresztą niezauważalny (ale może być inaczej, jeśli korzystamy z rozbudowanych list obrazowych i zwłaszcza tekstur).

Aby to urzeczywistnić, powinniśmy określać parametry rzutowania dla każdego okna w procedurze wyświetlania zawartości okna (tej rejestrowanej przez `glutDisplayFunc`). Wtedy zbędne są procedury obsługi zmiany wielkości okna (rejestrowane przez `glutReshapeFunc`), które informują OpenGL-a o wielkości okna (przez wywołanie `glViewport`) i obliczają macierz rzutowania. Zatem w programie może być tylko jedna procedura obsługi zmiany kształtu okna — ta związana z oknem głównym, bo ona ma zmienić kształt podokien.

O.4.4 Figury geometrczne dostępne w GLUCie

Rysowanie za pomocą ciągów wywołań `glVertex*` między `glBegin` i `glEnd` jest dość uciążliwe, ale to jest zwykła rzecz na niskim poziomie abstrakcji realizowanym w sprzęcie. Również procedury „wyższego poziomu" dostępne w bibliotece GLU są nie najprostsze w użyciu. Natomiast w bibliotece GLUT mamy proste w użyciu procedury

`glutSolidCube`, `glutWireCube` — rysuje sześcian, za pomocą wielokątów albo krawędzi. Parametr określa długość krawędzi. Dowolny prostopadłościan możemy zrobić poddając sześcian odpowiedniemu skalowaniu,

`glutSolidTetrahedron`, `glutWireTetrahedron` — rysuje czworościan (nie ma parametrów),

`glutSolidOctahedron`, `glutWireOctahedron` — rysuje ośmiościan,

`glutSolidDodecahedron`, `glutWireDodecahedron` — rysuje dwunastościan,

`glutSolidIcosahedron`, `glutWireIcosahedron` — rysuje dwudziestościan,

`glutSolidSphere`, `glutWireSphere` — rysuje przybliżenie sfery; kolejne parametry to promień i dwie liczby większe od 2, określające z ilu czworokątów składa się to przybliżenie (wystarczy rzędu kilku do kilkunastu),

`glutSolidTorus`, `glutWireTorus` — rysuje przybliżenie torusa; pierwsze dwa parametry to promień wewnętrzny i zewnętrzny, dwa następne określają dokładność przybliżenia (przez podanie liczby ścian),

`glutSolidCone`, `glutWireCone` — rysuje stożek o promieniu podstawy i wysokości określonych przez pierwsze dwa parametry. Dwa następne określają liczbę ścianek (a zatem dokładność) przybliżenia,

`glutSolidTeapot`, `glutWireTeapot` — rysuje czajnik z Utah, którego wielkość jest określona przez parametr.

Wszystkie powyższe procedury zawierają odpowiednie wywołania `glBegin` i `glEnd`, a także `glNormal` (tylko te ze słowem Solid w nazwie). Oczywiście, nie wystarczą one do narysowania np. sześcianu, którego ściany mają różne kolory (chyba, że na sześcian ten nałożymy teksturę).

O.5 Określanie wyglądu obiektów na obrazie

O.5.1 Oświetlenie

OpenGL umożliwia określenie kilku źródeł światła; mają one wpływ na wygląd rysowanych obiektów na obrazie. Oprócz oświetlenia, na wygląd obiektów wpływają własności materiału, z którego są „zrobione" obiekty, tekstura, a także ustawienia różnych parametrów OpenGL-a. Zacznijmy od opisu sposobu określania źródeł światła.

Źródła te są punktowe. Każda implementacja OpenGL-a obsługuje co najmniej 8 źródeł światła, są one identyfikowane przez stałe symboliczne `GL_LIGHT0 ... GL_LIGHT7`. Przykład określenia własności źródła światła:

```
glLightfv ( GL_LIGHT0, GL_AMBIENT, amb0 );
glLightfv ( GL_LIGHT0, GL_DIFFUSE, diff0 );
glLightfv ( GL_LIGHT0, GL_SPECULAR, spec0 );
glLightfv ( GL_LIGHT0, GL_POSITION, pos0 );
glLightf ( GL_LIGHT0, GL_CONSTANT_ATTENUATION,
           catt0 );
glLightf ( GL_LIGHT0, GL_LINEAR_ATTENUATION,
           latt0 );
glLightf ( GL_LIGHT0, GL_QUADRATIC_ATTENUATION,
           qatt0 );
glLightf ( GL_LIGHT0, GL_SPOT_CUTOFF, spco0 );
glLightfv ( GL_LIGHT0, GL_SPOT_DIRECTION, spdir0 );
glLightfv ( GL_LIGHT0, GL_SPOT_EXPONENT, spexp0 );
glEnable ( GL_LIGHT0 );
```

Pora na wyjaśnienie. Mamy tu ciąg wywołań procedur określających własności źródła światła 0, a na końcu wywołanie procedury `glEnable`, które ma na celu „włączenie" tego światła. Procedura `glLightf` określa własność źródła opisaną przez *jeden* parametr, natomiast procedura `glLightfv` otrzymuje tablicę zawierającą *cztery* liczby typu `GLfloat`. Pierwszy parametr każdej z tych procedur określa, którego źródła światła dotyczy wywołanie. Drugi parametr określa, jaką własność zmienia to wywołanie. Kolejno są to:

GL_AMBIENT — kolor światła „rozproszonego” w otoczeniu (niezależnie od położenia źródła światła). Cztery elementy tablicy amb0 to liczby od 0 do 1, opisujące składowe czerwoną, zieloną i niebieską, oraz współczynnik alfa, który ma znaczenie tylko w pewnych trybach obliczania koloru pikseli, o których tu nie piszę. Domyślnie (czyli jeśli nie wywołamy glLightfv z drugim parametrem równym GL_AMBIENT), składowe koloru światła rozproszonego mają wartości 0.0, 0.0, 0.0, 1.0.

GL_DIFFUSE — kolor światła, które dochodząc do punktu powierzchni od źródła światła podlega odbiciu rozproszonemu (tzw. lambertowskiemu). Jeśli nie ma innych składowych światła, to obiekty pokolorowane na podstawie takiego oświetlenia są idealnie matowe. Domyślny kolor tego składnika oświetlenia dla źródła GL_LIGHT0 ma składowe 1.0, 1.0, 1.0, 1.0, czyli jest to światło białe o maksymalnej intensywności, pozostałe źródła światła mają cztery zera.

GL_SPECULAR — kolor światła, które podlega odbiciu zwierciadlanemu (własności tego lustra są opisane dla rysowanych obiektów). W zwykłych sytuacjach składowe tego składnika światła powinny być takie same jak światła podlegającego odbiciu rozproszonemu i takie są domyślne wartości.

GL_POSITION — trzeci parametr procedury glLightfv określa współrzędne położenia źródła światła. To są współrzędne jednorodne; jeśli ostatnia z nich jest równa 0, to źródło światła jest położone w odległości nieskończonej, w kierunku określonym przez pierwsze trzy współrzędne. W przeciwnym razie punkt położenia źródła światła znajduje się w skończonej odległości, może być nawet między obiektami w scenie. Domyślnie współrzędne położenia źródła światła są równe 0.0, 0.0, 1.0, 0.0.

GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION — trzy parametry, k_c, k_l i k_q, określane przez wywołania glLightf z tymi argumentami określają, w jaki sposób intensywność światła maleje z odległością od niego. Współczynnik osłabienia światła jest obliczany ze wzoru

$$a = \frac{1}{k_c + k_l d + k_q d^2},$$

w którym d oznacza odległość źródła światła od oświetlanego punktu. Domyślnie jest $k_c = 1.0$, $k_l = k_q = 0.0$, co jest odpowiednie dla źródeł światła bardzo odległych od sceny. Zmienianie tych parametrów może spowodować nieco wolniejsze rysowanie, ale jak trzeba, to trzeba.

GL_SPOT_DIRECTION, GL_SPOT_CUTOFF, GL_SPOT_EXPONENT — parametry określane za pomocą tych argumentów opisują źródła światła o charakterze reflektora. Podany jest kierunek osi reflektora (domyślnie 0.0, 0.0, −1.0), kąt rozwarcia stożka, w jakim rozchodzi się światło (domyślnie 180°, co oznacza rozchodzenie się światła w całej przestrzeni) i wykładnik (domyślnie 0), którego większa wartość oznacza większe osłabienie światła w pobliżu brzegu stożka.

Wektory współrzędnych opisujących położenie źródeł światła lub kierunek osi reflektora są poddawane przekształceniu opisanemu przez bieżącą macierz na stosie GL_MODELVIEW. Rozważmy następujące możliwości:

- Aby położenie źródła światła było **ustalone względem całej sceny**, należy je określić po ustawieniu położenia obserwatora (czyli np. po wywołaniu procedury gluLookAt).
- Aby źródło światła było **ustalone względem obserwatora** (który snuje się po scenie ze świeczką i w szlafmycy), parametry położenia źródła światła należy określić po ustawieniu na wierzchołku stosu macierzy jednostkowej, przed wywołaniem gluLookAt.
- Aby **związać źródło światła z dowolnym obiektem** w scenie, trzeba położenie źródła światła określić po ustawieniu macierzy przekształcenia, która będzie ustawiona w czasie rysowania tego przedmiotu. Ponieważ źródło to ma oświetlać także wszystkie inne przedmioty, być może rysowane wcześniej niż przedmiot względem którego pozycjonujemy źródło światła (i możemy mieć wtedy inne ustawione przekształcenie), więc powoduje to konieczność obliczenia i umieszczenia na stosie przekształcenia właściwego, co niekoniecznie jest trywialne.

Jeszcze jedno: światła włączamy i wyłączamy indywidualnie, wywołując np. glEnable (GL_LIGHT0); lub glDisable (GL_LIGHT1);. Aby jednak światła były w ogóle brane pod uwagę podczas rysowania, trzeba wywołać glEnable (GL_LIGHTING);.

O.5.2 Własności powierzchni obiektów

Teraz zajmiemy się określaniem własności powierzchni, wpływającymi na jej kolor na obrazie, w oświetleniu określonym w sposób opisany przed chwilą. Własności te określa się za pomocą procedur glMaterialf i glMaterialfv, które mają trzy parametry.

Pierwszy z nich może przyjmować wartości GL_FRONT, GL_BACK albo też GL_FRONT_AND_BACK i oznacza stronę (albo strony) powierzchni, której dotyczy podana wartość parametru.

Drugi parametr określa własność materiału. Może on być równy

GL_AMBIENT — trzeci parametr procedury glMaterialfv jest tablicą zawierającą cztery liczby od 0.0 do 1.0. Przez te liczby są mnożone składowe czerwona, zielona, niebieska i alfa światła rozproszonego związanego z każdym źródłem i to jest składnikiem ostatecznego koloru piksela. Domyślnie parametry te mają wartości 0.2, 0.2, 0.2 i 1.0, co oznacza, że obiekt jest ciemnoszary (jak o zmierzchu wszystkie koty ...).

GL_DIFFUSE — cztery liczby opisujące zdolność powierzchni do odbijania w sposób rozproszony światła dochodzącego ze źródła światła. W obliczeniu koloru jest uwzględniane jego osłabienie związane z odległością i orientacja powierzchni (kąt między kierunkiem padania światła a wektorem normalnym powierzchni). Aby poprawnie ją uwzględnić, każde wywołanie glVertex* należy poprzedzić wywołaniem glNormal* z odpowiednim wektorem jednostkowym podanym jako parametr. Domyślnie mamy składowe 0.8, 0.8, 0.8, 1.0.

GL_AMBIENT_AND_DIFFUSE — można jednocześnie określić parametry odbicia rozproszonego światła rozproszonego w otoczeniu i światła dochodzącego z konkretnego kierunku.

GL_SPECULAR — cztery liczby opisujące sposób odbicia zwierciadlanego, domyślnie 0.0, 0.0, 0.0, 1.0. O ile kolor obiektu jest widoczny w świetle odbitym w sposób rozproszony, to kolor światła z „zajączków" jest bliski koloru światła padającego. Dlatego składowe czerwona, zielona i niebieska powinny mieć takie same wartości w tym przypadku.

GL_SHININESS — to jest drugi parametr procedury glMaterialf. Oznacza on określanie wykładnika w tzw. modelu Phonga odbicia zwierciadlanego. Trzeci parametr jest liczbą rzeczywistą, domyślnie 0.0. Im jest większy, tym lepsze lustro, w praktyce można stosować wartości od kilku do kilkuset.

GL_EMISSION — cztery składowe światła emitowanego przez powierzchnię (niezależnego od jej oświetlenia), domyślnie 0.0, 0.0, 0.0, 1.0. Światło to nie ma, niestety, wpływu na wygląd innych powierzchni sceny.

Własności materiału na ogół określa się podczas rysowania, tj. bezpośrednio przed narysowaniem obiektu, albo nawet przed wyprowadzeniem każdego wierzchołka (między glBegin (...); i glEnd ( );). Proces ten może więc zabierać dużo czasu. Należy pamiętać, że nie trzeba za każdym razem specyfikować wszystkich własności materiału, wystarczy tylko te, które są inne od domyślnych lub ustawionych ostatnio. Inny sposób przyspieszenia tego procesu polega na użyciu procedury glColorMaterial. Procedura ta ma dwa parametry, identyczne jak procedura glMaterialfv. Po jej wywołaniu kolejne wywołania glColor* mają taki skutek, jak określanie parametrów materiału (czyli kolor nie

jest bezpośrednio nadawany pikselom, tylko używany do określenia koloru pikseli z uwzględnieniem oświetlenia). Rysowanie w tym trybie należy poprzedzić wywołaniem glEnable (GL_COLOR_MATERIAL); i zakończyć wywołaniem glDisable (GL_COLOR_MATERIAL);.

O.5.3 Powierzchnie przezroczyste

Pierwsze 3 współrzędne koloru (podawane na przykład jako parametry procedury glColor*) opisują składowe R, G, B (tj. czerwoną, zieloną i niebieską). Czwarta współrzędna, A (alfa), opisuje „przezroczystość". Podczas wyświetlania pikseli obliczany jest kolor (np. na podstawie oświetlenia i własności materiału), który następnie służy do wyznaczenia ostatecznego koloru przypisywanego pikselowi na podstawie *poprzedniego* koloru piksela i koloru nowego. Dzięki temu wyświetlany obiekt może wyglądać jak częściowo przezroczysty. Opisane obliczenie koloru pikseli nazywa się **mieszaniem** (ang. *blending*) i odbywa się po włączeniu go. Do włączania i wyłączania mieszania służą procedury glEnable i glDisable, wywoływane z parametrem GL_BLEND.

