Programowanie OpenGL

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf”.
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Programowanie OpenGL

Rafał Stęgierski

Lublin 2011

Instytut Informatyki UMCS
Lublin 2011

Rafał Stęgierski
PROGRAMOWANIE OPENGL

Recenzent: Andrzej Bobyk

Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl.

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
ESUS Agencja Reklamowo-Wydawnicza Tomasz Przybylak
ul. Ratajczaka 26/8
61-815 Poznań
www: www.esus.pl

ISBN: 978-83-62773-14-5

Spis treści

Przedmowa vii

1 Narodziny OpenGL 1
1.1. Historia grafiki 3D 2
1.2. IRIS GL i OpenGL 3

2 Architektura OpenGL 7
2.1. Maszyna stanów 9
2.2. Potok przetwarzania 9
2.3. Konwencja nazewnicza i typy danych 10

3 Integracja i interakcja ze środowiskiem 15
3.1. GLUT 16
3.2. Inne API 35

4 Prymitywy 37
4.1. Prymitywy OpenGL 38
4.2. Predefiniowane modele GLU 49
4.3. Predefiniowane modele GLUT 53

5 Macierze i przekształcenia geometryczne 57
5.1. Współrzędne jednorodne 58
5.2. Macierz modelu-widoku 59
5.3. Macierze przekształceń 59
5.4. Złożenie przekształceń 60
5.5. Przesunięcie 61
5.6. Obrót 61
5.7. Skalowanie 63
5.8. Stos macierzy modelu-widoku 64
5.9. Odwzorowanie hierarchii 66
5.10. Normalne 71

6 Rzutowania 81

6.1. Macierz projekcji 83
6.2. Rzutowanie ortogonalne 84
6.3. Rzutowanie perspektywiczne 84
6.4. Obsługa rzutowań w OpenGL i GLU 86
6.5. Pozycja i parametry kamery 88

7 Przygotowanie sceny, listy wyświetlania i tablice wierzchołków **93**
7.1. Czyszczenie sceny 94
7.2. Tryb cieniowania 96
7.3. Wielokrotne próbkowanie 97
7.4. Listy wyświetlania 97
7.5. Tablice wierzchołków 100

8 Model oświetlenia **105**
8.1. Światło w OpenGL 107
8.2. Modyfikacje świateł 110
8.3. Materiał 112

9 Teksturowanie **117**
9.1. Biblioteka GLPNG 118
9.2. Obsługa teksturowania 119
9.3. Mipmapy 127
9.4. Tekstury trójwymiarowe 127

10 Mgła **131**
10.1. Podstawowe parametry mgły 132
10.2. Konfiguracja zaawansowana 133

A Parametry funkcji `pngBind()` **137**

B Miejsca w sieci **139**

Bibliografia **141**

Skorowidz **143**

Przedmowa

Trudno przecenić znaczenie OpenGL dla współczesnej grafiki trójwymiarowej czasu rzeczywistego. Bardzo często w systemach wizualizacji danych, projektowaniu przemysłowym, modelowaniu trójwymiarowym, wirtualnej rzeczywistości, symulatorach lotu, czy nawet, serwerach okien (*XGL*) sercem całej aplikacji są właśnie implementacje jego specyfikacji. Niezwykła elastyczność w możliwościach dostosowania do własnych wymagań zawdzięczana swoistej niskopoziomowości i łatwość migracji między platformami tak programowymi, jak i sprzętowymi spowodowała szybkie zakotwiczenie w rynku. Powszechna akceptacja ze strony liczących się producentów procesorów graficznych, którzy nie tylko zapewnili aktywne wsparcie w postaci implementacji nowych rozwiązań w kolejnych generacjach oferowanych przez siebie rozwiązań, ale i walnie przyczyniły się do wprowadzenia istotnych zmian w nowych wersjach specyfikacji przekładających się tak na wzrost wydajności jak i elastyczności. Zaproponowany specyficzny sposób komunikacji i działanie oparte na maszynie stanów spowodowały, że jest OpenGL w wielu wypadkach rozwiązaniem wprost idealnym.

Zapoznanie się z całą specyfikacją jest zadaniem niebanalnym. Już dla wersji 1.4 papierowy opis samych funkcji, tak zwany "Blue Book", składających się na nią miał prawie osiemset stron. Od tego czasu dokonano znacznego rozszerzenia ich zestawu wprowadzając całe grupy nowych funkcjonalności. Dwa oficjalne podręczniki, które, niestety, do tej pory nie doczekały się polskiego wydania, to kolejne, prawie, dwa tysiące stron.

Pojawić się może pytanie, jak względem tych dogłębnie wyczerpujących temat pozycji ma się ta skromna książka. Otóż jej zadaniem jest wprowadzenie w tematykę programowania grafiki, a szczególnie OpenGL w sposób jak najbardziej przystępny. W kolejnych rozdziałach pojawiają się tak informacje na temat specyfikacji, jak i przykłady użycia konkretnych funkcji. Znaleźć można w niej też wskazówki, gdzie można natrafić na potencjalne pułapki, czy jakie narzędzia zewnętrze mogą ułatwić nam życie. Pokazana pokrótce została droga, która umożliwia obiektową implementację aplikacji na wyższym poziomie abstrakcji, ale opartej na OpenGL. Co ważne, cały czas zagadnienia, które się pojawiają miały, być jak najbardziej praktyczne,

ale nie odarte całkowicie z podstaw teoretycznych tak, aby wraz ze zdobywanymi umiejętnościami szła wiedza, gdzie leży przyczyna takich, a nie innych, efektów. Jeżeli ciekawość czytelnika zostanie pobudzona rozważaniami znajdującymi się na tych stronach, to powinien sięgnąć do książek ujętych w bibliografii. Niewątpliwie, nie zawiera ona wszystkiego co na ten temat zostało napisane, ale pozycje te dobrane zostały tak, że są w stanie rozwiać wszelkie wątpliwości. Warto też śledzić serwisy i fora internetowe skupione na tej tematyce, ponieważ pojawia się tam możliwość bezpośredniego poznania, czasem bardzo specjalistycznych, aspektów użycia biblioteki.

Rozdział 1

Narodziny OpenGL

1.1. Historia grafiki 3D **2**
1.2. IRIS GL i OpenGL **3**

1.1. Historia grafiki 3D

Trudno jednoznacznie wskazać punkt, w którym narodziła się grafika komputerowa i wizualizacja trójwymiarowa. Czy za cezurę przyjąć powstanie w starożytności geometrii, czy może wyłonienie się z prac naukowców kanonu algorytmów organizujących i optymalizujących przetwarzanie matematycznego opisu obiektów przestrzennych. Może, idąc po najmniejszej linii oporu, należy wskazać na moment zastąpienia w komputerach komunikacji tekstowej na wizualną. Wybór każdego z tych momentów historii, przy odrobinie samozaparcia, można równie dobrze uzasadnić. Każdy z nich trudno też rozpatrywać w oderwaniu od pozostałych.

Nie ma wątpliwości, że kamieniem węgielnym była geometria Euklidesa, ale to podejście analityczne, zaproponowanego przez Kartezjusza, które pozwoliło w sposób jednoznaczny opisać kształt i położenie obiektów w przestrzeni, dało możliwość budowy wirtualnych światów w pamięci komputera. Nie można zapominać też o Jamesie Josephie Sylvestrze i rachunku macierzowym, bo to on uprościł rzeczy nieproste i to nie tylko na poziomie notacji, ale przede wszystkim metod.

Duże znaczenie miała też praca Piero della Francesca *De Prospectiva Pingendi* formalizująca to, co artyści już od początku renesansu znali i używali i bez czego grafika komputerowa, opierająca się na liczbach, istnieć by nie mogła - perspektywę. Nie mniej istotne wydają się też prace Ptolemeusza, Johanna Lamberta, Pierre Bouguera poświęcone optyce i światłu.

Przykłady te można mnożyć w nieskończoność ponieważ, na fundament grafiki komputerowej składa się wiele cegieł wziętych z różnych dziedzin nauki i sztuki. Odpowiednio przetworzone, czasem uproszczone, pozwalają na swobodną kreację obrazów, zarówno tych próbujących naśladować rzeczywistość, jak i całkiem od niej odmiennych.

Jeżeli zaś popatrzymy na historię komputerów, to okaże się, że grafika jako sposób przekazywania wyników przetwarzania danych pojawiła się wyjątkowo szybko. w roku 1954, niecałe dziesięć lat po tym, jak zaprezentowany Światu został ENIAC, wielki błękitny gigant miał już w swojej ofercie wyświetlacz CRT o rozdzielczości 1024 na 1024 punkty (sic!) i pracujący w trybie wektorowym. Używany był wraz komputerami IBM 701, 704 i 709. Sześć lat później pojawia się wyjątkowa maszyna, DEC PDP-1, która zmieniła spojrzenie na komputery. Była nie tylko pierwszym minikomputerem, posiadała w standardzie wyświetlacz, i jako jedyna w tamtych czasach dostarczana była z pierwszą grą komputerową z prawdziwego zdarzenia. Była nią, napisaną w 1961 roku przez Steve Russella, *Spacewars*.

W tym samym roku co PDP-1 zaprezentowane też są pierwsze komputery przeznaczone do projektowania przemysłowego DAC-1, a pracujący w Boeingu William Fetter wprowadza termin "grafika komputerowa".

W połowie lat sześćdziesiątych znaczenie grafiki rośnie błyskawicznie. Jack Elton Bresenham prezentuje algorytmy umożliwiające rasteryzację linii, a potem również kół i elips, a Arthur Appel opisuje algorytm raycastingu. Dziesięciolecie zamyka powołanie do życia SIGGRAPH, organizacji, której znaczenie dla grafiki trudno przecenić.

Lata siedemdziesiąte dostarczą nam cieniowanie Gouraud i Phong, mechanizmy ukrywania niewidocznych powierzchni, bufor głębokości, algorytmy teksturowania obiektów, fraktale, mapowania środowiskowe, bumpmapping i milion lub więcej pomniejszych rozwiązań. Jednym słowem były szalenie owocne.

W kolejnym dziesięcioleciu dołączą, do długiej już listy metod, systemy cząsteczkowe, mipmapping, bufor przezroczystości, cieniowanie Cooka, radiosity co, w zasadzie, zamknie grupę najważniejszych algorytmów dla współczesnej grafiki. w następnych latach widoczne będzie ich udoskonalanie i optymalizacja, a w połączeniu z nieustannym wzrostem mocy obliczeniowych pozwoli na sukcesywne przenoszenie technik znanych z oprogramowania renderingu offline do rozwiązań czasu rzeczywistego.

1.2. IRIS GL i OpenGL

Firma Silicon Graphics, znana również pod akronimem SGI, w 1981 zaczęła swoją działalność jako producent wydajnych terminali graficznych na potrzeby, między innymi, rodziny minikomputerów DEC VAX. Rozwiązanie okazało się na tyle udane, że szybko zostało zaoferowane jako samodzielna stacja robocza wyposażona w sprzętową akcelerację grafiki. Gwałtownie powiększające się na początku lat dziewięćdziesiątych dwudziestego wieku zapotrzebowanie na grafikę trójwymiarową wysokiej jakości, którego źródłem był w tamtym momencie przemysł filmowy, wygenerowało na rynku konieczność znalezienia odpowiednio systemów, które byłyby zoptymalizowane pod kątem nowych potrzeb. SGI szybko wskoczyła w nową niszę z ofertą bardzo wydajnych systemów wizualizacyjnych Onyx, stacji roboczych Indigo, czy w późniejszych latach O2 i Octane. Wszystkie one posiadały sprzętowe wsparcie dla trójwymiaru i walnie przyczyniły się do sukcesu wielu hollywoodzkich superprodukcji tamtego czasu. w tym okresie na potrzeby oprogramowania oferowanego w ramach systemu operacyjnego IRIX i oprogramowania oferowanego dla niego opracowana została biblioteka IRIS GL (Integrated Raster Imaging System Graphics Library). Okazała się na tyle dobra, prosta i, przede wszystkim, elastyczna, że błyskawicznie i praktycznie do zera spadło znaczenie otwartego standardu PHIGS.