Niech R_s, G_s, B_s i A_s oznaczają nowy kolor, zaś R_d, G_d, B_d i A_d poprzedni kolor piksela. Kolor, który zostanie pikselowi przypisany, będzie miał składowe $R = s_r R_s + d_r R_d$, $G = s_g G_s + d_g G_d$, $B = s_b B_s + d_b B_d$, $A = s_a A_s + d_a A_d$, gdzie współczynniki $s_r, \ldots, d_a$ są ustalane wcześniej.

Do ustalania współczynników mieszania służy procedura glBlendFunc, która ma 2 parametry. Pierwszy określa współczynniki $s_r, \ldots, s_a$, a drugi współczynniki $d_r, \ldots, d_a$. Dopuszczalne wartości tych parametrów są m.in. takie (poniższa lista nie jest pełna):

GL_ZERO	0, 0, 0, 0	
GL_ONE	1, 1, 1, 1	
GL_DST_COLOR	R_d, G_d, B_d, A_d	(tylko dla nowego koloru)
GL_SRC_COLOR	R_s, G_s, B_s, A_d	(tylko dla poprzedniego koloru)
GL_SRC_ALPHA	A_s, A_s, A_s, A_s	
GL_DST_ALPHA	A_d, A_d, A_d, A_d	

O.5.4 Mgła

Wpływ mgły na barwę rysowanych obiektów zależy od odległości obiektu od obserwatora. Aby określić ten wpływ wywołujemy procedury (przykładowe wartości parametrów mogą być punktem wyjścia do eksperymentów)

```
GLfloat fogcolor = { 0.5, 0.5, 0.5, 1.0 };
glEnable ( GL_FOG );
glFogi ( GL_FOG_MODE, GL_EXP );
```

```
glFogfv ( GL_FOG_COLOR, fogcolor );
glFogf ( GL_FOG_DENSITY, 0.35 );
glClearColor ( 0.5, 0.5, 0.5, 1.0 );
```

W tym przykładzie wpływ mgły na barwę zależy w wykładniczy (GL_EXP) sposób od odległości punktu od obserwatora. Warto zwrócić uwagę, że tło obrazu powinno być wypełnione kolorem mgły *przed* rysowaniem obiektów na tym tle.

O.6 Ewaluatory

O.6.1 GL — krzywe i powierzchnie Béziera

Ewaluatory w OpenGL-u to są procedury (zawarte w bibliotece GL, a zatem mogą one być zrealizowane w sprzęcie) obliczające punkt na krzywej lub powierzchni Béziera. Jak łatwo się domyślić, służą one do rysowania krzywych i powierzchni, ale nie tylko. Mogą one służyć do obliczania współrzędnych tekstury i koloru. Niestety, nie znalazłem możliwości obliczenia współrzędnych punktu i przypisania ich zmiennym w programie, a szkoda. Ewaluatory są jednowymiarowe (odpowiada to krzywym) lub dwuwymiarowe (to dotyczy powierzchni). Aby użyć ewaluatora należy go najpierw określić i uaktywnić.

Określenie ewaluatora **jednowymiarowego** polega na wywołaniu np. procedury

```
glMap1f ( GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, p );
```

Pierwszy parametr o wartości GL_MAP1_VERTEX_3 oznacza, że punkty kontrolne krzywej mają trzy współrzędne. Inne możliwe wartości tego parametru to

GL_MAP1_VERTEX_4 — punkty mają cztery współrzędne (jednorodne). Dzięki temu można rysować tzw. krzywe wymierne, o których na wykładzie nie mówiłem, a które są bardzo pożyteczne.

GL_MAP1_COLOR_4 — punkty mają cztery współrzędne koloru, R, G, B, A. Ten ewaluator służy do obliczania koloru, a nie punktów w przestrzeni.

GL_MAP1_NORMAL — ewaluator służy do obliczania wektora normalnego.

GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, — ewaluator służy do obliczania jednej, dwóch, trzech lub czterech współrzędnych tekstury.

Drugi i trzeci parametr określają przedział zmienności parametru — typowe wartości to 0.0 i 1.0, przyjmowane w podstawowym sposobie określenia krzywej Béziera. Kolejny parametr określa liczbę współrzędnych każdego punktu w tablicy

$\boldsymbol{p}$, przekazanej jako ostatni parametr. Może być tak, że w tablicy punkty mają więcej współrzędnych niż chcemy uwzględnić (bo na przykład pakujemy obok siebie współrzędne punktu w przestrzeni, a zaraz potem współrzędne koloru i tekstury, które trzeba pomijać). Kolejny parametr, w tym przykładzie 4, to **rząd** krzywej, czyli liczba punktów kontrolnych (o jeden większa niż stopień). Ostatni parametr to tablica punktów kontrolnych.

Uaktywnienie ewaluatora odbywa się przez wywołanie procedury glEnable, z parametrem takim, jak pierwszy parametr wywołania procedury glMap1f. Ten sam parametr przekazujemy procedurze glDisable aby wyłączyć dany ewaluator. Można określić i uaktywnić jednocześnie kilka ewaluatorów, po to, aby jednocześnie określać punkty krzywej i ich kolory. Użycie ewaluatorów polega na wywołaniu glEvalCoord1f (t);, gdzie t jest liczbą — parametrem krzywej. Jeśli w chwili wywołania są aktywne ewaluatory GL_MAP1_VERTEX_3 i GL_MAP1_COLOR_4, to takie wywołanie jest prawie równoważne wywołaniu glColor4f (...); i glVertex3f (...);, z parametrami o wartościach odpowiednich współrzędnych obliczonych przez te ewaluatory. Różnica polega na tym, że bieżący kolor nie ulega zmianie, tj. kolor obliczony przez ewaluator jest nadawany tylko obliczonemu przez ewaluator punktowi.

Jeśli chcemy narysować ciąg punktów albo łamaną, przy czym punkty te są obliczane przez ewaluator dla argumentów (parametrów krzywej), które tworzą ciąg arytmetyczny, to możemy to zrobić wywołując kolejno:

```
glMapGrid1f ( n, t0, t1 );
glEvalMesh1 ( GL_LINE, i0, i1 );
```

Parametr n jest liczbą kroków (odcinków całej łamanej); parametry t_0 i t_1 określają końce przedziału zmienności parametru krzywej, który zostanie podzielony na n równych części. Parametr GL_LINE oznacza, że rysujemy łamaną (aby narysować punkty trzeba podać GL_POINT). Parametry i_0 i i_1 określają numer pierwszego i ostatniego punktu siatki określonej przez glMapGrid1f, które będą obliczone i narysowane.