PHIGS, czyli *Programmer's Hierarchical Interactive Graphics System* był skomplikowany, wymagał od użytkownika budowy grafu sceny, obsługiwał tylko dwa rodzaje cieniowania i żadnych metod teksturowania. Jego koncepcje oparte były na rozwiązaniach zawartych w GKS - *Graphical Kernel System*, dwuwymiarowym standardzie, który miał zapewniać łatwość portowania grafiki dwuwymiarowej pomiędzy nie tylko różnymi językami programowania, ale również platformami systemowymi. Ostatecznie PHIGS nie pomógł fakt, że został uznany przez szerokie grono organizacji standaryzacyjnych i został zatwierdzony jako normy ANSI X3.144-1988, FIPS 153 i ISO/IEC 9593.

IRIS GL w krótkim czasie stało się nie pisanym standardem i po pewnych zmianach związanych z usunięciem z API rzeczy, które okazały się zbędne, mało znaczące lub obwarowane ograniczeniami patentowymi i licencyjnymi doczekało się w roku 1992 inkarnacji w postaci standardu OpenGL 1.0. To był punkt przełomowy i część gorliwych, do tej pory, wyznawców PHIGS błyskawicznie przygotowało implementacje dla oferowanych przez siebie urządzeń.

Warto też wspomnieć, że połowa lat dziewięćdziesiątych to jest również okres, w którym pojawiają się pierwsze rozwiązania oferujące sprzętowe wsparcie dla akceleracji generacji trójwymiarowych obrazów przeznaczone dla mas. Ich cena i przeznaczenie na rynek komputerów osobistych spowodowały błyskawiczny spadek cen i wyindukowało przyspieszenie rozwoju technologii. Wśród firm przodujących w tamtym czasie było ATI z *Rage*, Matrox z *Mystique* oraz S3 z *Virge*. Jednak dopiero premiera układów 3sfx serii *Voodoo* i dostarczane wraz z nimi API Glide, które nomen, omen, wzorowane było na OpenGL spowodowało prawdziwe parcie na wykorzystanie nowych możliwości przez największych graczy pośród producentów gier. Co więcej, w 1996 roku 3dfx wypuścił nowy sterowni, który nazywał się MiniGL i był niczym innym jak okrojonym API OpenGL przygotowanym na potrzeby bijącej wszelkie rekordy popularności gry stworzonej przez id Software - *Quake*.

Kolejne pięć lat trzeba było czekać na poprawioną wersję 1.1. Widać po niej było, że twórcy byli szczególnie mocno wyczuleni na oczekiwania użytkowników i rozszerzyli standard o dodatkowe mechanizmy związane z teksturowaniem. Ten trend jest zachowany i następne wersje od 1.2 do 1.4 dodaje głównie kolejne rozwiązania mocno skupione na sprzętowym wsparciu tekstur.

Od 2000 roku nad rozwojem standardu czuwa Khronos Group, organizacja standaryzacyjna powołana przez kilka dużych firm skupionych na grafice trójwymiarowej z sektora tak producentów sprzętu, jak i oprogramowania.

Rysunek 1.1. id Software Quake.

Rok 2004 przynosi pierwszą specyfikację wersji 2 OpenGL, co w konsekwencji prowadzi do dużych zmian głównie związanych z uelastycznieniem renderingu za sprawą programowalnych mechanizmów cieniowana dla wierzchołków i fragmentów. Oznacza to też wprowadzenie języka GLSL - *OpenGL Shading Language* zbliżonego składnią do C i zorientowanego na obsłużenie nowych możliwości oferowanych przez standard. Równocześnie z nową wersją zostało wprowadzone wsparcie dla niskopoziomowego języka programowania procesorów graficznych - ARB.

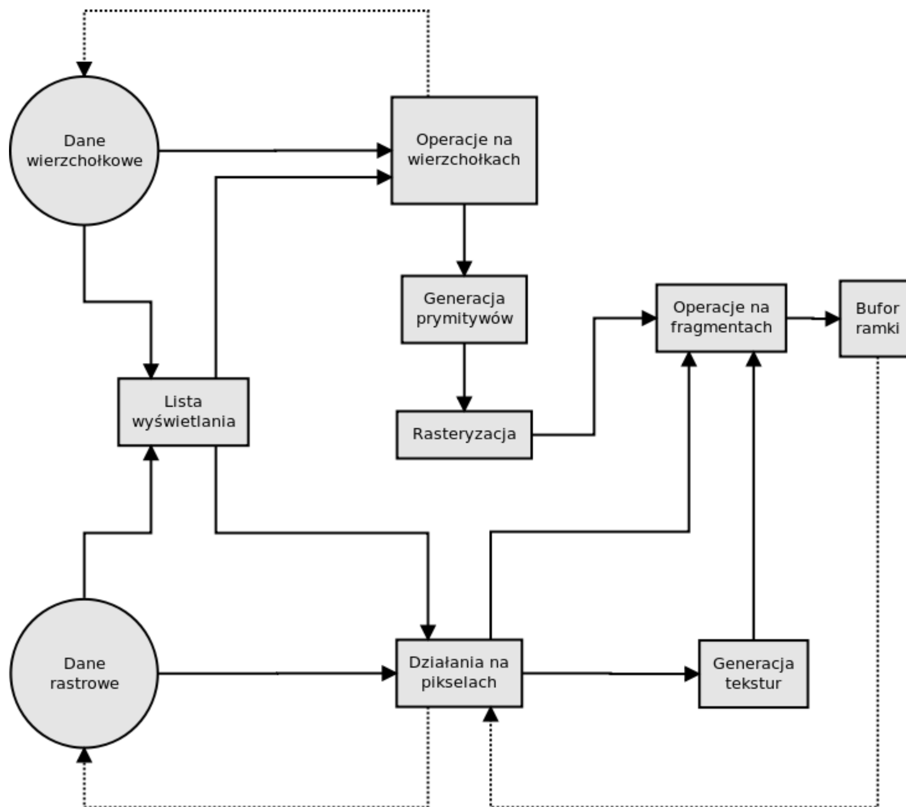
Kolejna wersja, została oznaczona kodowo *Longs Peak*, miała przynieść zasadnicze zmiany w sposobie obsługi w tym rezygnację ze ściśle ustalonego procesu przetwarzania danych na w pełni zarządzalny. Planowano też wprowadzenie mechanizmu deprecjacji, który pozwoliłby na usuwanie z kolejnych wersji specyfikacji tych funkcji, które obecnie nie są już używane, a znajdują się w niej ze względu na zgodność wstecz. Jednak po oficjalnej premierze pod koniec 2008 roku jasnym się stało, że rewolucja jeszcze nie nadeszła, a wielkie zmiany zostały odłożone na bliżej nieokreśloną przyszłość.

Rok 2010 stoi pod znakiem wersji 4.0, która wprowadziła, między innymi, obsługę sprzętowego wsparcia teselacji i wsparcie dla 64 bitowych liczb zmiennoprzecinkowych podwójnej precyzji.

Rozdział 2

Architektura OpenGL

2.1. Maszyna stanów **9**
2.2. Potok przetwarzania **9**
2.3. Konwencja nazewnicza i typy danych **10**



Rysunek 2.1. Potok przetwarzania w OpenGL

Na OpenGL, jeżeli doliczymy też standardowo dostępną wraz z nim bibliotekę GLU, składa się około 700 różnych funkcji, które umożliwiają tworzenie aplikacji trójwymiarowych od opisu sceny, poprzez definicję obiektów, ich teksturowanie, oświetlenie, po modyfikację i wyświetlenie. Kompletne API jest niezależne od sprzętu i może pracować w architekturze klient-serwer co pozwala na renderowanie sceny na zdalnej, wysokowydajnej maszynie i wyświetlanie na dużo słabszej końcówce. Dodatkowo, współczesny OpenGL jest skonstruowany w sposób umożliwiający dokonywania modyfikacji sposobu przetwarzania danych na poziomie poszczególnych bloków w potoku za pomocą programów pisanych z wykorzystaniem języka programowania zbliżonego do C.

Obsługa OpenGL współcześnie została zaimplementowana praktycznie na każdym dostępnym procesorze graficznym, a najnowsze produkty, szczególnie te z wyższych półek, najważniejszych producentów często oferują pełną obsługę najnowszej wersji standardu. Warto też wspomnieć o istniejącym implementacji API OpenGL z w pełni programowym renderingiem sceny w postaci projektu Mesa 3D Graphics Library (`http://www.mesa3d.org/`)

w obecnej chwili oferująca zgodność z wersją OpenGL 2.1 wraz z obsługą GLSL.

2.1. Maszyna stanów

OpenGL został zaprojektowany jako maszyna stanów. Jest to dość specyficzne podejście i zasadniczo różne od wspomnianego sposobu pracy dla PHIGS. Wszelkie wywoływane w ramach naszego programu funkcje wchodzące w skład biblioteki powodują zmianę stanu maszyny i jest on stabilny pomiędzy zmianami. Tyczy się to tak samo wybranego koloru, przekształceń jak i parametrów oświetlenia, czy rzutowania. Przykładowo, wszystkie narysowane elementy będą czerwone od momentu gdy ustawimy taki właśnie kolor. Jeżeli będziemy chcieli dodać element w innym kolorze konieczne będzie zmienienie stanu maszyny poprzez wskazanie na ten kolor, a następnie przekazanie opisu elementu.

Część stanów jest binarnych i polega na ich włączeniu z wykorzystaniem funkcji `glEnable()` z odpowiednim parametrem, lub wyłączeniu przy użyciu `glDisable()`. Przykładem może być tu włączenie obsługi oświetlenia i kolejnych świateł na scenie.

2.2. Potok przetwarzania

W OpenGL dane o położeniu poszczególnych wierzchołków i danych rastrowych przechodzą przez szereg etapów przetwarzania zawartych w potoku. Poszczególne bloki przetwarzania, jak widać na rysunku 2.1 mogą być ze sobą powiązane. Ostateczny wynik renderingu trafia do bufora ramki.

W bloku operacji na wierzchołkach ma miejsce przeliczanie współrzędnych dla poszczególnych wierzchołków w oparciu o zdefiniowane przekształcenia geometryczne, wynikające z wybranego sposobu rzutowania i zdefiniowanego obszaru ekranu. Tu też przeliczane są wszelkie zależności zależne od wybranego modelu oświetlenia i parametrów materiału przypisanego do obiektu.

Kolejnym etapem jest generacja prymitywów, na etapie której wykonywane jest również sortowanie wierzchołków w oparciu o z-bufor, o ile jest włączona jego obsługa, cięcie wielokątów poprzez płaszczyzny zdefiniowanego ostrosłupa widzenia.

Następnie dane opisujące scenę z postaci wektorowej zostają przetłumaczone na specjalną postać rastrową.

Równolegle przetwarzane są dane rastrowe, które mogą pochodzić tak z pamięci operacyjnej, jak i wyniku działania potoku w buforze ramki, poczynając od konwersji do określonego formatu, poprzez skalowanie i generację wariantów o różnym rozmiarze po generację tekstur.

Gdy wchodzimy w blok operacji na fragmentach, tory przetwarzania wektorowego i rastrowego zostają połączone a tekstury zostają nałożone na poszczególne płaszczyzny modelu. Dokonywane jest wygładzanie poprzez mechanizmy aliasingu. Ostatecznie wynik działania zostaje wprowadzony do bufora ramki.

Pomiędzy torami dla przetwarzania danych rastrowych i wierzchołkowych znajdują się listy wyświetlania, o których będzie mowa w rozdziale 7, a które umożliwiają powtórne użycie zdefiniowanych obiektów.

Do wprowadzenia specyfikacji OpenGL w wersji 2.0 akcje mające miejsce na każdym etapie potoku były sztywno narzucone. Współcześnie istnieje możliwość ich modyfikacji przy użyciu programów wierzchołków, fragmentów i geometrii. Te ostatnie dodane dopiero w wersji 3.1.

2.3. Konwencja nazewnicza i typy danych

W ramach API OpenGL dokonano ujednolicenia schematu nazw, co w znaczny sposób ułatwia szybkie ustalenie, z czym mamy do czynienia. Dodatkowo w przypadku funkcji wyszczególniona została liczba podawanych parametrów i ich typ. Zabieg ten pozwolił na wprowadzenie wielu wariantów tej samej funkcji dla różnych typów przekazywanych wartości. Dzięki temu, że działały tak samo, miały, poza ostatnim członem, takie same nazwy. Przykładem może być tu `glColor()`, który występuje aż w 32 wariantach.

Może się takie podejście wydawać dziwne dla osoby programującej w C++, czy innych językach, w których dostępne jest przeciążanie funkcji, ale należy mieć na względzie, że OpenGL powstał jako biblioteka C, a tam takich dobrodziejstw nie ma.

Przyjęto założenie, że wszystkie nazwy funkcji poprzedzone zostaną prefiksem *gl*, każdy kolejny człon nazwy pisany będzie z dużej litery, a sufiks opisywał będzie parametry (patrz tabela 2.1), o ile będzie istniał więcej niż jeden jej wariant. Dodatkowo, jeżeli parametr przekazywany będzie w postaci wektora, czyli jednowymiarowej tablicy, dodawana jest na końcu litera *v*. Ten schemat został w większości przypadków przyjęty również przy tworzeniu bibliotek pomocniczych odpowiednio zmianie uległ tylko prefiksem.

Pora na przykład. w tym celu wrócimy do wspomnianej już funkcji `glColor()`. Jej wariant `glColor3f(GLfloat red, GLfloat green, GLfloat blue)` przyjmuje trzy parametry typu zmiennoprzecinkowego 32 bitowego,

Tabela 2.1. Typy danych OpenGL i odpowiadające im sufiksy

Nazwa typu w OpenGL	Reprezentowany typ danych	Używany sufix
GLbyte	całkowity na 8 bitach	b
GLubyte, GLboolean	całkowity na 8 bitach bez znaku	ub
GLshort	całkowity na 16 bitach	s
GLushort	całkowity na 16 bitach bez znaku	us
GLint, GLsizei	całkowity na 32 bitach	i
GLuint, GLenum, GLbitfield	całkowity na 32 bitach bez znaku	ui
GLfloat, GLclampf	zmiennoprzecinkowy na 32 bitach	f
GLdouble, GLclampd	zmiennoprzecinkowy na 64 bitach	d

zaś `glColor3fv(const GLfloat *v)` jeden trzyelementowy wektor tego samego typu. Prawda, że proste?

OpenGL nie wprowadza własne typy danych (patrz tabela 2.1), które są tak na prawdę aliasami dla odpowiednich typów charakterystycznych dla danej platformy tak programowej, jak i sprzętowej. Takie rozwiązanie ma na celu maksymalną przenośność kodu. Dobrą praktyką jest więc ich używanie, nawet jeżeli w C i C++ *GLfloat* jest tym samym co *float*.

W przypadku zdefiniowanych stałych konwencja mówi, że cała nazwa jest pisana wielkimi literami, występuje stały prefiks `GL` i poszczególne człony nazwy rozdzielone zostały z wykorzystaniem znaku podkreślenia. Na przykład `GL_ALPHA_TEST` lub `GL_LIGHTING`.

W niniejszej książce wszystkie nazwy funkcji, które pojawią się w celu odpowiedniej czytelności, w tekście będą pozbawione sufiksów, w prezentowanych przykładach występować będzie odpowiedni dla prezentowanego przypadku wariant.

Sam OpenGL ze względu na swoją niskopoziomowość nie oferuje żadnych typów złożonych. Dość szybko przekonacie się, że dobrze jest posiadać możliwość opisu obiektu na wyższym poziomie abstrakcji. Dobrze jest więc zawczasu przygotować sobie proste klasy ułatwiające nam życie. w przyszłości zostaną uzupełnione o kilka dość użytecznych metod, ale na to przyjdzie czas w kolejnych rozdziałach.

Listing 2.1. Deklaracja klas MYGLvertex i MYGLtriangle

```
 4 class MYGLvertex {
       public:
 6
       MYGLvertex();
 8     MYGLvertex(GLfloat px, GLfloat py, GLfloat pz);
       void setVertex(GLfloat px, GLfloat py, GLfloat pz);
10
       GLfloat x;
12     GLfloat y;
       GLfloat z;
14 };

16 class MYGLtriangle {
       public:
18
       MYGLtriangle();
20     MYGLtriangle
           (MYGLvertex *pv1, MYGLvertex *pv2, MYGLvertex *pv3);
22     void setTriangle
           (MYGLvertex *pv1, MYGLvertex *pv2, MYGLvertex *pv3);
24     void calculateNormal (void);

26     private:

28     MYGLvertex *v1;
       MYGLvertex *v2;
30     MYGLvertex *v3;
       MYGLvertex n;
32 };
```

Jeżeli chodzi o same definicje metod dla klas z listingu 2.1 to w przypadku *MYGLvertex* konstruktor wywołuje tylko funkcję składową `setVertex()`, która przepisuje wartości kolejnych współrzędnych do prywatnych atrybutów x, y i z. Analogicznie rzecz się ma dla drugiej z zaprezentowanych klas, *MYGLtriangle*, z tym zastrzeżeniem, że tu przekazywane są wskaźniki do trzech kolejnych wierzchołków i w tej formie są przechowywane (por. listing 2.2). Musimy o tym pamiętać, tworząc aplikację, aby nie nabawić się problemów z obsługą pamięci.

Listing 2.2. Definicja konstruktora klasy MYGLtriangle

```
   void MYGLtriangle::setTriangle
26     (MYGLvertex *pv1, MYGLvertex *pv2, MYGLvertex *pv3) {
       v1 = pv1;
28     v2 = pv2;
       v3 = pv3;
30     calculateNormal();
   }
```

Metoda `calculateNormal()`, mimo że bardzo użyteczna, na ten moment pozostanie pusta, wrócimy jednak do niej.

Rozdział 3

Integracja i interakcja ze środowiskiem

3.1. GLUT **16**
3.1.1. Pierwszy program 17
3.1.2. Podwójny bufor 21
3.1.3. Obsługa klawiatury 22
3.1.4. Obsługa myszy 24
3.1.5. Wyświetlanie napisów 27
3.1.6. Generacja i obsługa menu 31
3.2. Inne API **35**

Pierwotnie dostęp do kontekstu renderowania OpenGL i jakakolwiek interakcja ze strony użytkownika możliwa była w przypadku systemów zgodnych ze standardem POSIX za pomocą biblioteki GLX[4], a w przypadku systemów Microsoftu, MFC. To pierwsze rozwiązanie powstało pod kątem konkretnych zastosowań i miało dużo pozytywnych cech. Między innymi zawierało kompletne mechanizmy umożliwiające na rozdzielenie procesu renderowania sceny i jej wyświetlania pomiędzy klientem i serwerem. Niestety taka, a nie inna, architektura użyta przy projektowaniu tej biblioteki zaowocowała dość dużym stopniem komplikacji ewentualnych aplikacji pisanych z jej użyciem.

3.1. GLUT

Jeżeli postanowiliście się zapoznać ze sposobami przygotowania aplikacji, która będzie korzystać z biblioteki OpenGL z wykorzystaniem natywnego API systemu X/Window, czy też Microsoft Windows, to jest to zadanie karkołomne. Musimy zadbać o wygenerowanie odpowiednich uchwytów, ustawić wartość kilkudziesięciu atrybutów oraz zbudować obsługę urządzeń wejściowych. Dlatego opracowana przez Marka Kilgarda biblioteka była niewątpliwym przełomem i pozwoliła na bezstresowe przygotowanie środowiska, zarządzanie wieloma oknami czy też praca w trybie pełnoekranowym w intuicyjny i banalnie prosty sposób[5]. Ustandaryzowana została też obsługa zdarzeń generowanych przez klawiaturę i urządzenia wskazujące. Możliwe stało się nawet budowanie menu, czy obsługa fontów, co w znacznym stopniu wykraczało poza możliwości czystego OpenGL. Dodatkowo, tworzenie aplikacji w pełni przenośnej między systemami operacyjnymi było znacznie ułatwione ponieważ kod, w większości przypadków, mógł zostać skompilowany bez większych zmian tak na X/Windows jak i rodzinę Microsoft Windows.

Niestety GLUT, czyli OpenGL Utility Toolkit, nie jest remedium na wszystkie problemy związane z OpenGL, co więcej, sam wprowadza pewne ograniczenia, które znacząco wpływają na brak elastyczności w tworzeniu aplikacji z niego korzystających. Dodatkowo, mimo że mamy dostępny kompletny kod biblioteki i doczekaliśmy się jej portów na inne systemy operacyjne, jej dalszy rozwój został zatrzymany na poziomie wersji 3.7 pod koniec lat dziewięćdziesiątych zeszłego stulecia. Przyczyna tej sytuacji leży w jej licencjonowaniu, która dopuszcza bezpłatne użycie, ale ponieważ nie jest otwarto-źródłowa nie pozwala na dokonywanie zmian i rozwijanie wersji pochodnych.

Okazało się jednak, że duża popularność GLUT i niezaprzeczalne jego zalety skłoniły osoby z kręgów wolnego i otwartego oprogramowania do

stworzenia bibliotek, które kopiują nazewnictwo tak funkcji, zmiennych i stałych jak i strukturę oraz funkcjonalność oryginału przy kodzie źródłowym napisanym praktycznie od zera. Co więcej, zarówno FreeGLUT, jak i zamrożony obecnie OpenGLUT, usuwają błędy i niedociągnięcia oryginału w tym obarczone poważnym błędem koncepcyjnym zachowanie głównej pętli.

3.1.1. Pierwszy program

Napisanie programu korzystającego z OpenGL z wykorzystaniem GLUT jest banalne i składa się z kilku prostych kroków. Po dodaniu dyrektywy *include* włączającej standardowy nagłówek biblioteki `glut.h` Zaczynamy od przygotowania środowiska, wywołując w funkcji main naszego programu funkcję inicjalizującą, następnie ustawiając tryb wyświetlania i ustalając rozmiary okna (listing 3.1). w przypadku GLUT, analogicznie jak ma to miejsce w OpenGL, wszystkie funkcje składowe biblioteki mają w nazwie unikalny przedrostek, w tym wypadku *glut*.