Ewaluatory **dwuwymiarowe** działają na takiej samej zasadzie. Zamiast znaków MAP1 w odpowiednich stałych symbolicznych pojawiają się znaki MAP2. Do określania ewaluatora np. dla płata Béziera stopnia (n, m) wywołujemy procedurę

```
glMap2f ( GL_MAP2_VERTEX_3, u0, u1, us, n+1,
          v0, v1, vs, m+1, p );
```

Parametry u_0, u_1, v_0, v_1 określają przedziały zmienności parametrów odpowiednio u i v. Parametry u_s i v_s określają odległości w tablicy (liczb zmiennopozycyjnych, typu GLfloat w tym przypadku) między współrzędnymi kolejnych punktów w wierszu i w kolumnie siatki kontrolnej, a zamiast stopnia ze względu na u

i v podaje się rząd. Aby użyć ewaluatora dwuwymiarowego należy go uaktywnić i można wywołać procedurę `glEvalCoord2f ( u, v );`. Są też dostępne procedury `glMapGrid2f` i `glEvalMesh2`, które pomagają w narysowaniu powierzchni w postaci siatki odcinków lub trójkątów, dla siatki regularnej określonej w dziedzinie płata.

O.6.2 GLU — krzywe i powierzchnie B-sklejane

Rysowanie krzywych i powierzchni B-sklejanych w OpenGL-u jest zrealizowane na dwóch poziomach: poziom „niższy" to opisane wcześniej ewaluatory, zdefiniowane w bibliotece GL, natomiast poziom „wyższy" jest określony w procedurach bibliotece GLU. Procedury te obliczają punkty krzywych i powierzchni za pośrednictwem ewaluatorów, po wyznaczeniu reprezentacji Béziera odpowiedniego fragmentu wielomianowego łuku lub powierzchni.

Aby użyć procedur obsługi krzywych i powierzchni sklejanych z biblioteki GLU, trzeba utworzyć obiekt dokonujący podziału krzywej lub powierzchni na kawałki wielomianowe. Robi się to tak:

```
GLUnurbsObj *nurbs_obj;
...
nurbs_obj = gluNewNurbsRenderer ();
```

Następnym krokiem jest określenie własności tego obiektu, czyli szczegółów jego działania. Służy do tego procedura `gluNurbsProperty`, która ma trzy parametry. Pierwszym z nich jest wskaźnik obiektu (w powyższym przykładzie zmienna `nurbs_obj`). Drugi parametr określa własność, którą specyfikujemy za pomocą trzeciego parametru, który jest liczbą rzeczywistą. Drugi parametr może być równy

`GLU_DISPLAY_MODE` — wtedy trzeci parametr równy `GLU_FILL` powoduje wypełnianie wielokątów, które stanowią przybliżenie powierzchni (można wtedy uaktywnić testy widoczności i „włączyć" oświetlenie). Jeśli trzeci parametr ma wartość `GLU_OUTLINE_POLYGON`, to narysowana będzie siatka odcinków przybliżających linie stałego parametru płata.

`GLU_SAMPLING_TOLERANCE` — trzeci parametr określa długość najdłuższego odcinka (na obrazie, w pikselach, domyślnie 50.0, czyli dużo), jaki może być wygenerowany w celu utworzenia obrazu.

`GLU_SAMPLING_METHOD` — wywołanie procedury z trzecim parametrem równym `GLU_PATH_LENGTH`, powoduje takie dobranie gęstości punktów, aby wielokąty przybliżające powierzchnię miały na obrazie boki nie dłuższe niż tolerancja zadana przez wywołanie procedury `gluNurbsProperty` z drugim parametrem równym `GLU_SAMPLING_TOLERANCE`.

Jeśli trzeci parametr jest równy `GLU_DOMAIN_DISTANCE`, to wywołując następnie procedurę `gluNurbsProperty` z drugim parametrem równym kolejno `GLU_U_STEP` i `GLU_V_STEP` należy podać kroki, z jakimi ma być stablicowana powierzchnia, w dziedzinie.

Obiekt przetwarzający krzywe i powierzchnie NURBS można zlikwidować wywołując `gluDeleteNurbsRenderer` (z parametrem — wskaźnikiem podanym wcześniej przez `gluNewNurbsRendeder`).

Aby narysować powierzchnię, należy ustawić oświetlenie i właściwości materiału, a następnie wywołać procedury

```
gluBeginSurface ( nurbs_obj );
gluNurbsSurface ( nurbs_obj, N+1, u, M+1, v,
                  dpu, dpv, d, n+1, m+1,
                  GL_MAP2_VERTEX_3 );
gluEndSurface ( nurbs_obj );
```

Parametry procedury `gluNurbsSurface` to kolejno wskaźnik obiektu przetwarzającego powierzchnię, liczba i tablica węzłów w ciągu „u", liczba i tablica węzłów w ciągu „v" (oznaczenia są takie jak w wykładzie), odległości `dpu` i `dpv` między pierwszą współrzędną punktów kontrolnych odpowiednio w wierszu i kolumnie siatki (porównaj z opisem ewaluatorów), tablica punktów kontrolnych, rząd ze względu na u i v (o 1 większy niż stopień). Ostatni parametr, określa wymiar przestrzeni (czyli liczbę współrzędnych punktów kontrolnych), w tym przykładzie 3 (rysujemy więc „zwykłą" powierzchnię B-sklejaną). Można też podać ostatni parametr równy `GL_MAP2_VERTEX_4`, który oznacza rysowanie powierzchni wymiernej (punkty kontrolne leżą wtedy w czterowymiarowej przestrzeni jednorodnej), a także `GL_MAP2_TEXTURE_COORD_*` (zamiast `*` musi być `1`, `2`, `3` lub `4`), co oznacza, że ewaluatory wywoływane przez `gluNurbsSurface` mają generować współrzędne w układzie tekstury, albo `GL_MAP2_NORMAL`, w celu wygenerowania wektorów normalnych powierzchni.

Rysując krzywą NURBS, mamy do dyspozycji procedury `gluBeginCurve`, `gluEndCurve` (mają one jeden parametr, wskaźnik obiektu przetwarzania krzywych, utworzonego przez wywołanie `gluNewNurbsRenderer`) i procedurę `gluNurbsCurve`, której parametrami są: wskaźnik obiektu, liczba i tablica węzłów, odstęp (w tablicy liczb rzeczywistych) między pierwszymi współrzędnymi kolejnych punktów kontrolnych, tablicę punktów kontrolnych i parametr określający typ ewaluatora jednowymiarowego, np. `GL_MAP1_VERTEX_3`.

O.7 Bufor akumulacji i jego zastosowania

Bufor akumulacji jest tablicą pikseli, dzięki której jest możliwy antyaliasing (przestrzenny i czasowy) oraz symulacja głębi ostrości. Sposób jego użycia jest następujący: wykonujemy kolejno kilka obrazów sceny, zaburzając dla każdego z nich położenie obserwatora i rzutni (dzięki czemu możemy osiągnąć antyaliasing przestrzenny i symulację głębi ostrości), oraz umieszczając poruszające się obiekty w położeniach odpowiadających różnym chwilom. Obrazy otrzymane w buforze ekranu (tym, który możemy wyświetlać na ekranie) sumujemy w buforze akumulacji. Dokładniej, wartości R, G, B, A każdego piksela obrazu mnożymy przez $\frac{1}{n}$, gdzie n jest liczbą „akumulowanych" obrazów, i dodajemy do odpowiednich składowych (o początkowej wartości 0) odpowiedniego piksela w buforze akumulacji. W ten sposób po wykonaniu n obrazów mamy w buforze akumulacji ich średnią arytmetyczną.