Listing 3.1. Minimalistyczna funkcja main() programu w C++ korzystającego z GLUT

```
1 #include <GL/glut.h>

3 void drawScene(void){
      glClear(GL_COLOR_BUFFER_BIT);
5     glBegin(GL_QUADS);
          glVertex3f(-0.5,-0.5,0.0);
7         glVertex3f(0.5,-0.5,0.0);
          glVertex3f(0.5,0.5,0.0);
9         glVertex3f(-0.5,0.5,0.0);
      glEnd();
11    glFlush();
  }
13
  int main(int argc, char **argv) {
15    glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_SINGLE);
17    glutInitWindowSize(600,600);
      glutCreateWindow("Pierwszy program w~GLUT");
19    glutDisplayFunc(drawScene);
      glutMainLoop();
21 }
```

Za ustalenie początkowych parametrów wyświetlania odpowiedzialne są trzy funkcje. Pierwsza, `glutInitWindowSize()`, przyjmująca dwa parametry całkowite opisujące wielkość okna wyrażone jako szerokość i wysokość w pikselach. Niewywołanie tej funkcji w procesie inicjalizacji zostawi wartość domyślną wynoszącą dla obu wymiarów 300. Drugą spośród nich jest

glutInitWindowPosition() i wskazuje ona położenie na ekranie. Domyślnie jest pomijana i oznacza to zostawienie decyzji w tej kwestii systemowi operacyjnemu. Ostatnią i mającą znaczący wpływ na sposób zachowania się kontekstu renderowania jest glutInitDisplayMode(), która przyjmuje jeden argument będący maską bitową określającą tak wybrany tryb obsługi koloru, działania bufora wyświetlania i obsługi buforów dodatkowych (tablica 7.1).

Tabela 3.1. Tryby funkcji glutInitDisplayMode()

Nazwa trybu	Działanie
GLUT_RGBA	Ustawia wyświetlanie okna w trybie 32 bitowym z przezroczystością. Domyślny.
GLUT_RGB	Tożsamy z GLUT_RGBA.
GLUT_INDEX	Włączenie trybu koloru indeksowanego (paleta).
GLUT_SINGLE	Pojedynczy bufor wyświetlania. Domyślny.
GLUT_DOUBLE	Podwójny bufor wyświetlania umożliwiający użycie przełączania stron.
GLUT_ACCUM	Okno z buforem akumulacyjnym.
GLUT_ALPHA	Ustawienie alokacji składowej kanału przezroczystości w modelu koloru dla okna. Samo użycie GLUT_RGBA nie alokouje potrzebnej pamięci, a jedynie wybiera tryb.
GLUT_DEPTH	Włączenie bufora głębokości.
GLUT_STENCIL	Włączenie bufora szablonów.
GLUT_MULTISAMPLE	Uruchomienie wielokrotnego próbkowania w sytuacji gdy rozszerzenie odpowiedzialne za jego obsługę jest dostępne.
GLUT_STEREO	Włączenie trybu stereoskopowego.
GLUT_LUMINANCE	Wykorzystuje składową czerwoną koloru do indeksowania palety domyślnie zainicjalizowanej odcieniami szarości. Tryb może być nie wspierany przez część implementacji OplenGL.

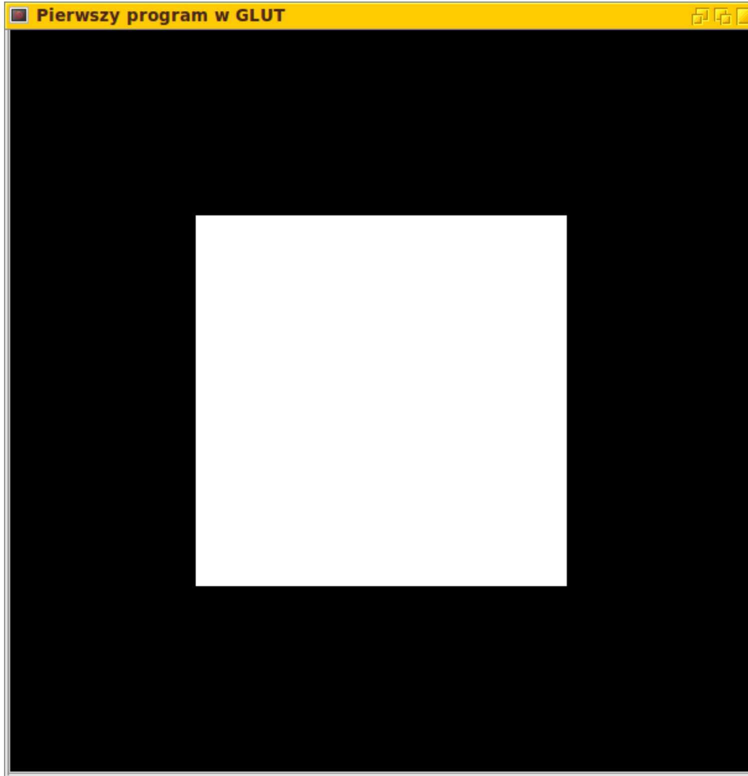
Kolejnym krokiem po konfiguracji głównego okna jest jego wyświetlenie, co osiągamy poprzez wywołanie funkcji `glutCreateWindow()`, która jako jedyny parametr przyjmuje jego nazwę, a zwraca identyfikator okna w postaci liczby. Okno po wyświetleniu nie zawiera kontekstu renderowania, dopóki nie przypiszemy mu funkcji odpowiedzialnej za przerysowanie sceny i nie uruchomimy głównej pętli za pomocą `glutMainLoop()`. Jak popatrzymy na kod w listingu 3.1, to znajdziemy tam definicję funkcji `drawScene()` zawierającą w sobie wywołania funkcji OpenGL. To właśnie ona wykonywana jest w ramach głównej pętli, musi być bezparametrowa i nie zwracać żadnej wartości. Oczywiście, jej nazywa może być dowolna, o ile jest dopuszczalna przez standard języka C, ponieważ wskazanie na nią na potrzeby GLUT odbywa się z użyciem `glutDisplayFunc()`.

Teraz, o ile mamy zainstalowanego GLUT, lub jego klony, możemy przystąpić do kompilacji programu. w wariancie minimalistycznym, jeżeli nie korzystacie z jakiegoś środowiska rozwojowego, możecie po prostu wywołać z linii poleceń kompilator, podając wskazania na konieczne do dołączenia w tym wypadku biblioteki poprzez parametry *-l*. w Linuksie, z którego korzystam, będzie to na przykład wyglądało tak:

```
g++ first_glut_program.c -lGL -lglut
```

W wyniku kompilacji i konsolidacji, o ile posiadamy zainstalowane odpowiednie pakiety, otrzymamy plik `a.out`, który po uruchomieniu powinien wyświetlić okno z wygenerowanym białym prostokątem na czarnym tle.

Jeżeli zaczniemy się bawić w zmienianie rozmiaru okna, okaże się, że nie jest tak dobrze, jakby się pierwotnie wydawało. Proporcje wyświetlonego kwadratu podążają za oknem, co nie powinno mieć miejsca. Zaradzić temu można, tworząc funkcję (listing 3.2), która będzie mówiła kontekstowi OpenGL, jakich ma dokonać zmian, żeby obraz był wyświetlany poprawnie. Niestety, na obecnym etapie większość zawartego w niej kodu pozostawię nieskomentowane, ale jeszcze do nich powrócę w rozdziale 7. To, na co warto w tej chwili warto zwrócić uwagę, to linijki, w których występuje funkcja `glOrtho()`, która nie tylko odpowiedzialna jest za włączenie obsługi rzutowania ortogonalnego, ale również definiuje prostopadłościan obcinania dla niego. Teraz odpowiednio modyfikując proporcje poszczególnych boków, dokonujemy korekty proporcji generowanej sceny w zależności od proporcji okna, w którym ma miejsce wyświetlanie. Analogicznie można ten efekt osiągnąć w rzutowaniu perspektywicznym, podając odpowiednio wyliczoną wartość stosunku szerokości do wysokości obrazu jako parametr *aspect* funkcji `gluPerspective()`.



Rysunek 3.1. Okno programu z listingu 3.1

Listing 3.2. Funkcja modyfikująca proporcje obrazu przy zmianie wielkości okna

```
14 void reShape(int w, int h) {
       if(w==0)
16         w=1;
       if(h==0)
18         h=1;
       glMatrixMode(GL_PROJECTION);
20     glLoadIdentity();
       glViewport(0, 0, w, h);
22     if(w<h)
           glOrtho(-1.0f,1.0f,-1.0f*h/w,1.0f*h/w,1.0f,-1.0f);
24     else
           glOrtho(-1.0f*w/h,1.0f*w/h,-1.0f,1.0f,1.0f,-1.0f);
26     glMatrixMode(GL_MODELVIEW);
       glLoadIdentity();
28 }
```

Gotowa funkcja zmiany proporcji wyświetlanego obrazu zostaje dodana do obsługi w ramach głównej pętli GLUT, w sposób analogiczny, jak miało to miejsce z funkcją przerysowującą scenę, tyle, że dokonuje się tego za pośrednictwem `glutReshapeFunc()`. Należy tylko pamiętać, że musimy zrobić to przed wywołaniem `glutMainLoop()`.

3.1.2. Podwójny bufor

W przypadku gdy mamy do czynienia z animowanymi scenami składającymi się z więcej niż kilku prostych elementów i chcemy uniknąć migotania wynikającego z przerysowywania sceny najlepszym możliwym rozwiązaniem jest użycie mechanizmu przełączania stron. w OpenGL możliwe jest to dzięki użyciu dwóch buforów służących rysowaniu i gdy jeden jest wyświetlany, na drugim, niewidocznym, ma miejsce przerysowanie sceny. Po zakończeniu procesu generacji obrazu w buforze tylnym następuje jego wyświetlenie. Użycie tej techniki w GLUT jest proste i sprowadza się do dwóch kroków. Pierwszym jest ustawienie dla okna renderingu opcji GLUT_DOUBLE w wywołaniu `glutInitDisplayMode()`. Drugim jest umieszczenie w funkcji odpowiedzialnej za przerysowanie naszej sceny funkcji `glutSwapBuffers()`. Dobra praktyka mówi, że kod OpenGL opisujący scenę, powinien być zgrupowany razem, a przełączenie stron zamyka taki blok. w momencie jej wywołania niejawnie ma miejsce wywołanie `glFlush()`, więc możemy spokojnie pominąć je w naszym kodzie oraz gdy bufor przedni zostaje przesunięty na tył, stan jego jest nieokreślony. Samo przełączenie ma miejsce raczej w trakcie najbliższego odświeżenia ekranu niż natychmiast, co ma zagwarantować niewystąpienie efektu ciętego obrazu.

Listing 3.3. Zmodyfikowana funkcja rysowania sceny z listingu 3.1

```
   void drawScene(void){
 4     static int rot=0;
       glClear(GL_COLOR_BUFFER_BIT);
 6     glPushMatrix();
       glRotatef(rot++,0.0,0.0,1.0);
 8     glBegin(GL_QUADS);
           glVertex3f(-0.5,-0.5,0.0);
10         glVertex3f(0.5,-0.5,0.0);
           glVertex3f(0.5,0.5,0.0);
12         glVertex3f(-0.5,0.5,0.0);
       glEnd();
14     glPopMatrix();
       glutSwapBuffers();
16 }
```

Jak widać na listingu 3.3 w naszej funkcji pojawiło się kilka nowych linijek. To one uczynią ze statycznego obrazu animację. o ile nawet nie znając OpenGL na podstawie nazwy możecie zgadnąć, że `glRotatef()` jest odpowiedzialne za dokonanie obrotu o określony kąt, to znaczenie funkcji `glPushMatrix()` i `glPopMatrix()` jest co najmniej zagadkowe. Na ten moment wystarczy, że w tym przypadku odpowiedzialne one są za swego rodzaju unieruchomienie naszej wirtualnej kamery.

Okazuje się, niestety, że to nie wszystko, co trzeba zmienić. Program po kompilacji i uruchomieniu nadal wyświetla nieruchomy obraz. Przyczyna tego stanu leży w fakcie, że nasz program nie otrzymuje żadnych zdarzeń, które powodowałyby zmianę jego stanu i ponowne renderowanie sceny. Rozwiązań jest kilka, ale najprostszym będzie wskazanie poprzez `glutIdleFunc()` na funkcję, którą wywołuje pętla główna, gdy nic się nie dzieje. w naszym wypadku jest to po prostu funkcja przerysowania `drawScene()` (listing 3.4).

Listing 3.4. Zmodyfikowana funkcja główna

```
34 int main(int argc, char **argv) {
       glutInit(&argc, argv);
36     glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
       glutInitWindowSize(600,600);
38     glutCreateWindow("Pierwsza animacja w~GLUT");
       glutIdleFunc(drawScene);
40     glutDisplayFunc(drawScene);
       glutReshapeFunc(reShape);
42     glutMainLoop();
   }
```

3.1.3. Obsługa klawiatury

Do tej pory nasze oprogramowanie korzystające z biblioteki GLUT nie miało cech interaktywności. Możliwe było za jego pomocą tylko obsłużenie utworzenia okna i uruchomienia wyświetlania obrazu statycznego, a następnie animowanego. Teraz zajmiemy się rozbudową go o mechanizmy pozwalające wpływać na przebieg działania programu przy użyciu klawiatury lub myszki. Zaczniemy od tej pierwszej.

W przypadku GLUT zdarzenie polegające na wciśnięciu klawisza na klawiaturze może zostać obsłużone na kilka różnych sposobów. Najprostszy z nich polega na odczytaniu wygenerowanego kodu ASCII bez sprawdzania stanu klawiszy modyfikatorów takich jak *Shift*, *Control* lub *Alt*. Bardziej zaawansowany zaangażuje dodatkowo funkcję `glutGetModifiers()`, co pozwoli poza kodem znaku poznać wartości flag, odpowiadających nazwą wspomnianym klawiszom, `GLUT_ACTIVE_SHIFT`, `GLUT_ACTIVE_CTRL` i `GLUT_ACTIVE_ALT`. Jeżeli potrzebna jest nam z jakichś powodów obsługa klawiszy

specjalnych, takich jak *F1-F12*, *Insert*, strzałek, czy tym podobnych, jest to też możliwe.

Analogicznie jak w przykładach przedstawionych do tej pory, konieczne jest zarejestrowanie w ramach pętli głównej GLUT funkcji opisujących sposób zachowania się programu w momencie zaistnienia zdarzenia, tym razem związanego z klawiaturą. Sama rejestracja dla kodów ASCII odbywa się z wykorzystaniem funkcji `glutKeyboardFunc()`, dla klawiszy specjalnych zaś `glutSpecialFunc()`. Opis obsługi zawarty jest w definicjach odpowiednich funkcji widocznych na listingach 3.5 i 3.6.

W pierwszej kolejności utworzymy funkcję, która będzie włączać i wyłączać poszczególne składowe koloru, gdy naciśnięte będą odpowiadające im klawisze z inicjałami nazwy w języku angielskim (r, g i b). Chcemy, aby prawidłowa reakcja miała miejsce niezależnie od tego czy włączony jest *Caps Lock* lub wciśnięty *Shift*. Na listingu 3.5 widać, że rozwiązane to zostało w sposób najprostszy z możliwych - poprzez kontrolę czy wartość kodu przekazywana do obsługi zdarzenia jest minuskułą czy majuskułą interesującego nas znaku.

Listing 3.5. Funkcja obsługi klawiatury w oparciu o zwracane kody ASCII

```
   void kbEvent(unsigned char key, int x, int y) {
38     switch(key) {
           case 'r':
40         case 'R':
               (r == 0.0) ? (r = 1.0) : (r = 0.0);
42             break;
           case 'g':
44         case 'G':
               (g == 0.0) ? (g = 1.0) : (g = 0.0);
46             break;
           case 'b':
48         case 'B':
               (b == 0.0) ? (b = 1.0) : (b = 0.0);
50             break;
       }
52 }
```

Zmienne `r`, `g` i `b` należy zadeklarować i zainicjować globalnie przed ich użyciem, co może nie jest bardzo eleganckie i dydaktyczne, ale skuteczne,

Następnie zajmiemy się funkcją, która będzie obsługiwała klawisze specjalne. Jej celem będzie ustawienie koloru białego dla wyświetlanego kwadratu przy naciśnięciu *F1*, szarego przy *F2* i czarnego przy *F3*.

Listing 3.6. Funkcja obsługi znaków specjalnych klawiatury

```
54 void kbEventSpecial(int key, int x, int y) {
       switch(key) {
```

```
56          case GLUT_KEY_F1:
                r = g = b = 1.0;
58              break;
            case GLUT_KEY_F2:
60              r = g = b = 0.5;
                break;
62          case GLUT_KEY_F3:
                r = g = b = 0.0;
64              break;
        }
66 }
```

Gdy mamy już gotowe funkcje, które obsłużą nam zdarzenia klawiatury, dokonujemy ich rejestracji w GLUT.

```
76 glutKeyboardFunc(kbEvent);
   glutSpecialFunc(kbEventSpecial);
```

Konieczne jest jeszcze uzupełnienie naszego kodu o elementy, które pozwolą nam na zmianę koloru kwadratu. Tutaj, podobnie jak poprzednio, musicie uwierzyć na słowo, że tak się to realizuje. Zmiana jest niewielka i polega na umieszczeniu wewnątrz ciała funkcji `drawScene()` bezpośrednio przed `glBegin(GL_QUADS)` linijki z następującą zawartością:

```
10 glColor3f(r,g,b);
```

Warto zwrócić uwagę, że do funkcji obsługi zdarzeń klawiatury przekazywana poza informacją o wybranym klawiszu jest również informacja o bieżącej pozycji kursora myszy względem okna, co w pewnych sytuacjach może okazać się przydatne.

3.1.4. Obsługa myszy

Po tym, jak zapoznaliśmy się z relatywnie prosto zrealizowaną obsługa klawiatury, przyszedł czas na mysz. Nie da się ukryć, że w środowisku graficznym ten sposób komunikacji jest jednym z podstawowych i na tyle mocno zakorzenionych, że wiele nowych rodzajów urządzeń wskazujących jest na poziomie sterowników tłumaczona do poziomu komunikatów myszy czasem uzupełnionych o dodatkowe informacje, jak na przykład siła nacisku. Tak ma to miejsce w przypadku trackballa, trackpada, touchpada, tabletów graficznych, a nawet ekranów dotykowych.

W GLUT cała interakcja realizowana jest poprzez implementację odpowiednich funkcji zajmujących się obsługą określonych grup zdarzeń, które następnie zostają skojarzone z samymi zdarzeniami z wykorzystaniem funkcji rejestrujących. w tym przypadku będzie nie inaczej, różnica polegać

na rozdzieleniu komunikatów na większa liczbę grup niż miało to miejsce w przypadku klawiatury.

Pierwsza grupa zdarzeń komunikowanych przez GLUT i możliwych tym samym do obsłużenia są naciśnięcia i puszczenia poszczególnych klawiszy myszy. Odpowiedzialna za przekazanie do pętli głównej wskazania na funkcję obsługi zdarzenia tego typu jest `glutMouseFunc()`. w ramach komunikatu zdarzenia otrzymujemy wartość czterech parametrów. Dwa z nich, jak można się spodziewać, to wskazania pozycji myszy w ramach okna. Pozostałe informują nas, który klawisz wywołał zdarzenie, oraz jaki jest jego stan. w tym pierwszym przypadku zdefiniowane są następujące stałe: `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` oraz `GLUT_RIGHT_BUTTON` odpowiednio dla lewego, środkowego i prawego klawisza. Stany możliwe są tylko dwa, gdy nastąpiło naciśnięcie klawisza jest nim `GLUT_DOWN` i `GLUT_UP` w przeciwnym razie.

Listing 3.7 przedstawia implementację obsługi zdarzenia polegającego na naciśnięciu klawisza myszy i powodującego, że każde kolejne kliknięcie zmienia kolor kwadratu na jego dopełnienie. Gdybyśmy pominęli sprawdzanie warunku z linii 70, każdorazowe puszczenie lewego klawisza przywracałoby kolor pierwotny.

Listing 3.7. Funkcja obsługi naciśnięcia lewego klawisza myszy

```
68 void mouseButton(int button, int state, int x, int y) {
       if(button == GLUT_LEFT_BUTTON) {
70         if(state == GLUT_DOWN) {
               r = 1.0 - r;
72             g = 1.0 - g;
               b = 1.0 - b;
74         }
       }
76 }
```

Na grupę drugą zdarzeń generowanych przez mysz składają się jej ruchy. Ich obsługa jest rejestrowana w GLUT za pomocą dwóch funkcji w zależności od tego, czy którykolwiek klawisz myszy został naciśnięty i przytrzymany, czy też nie. w tym pierwszym przypadku korzystać będziemy z `glutPassiveMotionFunc()` w drugim zaś z `glutMotionFunc()`. Obie z wymienionych funkcji przekazują tylko informację o położeniu kursora wewnątrz okna.

O ile do tej pory uzupełnialiśmy o kolejne funkcje jeden program, to w następnym przykładzie żeby poszczególne elementy nie kolidowały, konieczne jest usunięcie tych fragmentów, które są odpowiedzialne za obsługę klawiatury i myszy. Umieszczamy zaś w nim kod z listingu 3.8, którego zadaniem jest zmiana koloru kwadratu lub dopełnienia tego koloru w przypadku

przytrzymanego dowolnego klawisza myszy, w sposób płynny w zależności od położenia kursora nad oknem.

Jeżeli poza informacją o samym fakcie naciśnięcia i przytrzymania klawisza myszy potrzebna jest informacja o tym, który z klawiszy jest odpowiedzialny za wejście w ten stan pętli głównej GLUT konieczne jest użycie odpowiednich flag ustawianych przez funkcję rejestrowaną za pośrednictwem `glutMouseFunc()`.

Listing 3.8. Funkcje obsługi ruch myszy

```
   void mouseMove(int x, int y) {
38     int w, h;
       w = glutGet(GLUT_WINDOW_WIDTH);
40     h = glutGet(GLUT_WINDOW_HEIGHT);
       r = x/(float)(w);
42     b = y/(float)(h);
       g = 0.0;
44 }

46 void mouseButtonAndMove(int x, int y) {
       int w, h;
48     w = glutGet(GLUT_WINDOW_WIDTH);
       h = glutGet(GLUT_WINDOW_HEIGHT);
50     r = 1.0 - x/(float)(w);
       b = 1.0 - y/(float)(h);
52     g = 1.0;
   }
```

W kodzie listingu 3.8 dodatkowo występuje funkcja `glutGet()`, która służy do pobierania informacji o parametrach środowiska. w niniejszym przykładzie sprawdzamy wartość wysokości i szerokości okna w pikselach, co umożliwia nam normalizację wartości składowych poszczególnych kanałów koloru kwadratu.