Jeszcze jedna możliwość zastosowania bufora głębokości wiąże się z symulacją oświetlenia sceny przez nie-punktowe źródła światła. Mając źródła światła „liniowe" (np. świetlówki) lub „powierzchniowe" (takie jak lampy z dużym kloszem) możemy wybrać na każdym takim świecącym przedmiocie kilka punktów i na kolejnych obrazach zbieranych w buforze akumulacji uwidoczniać skutek oświetlenia przez źródła światła w tych punktach. Jeśli wyznaczymy za każdym razem cienie (co nie jest łatwe, ale możliwe przez odpowiednie wykorzystanie tekstur), to otrzymamy również „miękkie cienie", jakie powinny wystąpić w tak oświetlonej scenie.

O.7.1 Obsługa bufora akumulacji

Aby skorzystać z bufora akumulacji, należy najpierw go zarezerwować. W GLUCie robi się to wywołując

```
glutInitDisplayMode ( GLUT_RGBA | GLUT_DEPTH |
                      GLUT_ACCUM );
```

Przed rysowaniem pierwszego obrazka (uwaga: pierwszego z serii, która ma dać jeden obraz antyaliasowany) czyścimy bufor akumulacji wywołując

```
glClear ( GL_ACCUM_BUFFER_BIT );
```

Następnie, przed rysowaniem każdego kolejnego obrazka czyścimy ekran i z-bufor:

```
glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

a po narysowaniu go dodajemy wartości pikseli do bufora akumulacji:

```
glAccum ( op, value );
```


Pierwszy parametr powyższej procedury jest kodem operacji, w tym przypadku `GL_ACCUM`, a drugi to mnożnik, który powinien być równy $\frac{1}{n}$, jeśli chcemy zebrać dane z n obrazów. Zamiast kasować bufor akumulacji, możemy za pierwszym razem wywołać `glAccum` z pierwszym parametrem równym `GL_LOAD`. Są też operacje `GL_ADD` i `GL_MULT`, z których pierwsza dodaje *value* do pikseli bufora akumulacji, a druga mnoży ich wartości przez *value* (mnożnik jest obcinany do przedziału $[-1, 1]$). Aby skopiować zawartość bufora akumulacji do bufora obrazu (którego zawartość możemy oglądać na ekranie), wywołujemy

```
glAccum ( GL_RETURN, value );
```

Parametr *value* powinien mieć wartość 1, ponieważ do bufora obrazu wpisywane są wartości z bufora akumulacji pomnożone przez ten parametr. W zasadzie można by, zbierając informację w buforze akumulacji, podać mnożnik (parametr *value*) $\frac{a}{n}$ dla dowolnego $a \neq 0$, a podczas przepisywania do bufora obrazu podać $value = \frac{1}{a}$, ale dla $a > 1$ może nastąpić nadmiar, a dla $a < 1$ rosną błędy zaokrągleń (pamiętajmy, że w buforze akumulacji wartości R, G, B, A są prawdopodobnie reprezentowane przez bajty). Ponieważ jednak parametr *value* za każdym razem podajemy na nowo, więc zamiast średniej arytmetycznej możemy w buforze akumulacji obliczyć średnią ważoną obrazów (suma parametrów *value* musi być równa 1).

O.7.2 Antyaliasing przestrzenny

Przykłady użycia bufora akumulacji, zaczerpnięte z książki, są podane w katalogu `book` w dystrybucji Mesy. Dla wygody zostały określone procedury `accFrustum` i `accPerspective`, które odpowiadają procedurom bibliotecznym `glFrustum` i `gluPerspective`, ale mają dodatkowe parametry, określające zaburzenia położenia obserwatora i klatki na rzutni. Przyjrzymy się tym procedurom.

```
void accFrustum ( left, right, bottom, top,
                  near, far, pixdx, pixdy,
                  eyedx, eyedy, focus );
```

Pierwsze 6 parametrów jest identyczne jak w `glFrustum`. Parametry `pixdx` i `pixdy` określają przesunięcie klatki na rzutni, w pikselach. Parametry `eyedx` i `eyedy` określają przesunięcie obserwatora (środka rzutowania) równolegle do rzutni. Parametr `focus` określa odległość, w której położone punkty mają ostry obraz (o tym mowa dalej, w symulacji głębi ostrości).

Procedura `accFrustum` oblicza liczby

```
dx = -(pixdx*(right-left))/viewport[2] +
```

```
                eyedx*near/focus;
  dy = -(pixdy*(top-bottom))/viewport[3] +
                eyedy*near/focus;
```

(w zmiennych `viewport[2]` i `viewport[3]` są wymiary klatki w pikselach), a następnie wywołuje procedury

```
glMatrixMode ( GL_PROJECTION );
glLoadIdentity ();
glFrustum ( left+dx, right+dx, bottom+dy, top+dy,
            near, far );
glMatrixMode ( GL_MODELVIEW );
glLoadIdentity ();
glTranslatef ( -eyedx, -eyedy, 0.0 );
```

Procedura `accPerspective` oblicza parametry `left`, `right`, `bottom` i `top` na podstawie swoich pierwszych czterech parametrów (takich jak w procedurze `gluPerspective`) i wywołuje `accFrustum`.

Załóżmy na razie, że `eyedx` $=$ `eyedy` $= 0$. Rzuty wszystkich punktów będą przesunięte o `pixdx` pikseli w prawo i o `pixdy` pikseli do góry. Wykonując kolejne obrazki wywołamy procedurę `accPerspective` lub `accFrustum`, podając za każdym razem inne przesunięcia. Nie powinny one wyznaczać regularnej siatki podpikseli (np. dla 4 lub 9 próbek nie powinny one leżeć w środkach kwadracików o boku $\frac{1}{2}$ lub $\frac{1}{3}$). Zamiast tego można je pozaburzać, dodając do każdego przesunięcia na takiej regularnej siatce losowy przyrost o współrzędnych mniejszych niż $\frac{1}{4}$ albo $\frac{1}{6}$.

Przykład jest w programie `book/accpersp.c`.

O.7.3 Symulacja głębi ostrości

Aby otrzymać obraz z głębią ostrości, można wykonać kilka obrazów sceny, zbierając je w buforze akumulacji i podając parametry `eyedx` i `eyedy` procedury `accFrustum` lub `accPerspective`, określające za każdym razem inne przesunięcie obserwatora. Parametr `focus` określa odległość ostrego planu (w jednostkach osi globalnego układu współrzędnych). Parametry `eyedx` i `eyedy` mogą być współrzędnymi punktów zaburzonej regularnej siatki (z dodanym jitterem, tak jak w poprzednim punkcie), trzeba tylko określić mnożnik, który reprezentuje wielkość otworu przysłony (im mniejszy tym mniejsze przesunięcia środka rzutowania, a więc większa głębia ostrości).

Przykład osiągnięcia głębi ostrości jest w programie `book/dof.c`.

O.8 Nakładanie tekstury

Liczba różnych efektów możliwych do osiągnięcia przez nałożenie tekstury na rysowane przedmioty jest trudna do oszacowania. Najprostsze zastosowanie to „pokolorowanie" przedmiotu, którego poszczególne punkty mogą mieć różne własności odbijania światła, przez co na powierzchni tworzy się pewien obraz. Tekstura w OpenGL-u jest jedno-, dwu- albo (nie w każdej implementacji) trójwymiarową tablicą pikseli. Tablica taka może przedstawiać dowolny obraz, np. fotografię, albo obraz wygenerowany przez komputer.

O.8.1 Tekstury dwuwymiarowe

Aby nałożyć na obiekt teksturę, trzeba ją najpierw utworzyć. W tym celu przygotowujemy tablicę tekseli z odpowiednią zawartością. Zaczniemy od tekstury dwuwymiarowej, którą może być obrazek przeczytany z pliku, albo utworzony w dowolny inny sposób.

Wymiary (szerokość i wysokość) tablicy tekseli muszą być równe 2^k, dla $k \geqslant 6$. Może też być $2^k + 2$, co oznacza, że określamy teksturę na całej płaszczyźnie — pierwszy i ostatni wiersz lub kolumna tekseli może być powielona. W specyfikacji OpenGL 2.0 dopuszczalne są też inne wymiary tekstur, natomiast w razie konieczności, jeśli tablica pikseli, którą dysponujemy, ma inne wymiary, to możemy użyć procedury