Ostatnia, trzecia grupa, to kontrola czy mysz znajduje się w obrębie okna. Służy do tego jedna funkcja rejestrująca obsługę `glutEntryFunc()`, która przekazuje jeden parametr mogący przyjmować wartości `GLUT_LEFT`, gdy kursor znalazł się poza oknem i `GLUT_ ENTERED`, gdy jest ponownie w jego obrębie. Przykładowa implementacja obsługi zawarta jest w listingu 3.9.

Listing 3.9. Funkcja obsługi opuszczenia obszaru okna przez kursor myszy

```
    void mouseLeft(int state) {
56     if(state == GLUT_LEFT) {
           r = 1.0;
58         g = 0.0;
           b = 0.0;
```

```
60        }
    }
```

3.1.5. Wyświetlanie napisów

Istotnym elementem prawie każdego oprogramowania jest możliwość przekazywania komunikatów tekstowych użytkownikowi. OpenGL ze względu na swoją niskopoziomowość natywnie nie oferuje nam takich mechanizmów. Możliwe jest to jednak do zrealizowania na wiele innych sposobów, poczynając od prostego wypisywania wiadomości w oknie konsoli po korzystanie z mechanizmów dostępnych za pośrednictwem API systemu. Niestety, pierwsze rozwiązanie jest po prostu złe, drugie trochę lepsze, ale utrudnia przenośność gotowego programu na inne systemy operacyjne. Dlatego najkorzystniejsze wydaje się użycie w tym celu funkcji dostarczanych przez GLUT. Tak się jednak składa, że tu też nie wszystko jest takie, jak byśmy tego chcieli, brakuje na przykład znaków narodowych, tekst musi być kodowany z użyciem iso 8859-1, a liczba dostępnych krojów czcionek jest bardzo ograniczona (tabela 3.2). Za to mamy możliwość pisania bezpośrednio po oknie, w którym ma miejsce rendering naszej sceny, co niewątpliwie jest dużą zaletą tego rozwiązania.

Do wykorzystania mamy dwa różne typy czcionek, które różnią się między sobą tak sposobem użycia, jak i ich potencjalnym wykorzystaniem. Czcionki bitmapowe mają stały rozmiar na ekranie i nie podlegają mechanizmom obrotu i skalowania, a jedynie przesunięciom. Jest to korzystne w sytuacji, gdy zależy nam na wyświetlaniu informacji o bieżących parametrach środowiska w określonym miejscu w sposób zawsze czytelny. Teksty napisane w oparciu o znaki wektorowe zachowują się jak każdy inny element sceny i świetnie się nadają do wskazywania i opisu obiektów z zachowaniem odpowiednich proporcji i perspektywy.

Niestety, twórcy GLUT nie pokusili się o zapewnienie rozwiązania umożliwiającego wyświetlenie całego łańcucha znaków, a jedynie dali funkcję do wyświetlania pojedynczych glifów, poradzimy sobie jednak z tym, dodając własne. w przypadku czcionek bitmapowych (listing 3.10) korzystamy z funkcji `glutBitmapCharacter()`, która przyjmuje jako parametry zdefiniowaną stałą wskazującą konkretny, wbudowany zestaw znaków i kod znaku, który ma zostać wyświetlony. Jak widać, w kodzie pokusiliśmy się też o ustalenie wartości domyślnej dla fontu, co nie tylko jest dobrą praktyką, ale w wielu wypadkach oszczędza nasz czas jako programisty.

Listing 3.10. Funkcja pisania łańcucha znaków bitmapowych

```
void myglutDrawBitmapString
```

Tabela 3.2. Czcionki natywnie dostępne w GLUT

Nazwa	Wysokość	Proporcjonalny	Szeryfowy	Bitmapowy
GLUT_STROKE_ROMAN	85,72 u	x		
GLUT_STROKE_MONO_ROMAN	85,72 u			
GLUT_BITMAP_9_BY_15	15 px			x
GLUT_BITMAP_8_BY_13	13 px			x
GLUT_BITMAP_TIMES_ROMAN_10	10 pt	x	x	x
GLUT_BITMAP_TIMES_ROMAN_10	24 pt	x	x	x
GLUT_BITMAP_HELVETICA_10	10 pt	x		x
GLUT_BITMAP_HELVETICA_12	12 pt	x		x
GLUT_BITMAP_HELVETICA_18	18 pt	x		x

```
8       (char *text, void *font = GLUT_BITMAP_9_BY_15) {
        for (int i=0; i<strlen(text); i++)
10          glutBitmapCharacter(font, text[i]);
    }
```

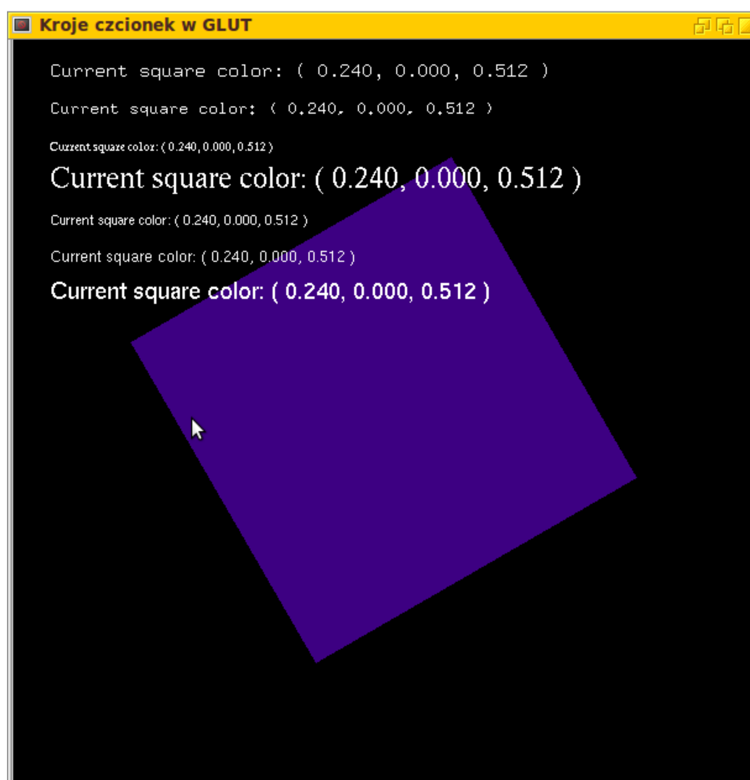
Analogicznie (listing 3.11) wygląda funkcja, która jest odpowiedzialna za wyświetlenie linijki tekstu z użyciem krojów wektorowych, różnica polega na użyciu w tym wypadku funkcji GLUT `glutStrokeCharacter()`.

Do poprawnego działania zaprezentowanych funkcji konieczne jest dołączenie nagłówka zawierającego funkcję `strlen()`, czyli *string.h.* w przykładowych programach dodatkowo użyta został nagłówek *stdio.h* co znacznie ułatwiło nam życie przy generowaniu komunikatu informującego o obecnie ustawionych wartościach poszczególnych składowych koloru. Można to oczywiście zrealizować na wiele innych sposobów, ale ponieważ jest to, prawdopodobnie, najprostsza z metod, a sama książka jest o OpenGL, a nie programowaniu w C czy C++, pominę pozostałe w moich rozważaniach.

Listing 3.11. Funkcja pisania łańcucha znaków wektorowych

```
    void myglutDrawStrokeString
8       (char *text, void *font = GLUT_STROKE_ROMAN) {
        for (int i=0; i<strlen(text); i++)
10          glutStrokeCharacter(font, text[i]);
    }
```

Niestety, ze względu na charakter mechanizmu obsługi napisów w GLUT na barkach programisty leży odpowiedzialność za poprawną interpretację znaków formatujących występujących tekście. Nie jest to trudne do realizacji, ale wykracza poza tematykę książki. Jako podpowiedź powiem, że GLUT ma wśród swoich fukcji ma po dwie na każdy typ glifów, które zwracające informacje o szerokości tak poszczególnych liter jak i całych łańcuchów znaków. i tak, odpowiednio dla czcionek bitmapowych jest to `glutBitmapWidth()` i `glutBitmapLength()`, a dla wektorowych `glutStrokeWidth()` oraz `glutStrokeLength()`.



Rysunek 3.2. Różne kroje czcionek bitmapowych dostępne w GLUT

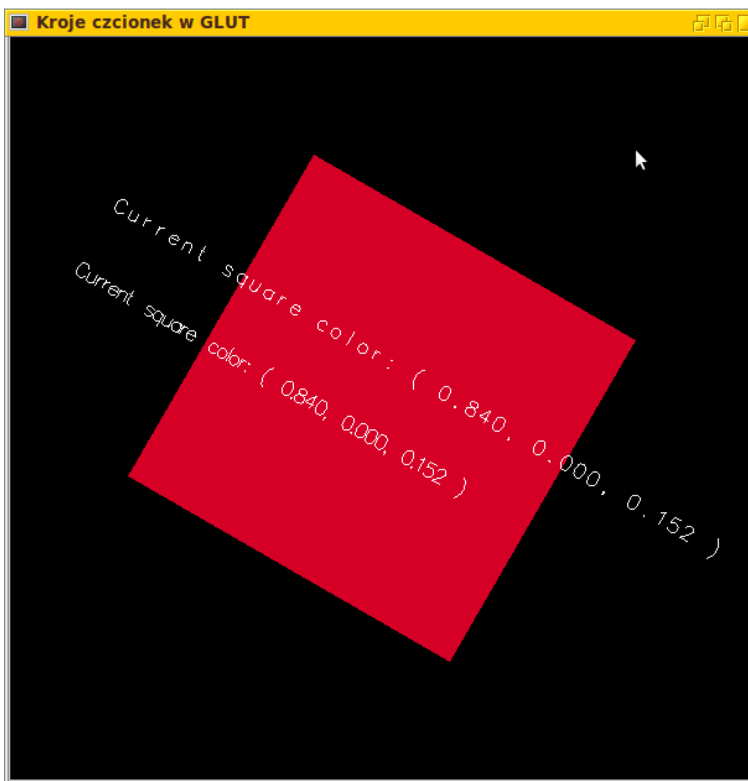
Wyświetlając tekst chcemy mieć kontrolę nad jego pozycją na ekranie. Ze względu na różną naturę czcionek bitmapowych i wektorowych w różny sposób jest ona również dla nich określana. w tym pierwszym przypadku najlepiej zdać się na funkcje `glRasterPos3f()` lub `glRasterPos2f()`. i tutaj uwaga, współrzędne podawane w funkcji nie odpowiadają pozycji w pikselach względem okna, w którym ma miejsce wyświetlanie, a współrzędnych wynikających z wybranego sposobu rzutowania i wybranego widoku sceny.

w naszych przykładach będzie to zakres od -1,0 do 1,0 dla współrzędnych x i y. Przykładowy fragment zaprezentowany jest na listingu 3.12

Listing 3.12. Fragment kodu programu odpowiedzialny za pozycjonowanie napisu bitmapowego

```
30      glRasterPos3f(-0.9,0.8,0);
        myglutDrawBittmapString
32          (colorinfo, GLUT_BITMAP_8_BY_13);
```

To, o czym należy pamiętać przy wyświetlaniu znaków z wykorzystaniem znaków bitmapowych to fakt, że jeżeli chcemy, aby nie zostały przykryte przez obiekty znajdujące się na scenie, muszą one zostać narysowane na samym końcu.