```
gluScaleImage ( format, inw, inh, intype, indata,
                outw, outh, outtype, outdata );
```

Parametr `format` określa zawartość tablicy, np. `GL_RGB`. Parametry `inw` i `inh` to wymiary (szerokość i wysokość) tablicy wejściowej. Parametr `intype` określa typ elementów, na przykład `GL_UNSIGNED_BYTE` (w połączeniu z formatem `GL_RGB` oznacza to, że każdy teksel jest reprezentowany przez kolejne 3 bajty, określające składowe czerwoną, zieloną i niebieską). Parametr `indata` jest wskaźnikiem tablicy z danymi wejściowymi. Parametry `outw` i `outh` określają wymiary tablicy docelowej. Parametr `outtype` może mieć też wartość `GL_UNSIGNED_BYTE`, a `outdata` jest wskaźnikiem tablicy, w której ma się znaleźć wynik. Tablicę taką o właściwej wielkości należy utworzyć przed wywołaniem tej procedury. Jej wartość 0 oznacza sukces, a 1 błąd.

Aby przygotować teksturę do nałożenia na powierzchnię, trzeba kolejno wykonać instrukcje:

```
glPixelStorei ( GL_UNPACK_ALIGNMENT, 1 );
glGenTextures ( 1, &texName );
glBindTexture ( GL_TEXTURE_2D, texName );
```

```
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                  GL_REPEAT );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                  GL_REPEAT );
glTexParameteri ( GL_TEXTURE_2D,
                 GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri ( GL_TEXTURE_2D,
                 GL_TEXTURE_MIN_FILTER, GL_NEAREST );
glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGB, w, h,
               0, GL_RGB, GL_UNSIGNED_BYTE, Image );
glTexEnvf ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
            GL_DECAL );
glEnable ( GL_TEXTURE_2D );
```

Wywołanie procedury `glPixelStorei` powiadamia OpenGL-a, że poszczególne wiersze danych nie są dopełniane nieznaczącymi bajtami, w celu np. wyrównania długości do wielokrotności 2 lub 4 (co czasem jest istotne dla programu generującego teksturę).

Procedura `glGenTextures` tworzy obiekt (lub obiekty) reprezentujący teksturę w OpenGL-u. Pierwszy parametr określa ile takich obiektów ma być utworzonych, drugi jest tablicą (o elementach typu `GLuint`), o odpowiedniej długości — procedura wstawi do niej identyfikatory utworzonych obiektów.

Procedura `glBindTexture` „uaktywnia" odpowiedni obiekt (o podanym identyfikatorze); dalsze wywołania procedur dotyczą tego obiektu. Procedurę tę wywołamy również przed rysowaniem czegoś, w celu związania konkretnej tekstury z tym czymś. **Uwaga:** w starszych wersjach OpenGL-a ponowne wywołanie `glBindTexture` powoduje błąd wykonania programu, dlatego programy przykładowe w Mesie, które nakładają teksturę, sprawdzają numer wersji. Wypadałoby naśladować te przykłady.

Procedura `glTexParameteri` ustawia różne parametry, które mają wpływ na sposób przetwarzania tekstury. Pierwsze dwa wywołania wyżej powodują, że jeśli pewien punkt ma współrzędne poza kwadratem jednostkowym (dziedziną tekstury), to otrzyma kolor taki, jak gdyby tekstura była powielona okresowo w celu pokrycia całej płaszczyzny. Zamiast `GL_REPEAT` można podać `GL_CLAMP`, i wtedy tekstura poza dziedziną będzie taka, jak w pierwszej lub ostatniej kolumnie lub wierszu tablicy tekseli.

Kolejne dwa wywołania `glTexParameteri` określają sposób filtrowania tekstury, jeśli teksele podczas odwzorowania na piksele będą zmniejszane oraz zwiększane. Wartość `GL_NEAREST` trzeciego parametru oznacza wzięcie próbki z tablicy tekseli, a `GL_LINEAR` oznacza liniową interpolację.

Wreszcie `glTexEnvf` powoduje określenie sposobu traktowania tekstury; parametr `GL_DECAL` oznacza kalkomanię; kolor pikseli jest uzyskiwany tylko przez przefiltrowanie tekstury, bez uwzględnienia własności powierzchni określanych za pomocą procedury `glMaterialf` (ale z uwzględnieniem współczynnika α, jeśli go używamy). Inne możliwe tryby to `GL_REPLACE` (przypisanie koloru oraz współczynnika α), `GL_MODULATE` (mnożenie koloru obiektu przez składowe koloru i przypisanie współcz. α tekstury) i `GL_BLEND` (obliczanie kombinacji afinicznej koloru obiektu i tekstury, ze współcz. α).

Bezpośrednio przed wyświetlaniem obiektów, na które ma być nałożona tekstura, powinniśmy wywołać procedurę `glBindTexture` (ale zobacz uwagę wyżej). Następnie *przed* wyprowadzeniem każdego wierzchołka wielokąta powinniśmy podać jego współrzędne w układzie tekstury. W przypadku tekstur dwuwymiarowych stosujemy do tego procedurę `glTexCoord2f`, której dwa parametry, s i t powinny (w zasadzie) mieć wartości z przedziału [0, 1].

O.8.2 Mipmaping

Aby przyspieszyć teksturowanie obiektów, które na obrazie mogą być małe, można określić kilka reprezentacji tekstury o zmniejszonej rozdzielczości. W tym celu możemy kilkakrotnie użyć procedury `gluScaleImage`, za każdym razem zmniejszając dwa razy wymiary tablicy tekseli. Następnie wywołujemy procedury jak wyżej, ale zamiast jednego wywołania `glTexImage2D`, wywołujemy tę procedurę dla każdej reprezentacji tekstury o zmniejszonej rozdzielczości. Drugi parametr procedury określa *poziom* reprezentacji; pierwsza reprezentacja (o maksymalnej rozdzielczości) ma poziom 0, druga (2 razy mniejsza) poziom 1 itd. Należy w takim przypadku określić *wszystkie* poziomy aż do tekstury o wymiarach 1×1, w przeciwnym razie będą kłopoty z filtrowaniem na końcowym obrazie.

Aby uprościć konstruowanie reprezentacji tekstury o mniejszych rozdzielczościach, można posłużyć się procedurą