Rysunek 3.3. Wyświetlanie czcionek wektorowych w GLUT

W przypadku czcionek zapisanych wektorowo ich pozycjonowanie, skalowanie i obrót podlega tym samym zasadom, co wszystkie inne prymitywy budujące scenę. Zostanie to omówione dokładnie w rozdziale 5, a na razie wystarczy, żebyśmy pamiętali o tym, że linia dolna umieszczona została na

wysokości 33,33 jednostek, a linia górna na 119,05. Dodatkowo w wariancie nieproporcjonalnym `GLUT_STROKE_MONO_ROMAN` każdy glif ma szerokość 104,76 jednostki. Przykład wyświetlenia napisu umieszczonego na stałej pozycji i ze skorelowanym obrotem względem kwadratu prezentuje listing 3.13.

Listing 3.13. Fragment kodu programu odpowiedzialny za pozycjonowanie napisu wektorowego

```
38 glPushMatrix();
       glRotatef(rot,0.0,0.0,1.0);
40     glTranslatef(-0.9,0.1,0.0);
       glScalef(0.0004,0.0004,1);
42     myglutDrawStrokeString(colorinfo,
           GLUT_STROKE_MONO_ROMAN);
44     glPopMatrix();
   }
```

3.1.6. Generacja i obsługa menu

Fakt, że interakcja z systemem za pomocą myszy i klawiatury w wielu wypadkach jest zupełności wystarczająca, vide profesjonalne systemy montażu wideo, czy oprogramowanie CAD, to przeciętny użytkownik chętniej korzysta z odpowiedniego menu kontekstowego, z którego może wybrać interesujące go operacje. GLUT oferuje nam cały szereg rozwiązań, które pozwalają na budowanie menu tak głównych jak i podrzędnych i ich modyfikację w zależności od stanu aplikacji.

Jak za chwilę się przekonamy, funkcje obsługujące zdarzenia wyboru pozycji z menu są bardzo zbliżone do tych, które stworzyliśmy na potrzeby obsługi klawiatury. z tego prostego faktu wynika możliwość tworzenia tych funkcji w sposób zunifikowany, zapewniając spójną obsługę obu sposobów interakcji. Tutaj nie pójdziemy tą drogą, mając na względzie klarowność przekazu.

Menu, które zbudujemy, będzie funkcjonalnie odpowiadać naszej obsłudze programu z użyciem klawiatury, czyli będzie możliwe przełączanie wartości składowych koloru oraz skorzystaniem z trzech predefiniowanych odcieni szarości. Budowa każdego menu jest dwuetapowa. Pierwszy krok to stworzenie funkcji, która zawiera opis pozycji w nim zawartych. Pierwszy krok to wskazanie na funkcję obsługi zdarzenia, o której za chwilę, za pomocą `glutCreateMenu()`. Następnie, korzystając z `glutAddMenuEntry()`, dodajemy kolejne pozycje podając dwa parametry. Pierwszy odpowiada tekstowi, który zostanie wyświetlony na danej pozycji, drugi to wartość liczbowa wysyłana przy wybraniu jej przez użytkownika. Całość zamyka dowiązanie wywołania do wybranego klawisza myszy - `glutAttachMenu()`.

Listing 3.14. Definicja listy pozycji w menu głównym

```
    void createMenus() {
56      glutCreateMenu(menuEvents);
        glutAddMenuEntry("Czerwony", MENU_R);
58      glutAddMenuEntry("Zielony", MENU_G);
        glutAddMenuEntry("Niebieski", MENU_B);
60      glutAttachMenu(GLUT_RIGHT_BUTTON);
    }
```

Na listingu 3.14 możemy zauważyć, że korzystamy ze zdefiniowanych stałych *MENU_R*, *MENU_G* i *MENU_B* skojarzonych z poszczególnymi pozycjami. w tym przypadku rozwiązanie takie jest dobrą praktyką gdyż wartości te mogą być użyte kilka razy w kodzie i operowanie nimi jako nazwą zmniejsza ryzyko wystąpienia pomyłki.

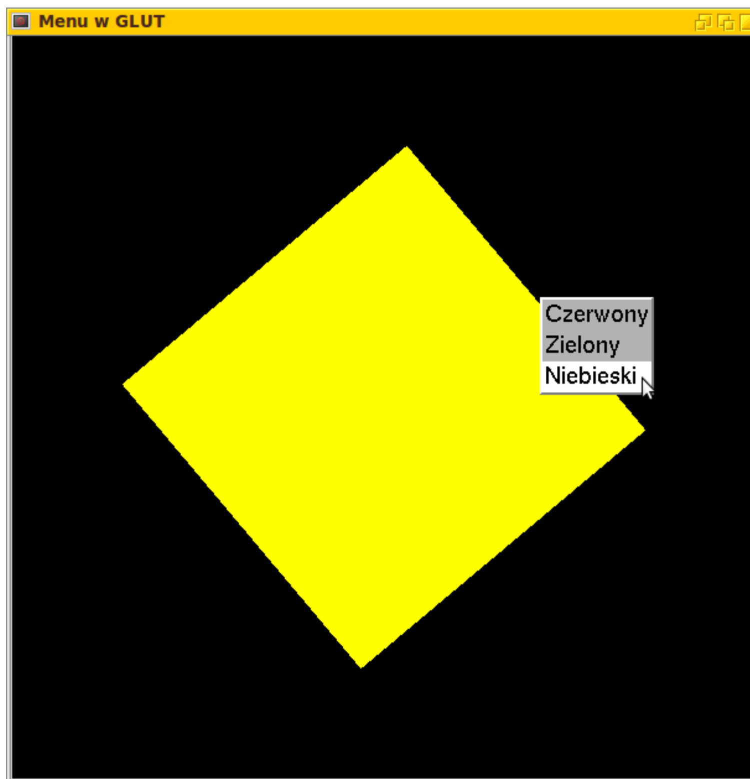
Funkcja obsługi zdarzenia (listing 3.15) do złudzenia przypomina swój odpowiednik z części poświęconej obsłudze klawiatury. Nic dziwnego, w końcu ich zadanie w naszym programie są tożsame.

Listing 3.15. Funkcja obsługi zdarzeń dla menu głównego

```
    void menuEvents(int option) {
42      switch (option) {
            case MENU_R:
44              (r == 0.0) ? (r = 1.0) : (r = 0.0);
                break;
46          case MENU_G:
                (g == 0.0) ? (g = 1.0) : (g = 0.0);
48              break;
            case MENU_B:
50              (b == 0.0) ? (b = 1.0) : (b = 0.0);
                break;
52      }
    }
```

Pozostaje wywołanie w funkcji głównej funkcji `createMenu()`, aby po uruchomieniu programu i kliknięciu prawym klawiszem myszy w obszarze okna programu zostało wyświetlone stosowne menu (rysunek 3.4).

Niestety, na razie nasze menu nie reaguje na kontekst. Dla każdej jego pozycji mamy tylko lakoniczne określenie koloru, podczas gdy dobrze byłoby mieć pełną wiedzę o to co wybranie poszczególnych z nich zmieni w stanie programu. Użyjemy do tego funkcji `glutChangeToMenuEntry()` i modyfikując poszczególne wywołania *case* w funkcji obsługi według schematu widocznego na listingu 3.16.

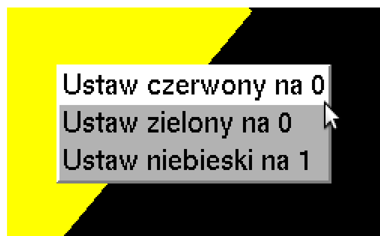


Rysunek 3.4. Menu główne w GLUT

Listing 3.16. Fragmnent zmodyfikowanej funkcji obsługi zdarzeń dla menu głównego

```
            case MENU_R:
44              if(r == 0.0) {
                    r = 1.0;
46                  glutChangeToMenuEntry
                        (MENU_R, "Ustaw czerwony na 0", MENU_R);
48              } else {
                    r = 0.0;
50                  glutChangeToMenuEntry
                        (MENU_R, "Ustaw czerwony na 1", MENU_R);
52              }
                break;
54 }
```

Taka modyfikacja, podobnie jak usunięcie poszczególnych pozycji za pomocą funkcji `glutRemoveMenuItem()` jest możliwe w dowolnym momencie działania programu.



Rysunek 3.5. Kontekstowe menu główne w GLUT

Kolejną funkcjonalnością oferowaną przez GLUT jest możliwość tworzenia dla każdego głównego menu dowiązanych menu niższego rzędu. Daje to sposobność do redukcji ilości pozycji możliwych do wyboru w pojedynczym dialogu poprzez grupowanie ich w oparciu o wybrane kryterium podobieństwa.

Obsługę zdarzeń dla każdego podmenu możemy włączyć do jednej wspólnej funkcji, lub, tak jak ma to miejsce w prezentowanym przykładzie (listing 3.17), porozdzielać po osobnych. To pierwsze podejście daje możliwość ewentualnego przemieszczania pozycji pomiędzy poszczególnymi menu niższego rzędu bez konieczności modyfikacji w funkcji obsługi, drugi zaś jest bardziej dydaktyczny i w sposób przejrzysty organizuje kod.

Listing 3.17. Funkcja obsługi zdarzeń dla menu podrzędnego

```
    void menuEventSpecial(int option) {
42      switch(option) {
            case MENU_WA:
44              r = g = b = 1.0;
                break;
46          case MENU_GR:
                r = g = b = 0.5;
48              break;
            case MENU_BL:
50              r = g = b = 0.0;
                break;
52      }
    }
```

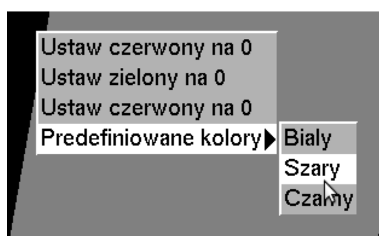
W GLUT opis kolejnych menu jest nie do końca intuicyjny, jeżeli jednak będziecie trzymać się kilku prostych zasad, nic złego nie powinno się wydarzyć. Wszystkie elementy menu składają się strukturę drzewiastą. Przy definiowaniu elementów składowych zawsze należy zaczynać menu będących liśćmi, potem węzły, a kończymy na korzeniu. Krawędzie opisywane są poprzez wartości zwracane za pośrednictwem funkcji `glutCreateMenu()` i ręcznie wstawiane do rodzica z użyciem `glutAddSubMenu()`. Poszczególne

pozycje przypisywane są do menu bieżącego, czyli, albo bezpośrednio je poprzedzającego, albo wskazanego z użyciem `glutSetMenu()`, w kolejności ich podawania.