```
gluBuild2DMipmaps ( GL_TEXTURE_2D, GL_RGB, w, h,
                    GL_RGB, GL_UNSIGNED_BYTE, data );
```

Procedura ta dokonuje skalowania reprezentacji i wywołuje `glTexImage2D` dla kolejno otrzymanych tablic tekseli.

O.8.3 Tekstury jednowymiarowe

Tekstury jednowymiarowe nakłada się w podobny sposób. W poprzednich punktach wszędzie, gdzie występuje fragment identyfikatora `2D`, należy napisać `1D`, a poza tym procedura `glTexImage1D` zamiast dwóch parametrów określających

wymiary tablicy tekseli, ma tylko 1. W przypadku tekstur jednowymiarowych współrzędna w układzie tekstury nazywa się s, a zatem wywołujemy np. procedurę

```
glTexParameteri ( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S,
                  GL_REPEAT );
```

O.8.4 Tekstury trójwymiarowe

Przykład przygotowania tekstury:

```
glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );
glTexParameteri( GL_TEXTURE_3D,
              GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_3D,
              GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_WRAP_S,
                 GL_REPEAT );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_WRAP_T,
                 GL_REPEAT );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_WRAP_R,
                 GL_REPEAT );
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
           GL_DECAL );
glTexImage3D( GL_TEXTURE_3D, 0, GL_RGBA,
              tex_width, tex_height, tex_depth,
              0, GL_RGBA, GL_UNSIGNED_BYTE, voxels );
```

Jak widać odbywa się to podobnie jak w przypadku dwuwymiarowym. Dziedzina tekstury jest sześcianem, jej punkty są opisane trzema współrzędnymi, s, t, r. Jest jeszcze czwarta współrzędna, q, której domyślna wartość to 1. Czwórka współrzędnych jednorodnych (s,t,r,q) może być użyta do określenia punktu w dziedzinie tekstury; współrzędnych jednorodnych można też używać dla tekstur jedno- i dwuwymiarowych, podając cztery współrzędne (z których jedna, r, albo dwie, t i r są ignorowane).

O.8.5 Współrzędne tekstury

Współrzędne tekstury podawane przez wywołanie `glTexCoord*` są poddawane przekształceniu, które jest określone za pomocą macierzy przechowywanej na wierzchołku stosu przekształceń tekstury. Przypominam, że każda implementacja OpenGL-a gwarantuje minimum 2 miejsca na tym stosie i aby spowodować,

że procedury `glLoadIdentity`, `glTranslate*`, `glRotate*`, `glScale`, `glLoadMatrix*` i `glMultMatrix*` działały na tym stosie, należy wywołać najpierw

```
glMatrixMode ( GL_TEXTURE );
```

Domyślnie (tj. przed wykonaniem pierwszej akcji na tym stosie) macierz przekształcenia tekstury jest jednostkowa.

Ważnym elementem określania tekstury jest możliwość **automatycznego generowania współrzędnych tekstury**. Wywoływanie procedury `glTexCoord*` przed każdym wywołaniem `glVertex*` bywa niewygodne i czasochłonne, a poza tym jest czasem niemożliwe, na przykład wtedy, gdy chcemy nałożyć teksturę na „gotowe” obiekty, takie jak czajnik (tworzony przez `glutTeapot`). W takich przypadkach możemy posłużyć się procedurami

```
glTexGen* ( coord, pname, param );
glTexGen*v ( coord, pname, *param );
```

której kolejne parametry to:

- `coord` — musi mieć wartość `GL_S`, `GL_T`, `GL_U` lub `GL_Q`, która określa jedną z czterech współrzędnych do generowania.
- `pname` — ma wartość
  - `GL_TEXTURE_GEN_MODE` — parametr `param` musi mieć jedną z wartości `GL_OBJECT_LINEAR`, `GL_EYE_LINEAR` albo `GL_SPHERE_MAP`. W pierwszym przypadku odpowiednia współrzędna tekstury jest kombinacją liniową czterech współrzędnych (jednorodnych) wierzchołka, o współczynnikach podanych w tablicy przekazanej jako trzeci parametr procedury `glTexGen*v`, której drugi parametr jest równy `GL_OBJECT_PLANE`. Działa to tak, że wartość współrzędnej jest proporcjonalna do odległości (ze znakiem) od pewnej płaszczyzny.

    Jeśli `pname=GL_EYE_LINEAR`, to współrzędna tekstury powstaje przez pomnożenie wektora współrzędnych jednorodnych wierzchołka przez wektor, który jest iloczynem wektora $[p_1, p_2, p_3, p_4]$ i macierzy M^{-1}; wektor $[p_1, p_2, p_3, p_4]$ podajemy wywołując `glTexGen*v` z drugim parametrem równym `GL_EYE_LINEAR`, a macierz M jest przechowywana na szczycie stosu `GL_MODELVIEW`. Zatem, współrzędne tekstury określa się w tym przypadku w układzie obserwatora.

    Wartość `GL_SPHERE_MAP` służy do nakładania tekstury, która opisuje obraz otoczenia danego obiektu, odbijający się w tym obiekcie. Więcej powiem na konkretne zapotrzebowanie.

- GL_OBJECT_PLANE, GL_EYE_PLANE — te wartości drugiego parametru procedury glTexGen*v określają, w którym układzie podawane są współczynniki kombinacji liniowej branej do automatycznego generowania współrzędnych tekstury.

O.8.6 Pośrednik tekstury

Tekstury często zajmują dużo miejsca w pamięci (sytuacja bywa delikatna wtedy, gdy mamy akcelerator wyposażony w kilka–kilkanaście MB pamięci na tekstury, które mogą być nakładane bardzo szybko, w przeciwieństwie do tekstur przechowywanych w „zwykłej" pamięci operacyjnej). Istnieje obiekt, tzw. **pośrednik tekstury** (ang. *texture proxy*), którego celem jest dostarczanie informacji, czy dysponujemy odpowiednimi zasobami komputera wystarczającymi do określenia tekstury o pożądanej wielkości. Aby go użyć, wywołujemy

```
glTexImage2D ( GL_PROXY_TEXTURE_2D, ..., NULL );
```

(parametry zastąpione kropkami są takie same jak w przypadku określania tekstury). Możemy następnie dowiedzieć się, czy wystarczy pamięci na taką teksturę, wywołując

```
glGetTexLevelParameter*v ( target, level,
                           pname, *params );
```

Parametry target i level mają to samo znaczenie, co w procedurze glTexImage2D, natomiast pname określa, którą informację chcemy uzyskać. Jego wartość może być równa GL_TEXTURE_WIDTH, Gl_TEXTURE_HEIGHT, GL_TEXTURE_BORDER, GL_INTERNAL_FORMAT i inne. Informacja jest wpisywana przez procedurę do tablicy params.

Informacja podawana przez procedurę glGetTexLevelParameter*v jest prawdziwa w sytuacji, gdy tylko *jedna*, właśnie ta tekstura ma istnieć. Jeśli chcemy w danej chwili mieć *wiele* tekstur, to mogą one się nie zmieścić w pamięci akceleratora.

O.9 Listy obrazowe

O.9.1 Wiadomości ogólne

Lista obrazowa (ang. *display list*) jest strukturą danych, w której są przechowywane ciągi komend OpenGL-a, równoważne skutkom wywołań procedur biblioteki GL (z pewnymi wyjątkami, które nie mogą być umieszczone w liście). Listy obrazowe są przechowywane w pamięci akceleratora graficznego, w związku z czym

wykonanie tych komend może być znacznie szybsze niż wykonanie równoważnych procedur. Oszczędność czasu bierze się z wyeliminowania komunikacji między procesorem wykonującym program i akceleratorem oraz innych obliczeń (np. obliczeń wartości parametrów).

Listy obrazowe (jedną lub więcej na raz) tworzymy wywołując procedurę glGenLists, której parametr określa liczbę tworzonych list. Wartością procedury jest liczba całkowita, która jest identyfikatorem pierwszej utworzonej listy — pozostałe listy utworzone w tym wywołaniu procedury mają kolejne identyfikatory. Wartość 0 procedury oznacza, że nie było możliwe utworzenie żądanej liczby list.

Utworzone listy są początkowo puste. Aby umieścić w liście zawartość wywołujemy procedurę glNewList. Ma ona dwa parametry, z których pierwszy jest identyfikatorem listy obrazowej, a drugi określa tryb jej pracy. Jeśli parametr ten ma wartość GL_COMPILE, to komendy odpowiadające wywołaniom procedur OpenGL-a będą tylko umieszczane w liście. Jeśli ma on wartość GL_COMPILE_AND_EXECUTE, to procedury OpenGL-a są wykonywane i jednocześnie zapamiętywane w liście.

Po wywołaniu glNewList umieszczamy w liście zawartość, wywołując odpowiednie procedury OpenGL-a. Zamknięcie listy sygnalizujemy wywołując procedurę glEndList (bez parametrów). Aby wykonać komendy zawarte w liście wywołujemy procedurę glCallList z parametrem, który jest jej identyfikatorem.

Aby zlikwidować listy należy wywołać procedurę glDeleteLists, której dwa parametry określają pierwszy identyfikator i liczbę list (o kolejnych identyfikatorach), które mają zostać usunięte.

Należy pamiętać, że lista obrazowa jest po zamknięciu „czarną skrzynką", tj. jej zawartość nie może być zmieniona. Jeśli obiekty wyświetlane przez program uległy zmianie, to listy z komendami wyświetlającymi należy zlikwidować i utworzyć je na nowo. Sposób wykorzystania list może być taki: możemy utworzyć listę obrazową wyświetlającą obiekty, a następnie wyświetlać je wielokrotnie, np. przy różnie określonym rzutowaniu (w sytuacji, gdy oglądamy obiekty z różnych stron; wtedy oczywiście lista zawiera tylko komendy wyświetlania obiektów, ale nie komendy określające rzutowanie). Inny sposób wykorzystania list jest taki: mamy scenę złożoną z kilku obiektów, które mogą zmieniać wzajemne położenie. Wtedy każdy z tych obiektów będzie miał swoją listę obrazową. Możemy też utworzyć listę odpowiadającą całej scenie. Będzie ona zawierając komendy wyświetlenia tych list, przedzielone komendami określającymi przekształcenia mające na celu ustalenie położeń obiektów opisanych w poszczególnych listach. Zmiana położeń obiektów wymaga zlikwidowania i ponownego utworzenia tylko tej jednej listy.

O.9.2 Rysowanie tekstu

Aby umieścić na obrazie tekst przy użyciu OpenGL-a, należy zrobić dwie rzeczy: określić sposób tworzenia obrazów liter i innych znaków, a następnie, mając dany napis (czyli np. ciąg kodów ASCII kolejnych znaków), spowodować wyświetlenie odpowiednich znaków. OpenGL umożliwia zarówno tworzenie obrazów liter poprzez wyświetlanie gotowych obrazków rastrowych, jak i wyświetlanie figur geometrycznych (z odpowiednim rzutowaniem, oświetleniem, a nawet teksturowaniem), które tworzą litery. W obu przypadkach drugi etap (czyli spowodowanie narysowania tekstu) może być taki sam.

Znaki pisarskie możemy przygotować w ten sposób, że dla każdego znaku tworzymy listę obrazową, która go wyświetla. Lista taka zawiera polecenie umieszczenia na ekranie odpowiedniego obrazu rastrowego, albo narysowania np. bryły, która oglądana z odpowiedniej strony wygląda jak litera. Aby utworzyć zestaw znaków (font) rastrowy, składający się ze spacji i z 26 liter alfabetu angielskiego, na początku działania programu (podczas inicjalizacji) wykonujemy instrukcje

```
GLubyte space[13] = {};
GLubyte letters[][13] = {...};
glPixelStorei ( GL_UNPACK_ALIGNMENT, 1 );
fontofs = glGenLists ( 'Z' );
glNewList ( fontofs+' ', CL_COMPILE )
glBitmap ( 8, 13, 0.0, 2.0, 10.0, 0.0, sp );
glEndList ();
for ( i = 0, j = 'A'; j <= 'Z'; i++, j++ ) {
  glNewList ( fontofs+j, GL_COMPILE );
  glBitmap ( 8, 13, 0.0, 2.0, 10.0, 0.0,
             letters[i] );
  glEndList ();
}
```

W tym przykładzie litery mają wymiary 13 × 8 pikseli. Każda z nich jest reprezentowana przez 13 bajtów w tablicy `letters` (jest 1 bit na piksel, co jest określone przez wywołanie procedury `glPixelStorei`). Spacja jest opisana przez 13 bajtów zerowych, w tablicy space. Procedura `glBitmap` wyświetla odpowiedni obrazek rastrowy, ale w tym przykładzie jej wywołanie zostaje tylko odnotowane w odpowiedniej liście obrazowej. Dzięki utworzeniu bloku list numerowanych od zera, numer listy zawierającej komendę rysowania każdego znaku jest równy sumie indeksu pierwszej listy i odpowiedniego kodu ASCII.

Parametry procedury `glBitmap` to szerokość, wysokość, dwie współrzędne punktu referencyjnego obrazka (względem dolnego lewego rogu), współrzędne

określające przesunięcie następnego obrazka i bajty określające obrazek. Zatem kolejne znaki w przykładzie będą zajmowały szerokość 10 pikseli.

Aby wyświetlić napis składający się z `n` znaków przechowywanych w tablicy `s` ustawiamy miejsce, od którego ma się zaczynać napis, wywołując procedurę `glRasterPos*` (np. `glRasterPos2i` z dwiema współrzędnymi całkowitymi lub `glRasterPos4fv` z parametrem, który jest tablicą czterech liczb). Punkt podany przez wywołanie tej procedury jest rzutowany zgodnie z ogólnymi zasadami. Następnie wywołujemy procedury

```
glPushAttrib ( GL_LIST_BIT );
glListBase ( fontofs );
glCallLists ( n, GL_UNSIGNED_BYTE, (GLubyte*)s );
glPopAttrib ();
```

Procedury `glPushAttrib` i `glPopAttrib` w tym przykładzie zapamiętują i przywracają zmienne stanu związane z listami obrazowymi. Procedura `glCallLists` wyświetla zawartość kolejnych list obrazowych, których indeksy bierze z napisu. Wywołanie procedury `glListBase` powoduje, że do każdego indeksu będzie dodana liczba `fontofs`, czyli numer pierwszej listy. Warto zwrócić uwagę, że pokazany tu mechanizm umożliwia łatwe korzystanie z wielu różnych fontów, których znaki zajmują rozłączne bloki list obrazowych.