Listing 3.18. Zmodyfikowana definicja listy pozycji w menu głównym

```
    void createGLUTMenus() {
106     int submenu = glutCreateMenu(menuEventSpecial);
        glutAddMenuEntry("Bialy",MENU_WA);
108     glutAddMenuEntry("Szary",MENU_GR);
        glutAddMenuEntry("Czarny",MENU_BL);
110     glutCreateMenu(menuEvent);
        glutAddMenuEntry("Ustaw czerwony na 0",MENU_R);
112     glutAddMenuEntry("Ustaw zielony na 0",MENU_G);
        glutAddMenuEntry("Ustaw niebieski na 0",MENU_B);
114     glutAddSubMenu("Predefiniowane kolory",submenu);
        glutAttachMenu(GLUT_RIGHT_BUTTON);
116 }
```

Założona w procesie projektowania i tworzenia GLUT niskopoziomowość oraz skupienie się na maksymalnym uproszczeniu mechanizmów obsługi zdarzeń wynikających z interakcji z użytkownikiem spowodowały, że zaimplementowany system budowy menu był wyjątkowo minimalistyczny. Nie zawarto w nim nawet uproszczonych kontrolek, o oknach dialogowych nawet nie wspominając i jeżeli potrzebne jest stworzenie bardziej rozbudowanego graficznego interfejsu rozwiązaniem jest skorzystanie z GLUI User Interface Library. Wprawdzie pewnym ograniczeniem może być konieczność tworzenia aplikacji obiektowej w C++ podczas gdy tak OpenGL, jak i GLUT umożliwiają również programowanie proceduralne w czystym C.



Rysunek 3.6. Użycie podmenu w GLUT

3.2. Inne API

Ze względu na dużą współcześnie popularność OpenGL praktycznie każda biblioteka, która umożliwia budowę graficznego interfejsu użytkownika

oferuje mniej lub bardziej rozbudowane mechanizmy wspierające tworzenie kontekstów OpenGL. w większości wypadków są one dostępne w standardowym zestawie kontrolek, jak ma to miejsce w *Qt*, czy *FLTK*. Istnieje też szereg bibliotek graficznych i wizualizacyjnych, które dają abstrakcyjne interfejsy, ale wewnętrzny potok przetwarzania i generacji sceny jest oparty na OpenGL. Warto w tym wypadku wspomnieć o *Visualization Toolkit (VTK)* `http://www.vtk.org/`, czy *Crystal Space 3D* `http://www.crystalspace3d.org`.

Rozdział 4

Prymitywy

4.1. Prymitywy OpenGL **38**
4.1.1. GL_POINTS 40
4.1.2. GL_LINES 40
4.1.3. GL_LINE_STRIP 41
4.1.4. GL_LINE_LOOP 41
4.1.5. GL_TRIANGLES 42
4.1.6. GL_TRIANGLE_STRIP 42
4.1.7. Biblioteki Tri Stripper i NvTriStrip 43
4.1.8. GL_TRIANGLE_FAN 43
4.1.9. GL_QUADS 44
4.1.10. GL_QUAD_STRIP 44
4.1.11. GL_POLYGON 45
4.2. Predefiniowane modele GLU **49**
4.3. Predefiniowane modele GLUT **53**

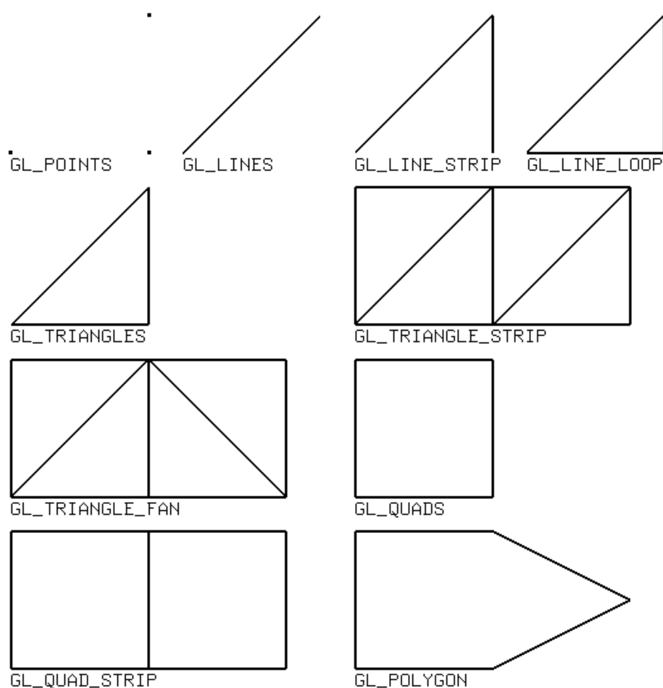
Rysunek 4.1. Herbatka u Szalonego Kapelusznika. ©*sir John Tenniel*

W tym rozdziale nie poświęcimy się jeszcze temu, jak sprawić możliwym budowanie złożonych, czasem nawet bardzo, modeli, które użyć będziemy mogli rozmieszczać w ramach tworzonej przez nas sceny. Dowiemy się za to o klockach, bez których nie byłoby możliwe zaistnienie czegokolwiek w wirtualnym świecie. Będzie więc między innymi o prymitywach, kwadrykach i bryłach, nie tylko platońskich. Na koniec zobaczymy, co ma imbryk, prawie taki jak w Alicji w Krainie Czarów, do... grafiki trójwymiarowej.

Każda, nawet najbardziej rozbudowana scena, składa się z określonej liczby podstawowych elementów, które odpowiednio połączone według opisu matematycznego dają zamierzony wynik. Mimo że dla grafiki trójwymiarowej przyjęło się, że jednostkami niejako atomowymi są punkt, linia, trójkąt i, w niektórych podejściach, także czworokąt, to współczesne biblioteki graficzne oferują nam również obiekty nieatomowe z nich złożone, a w swoim kształcie i formie na tyle uniwersalne, że mogą w znaczący sposób przyspieszyć akt kreacji. w niniejszym rozdziale zapoznamy się ze sposobem ich opisu w ramach sceny i potencjalnymi pułapkami czyhającymi na programistę OpenGL.

4.1. Prymitywy OpenGL

W ramach biblioteki OpenGL mamy tylko kilka podstawowych rodzai prymitywów, co niewątpliwie wynika z jej niskopoziomowości, konieczne jest jednak ich dogłębne poznanie ze względu na możliwość budowania w oparciu



Rysunek 4.2. Prymitywy dostępne w OpenGL

o nie praktycznie dowolnej formy, która jest możliwa do opisu w sposób numeryczny. Mogą to być zarówno obiekty wyprowadzone z modeli analitycznych, jak i informacja o położeniu wierzchołków wynikająca z algorytmów triangulacyjnych dla danych przestrzennych. Tu przykładem mogą być rekonstrukcje powierzchni z wykorzystanie algorytmu wędrującego sześcianu dla danych pochodzących z akwizycji obrazową aparaturą diagnostyczną czyli Tomografem Komputerowym, Rezonansem Magnetycznym i podobnymi.

Opis każdego z prymitywów w najprostszym przypadku składa się z lisy wierzchołków, którą rozpoczynamy od użycia funkcji `glBegin()` i kończymy `glEnd()`. Jako parametr przy otwarciu listy podajemy stałą wskazującą na typ prymitywu. Od tego będzie zależała ich dalsza interpretacja. Każdy z wierzchołków jest definiowany z użyciem funkcji `glVertex()` niezależnie od typu prymitywu.

4.1.1. GL_POINTS

Pierwszym z omawianych prymitywów jest punkt, który jak każdy z elementów składowych sceny może mieć przypisany kolor, o czym będzie mowa w przyszłości. Dodatkowo mamy możliwość zdefiniować rozmiar w pikselach wykorzystując w tym celu funkcję `glPointSize()`. Podawany parametr jest liczbą zmiennoprzecinkową, jednak jeżeli w OpenGL antyaliasing i wielokrotne próbkowanie jest wyłączone to wymiar zostanie obcięty do liczby całkowitej i wyświetlony w postaci kwadratu. Należy pamiętać, że na rozmiar wyświetlonego punktu nie ma wpływu jego odległość od obserwatora. w ramach jednej listy wierzchołków możemy jednorazowo podać wiele punktów do wyświetlenia (patrz listing 4.9).

Listing 4.1. Narysowanie trzech punktów

```
glBegin(GL_POINTS);
2     glVertex3f(-1.0, -1.0, 0.0);
      glVertex3f(1.0, -1.0, 0.0);
4     glVertex3f(1.0, 1.0, 0.0);
glEnd();
```

4.1.2. GL_LINES

Kolejnym z prymitywów są odcinki łączące punkty. w OpenGL jest ich aż trzy typy. Pierwszym są odcinki, które łączą dwa punkty. w liście wierzchołków każda z podanych par wygeneruje jeden. Jeżeli podana zostanie nieparzysta ich liczba, to ostatnia, niekompletna, para zostanie pominięta.

Odcinki, podobnie jak punkty, pozwalają na specjalne ich traktowanie. Mamy tu za pomocą funkcji `glLineWidth()` możliwość zdefiniowania jej szerokości w pikselach. Analogicznie do punktów wartość ta będzie obcinana do całkowitej przy nieaktywnym antyaliasingu i nie podlega skalowaniu. Dodatkowo możemy ustalić wzór bitowy, który będzie wykorzystany do rysowania linii. w tym celu konieczne jest wykonanie dwóch operacji. Pierwszą jest włączenie obsługi kreskowania przy użyciu funkcji `glEnable()` z parametrem `GL_LINE_STIPPLE`. Drugą zaś podanie współczynnika powtórzenia bitu przy generacji wzoru i ustawienie schematu, który może być zapisany tak w postaci binarnej, jak i przeliczony na szesnastkową. Korzystamy tu z `glLineStipple()`.

Można pokusić się o pytanie, po co taki mechanizm? Przy generacji wirtualnych światów wydaje się całkiem zbędny. Odpowiedź jest zdumiewająco prosta, nie samą rozrywką człowiek żyje, a OpenGL został zaprojektowany również z myślą o systemach inżynierskich w tym CAD, gdzie wzory linii mają niebanalne znaczenie.

schemat = 0xAAAA, skala = 1.0
schemat = 0xAAAA, skala = 3.0
schemat = 0xAAAA, skala = 5.0
schemat = 0xF0F0, skala = 1.0
schemat = 0xF0F0, skala = 3.0
schemat = 0xF0F0, skala = 5.0
schemat = 0xF0CC, skala = 3.0
schemat = 0xAB6C, skala = 3.0

Rysunek 4.3. Definiowanie wzory linii

Listing 4.2. Odcinek łączący dwa punkty

```
1 glBegin(GL_LINES);
      glVertex3f(1.0, 1.0, 0.0);
3     glVertex3f(-1.0, -1.0, 0.0);
  glEnd();
```

4.1.3. GL_LINE_STRIP

Czasem konieczne jest stworzenie ciągu odcinków, gdzie koniec poprzedniego jest jednocześnie początkiem następnego. Można w tym celu użyć oczywiście pojedyncze elementy dublując wystąpienie każdego punktu wspólnego, ale dużo lepszym rozwiązaniem wydaje się skorzystanie z `GL_LINE_STRIP`. Ten prymityw, tak jak i następny omawiany, może być podlegać modyfikacji w taki sam sposób jak ma to miejsce w przypadku `GL_LINES`.

Listing 4.3. Ciąg odcinków

```
  glBegin(GL_LINE_STRIP);
2     glVertex3f(-1.0,- 1.0, 0.0);
      glVertex3f(1.0, 1.0, 0.0);
4     glVertex3f(1.0, -1.0, 0.0);
  glEnd();
```

4.1.4. GL_LINE_LOOP

Trzeci typ rysowania odcinków dostępny w OpenGL przypomina mocno ciąg odcinków z wyjątkiem tego, że dokonuje domknięcia, co w praktyce oznacza, że zostaje wygenerowana dodatkowa linia łącząca punkt ostatni

z listy z pierwszym. Mimo że wynik przypomina wielokąt, to nie należy dać się temu wrażeniu zwieść, ponieważ tak powstały obiekt nie ma wypełnienia.

Listing 4.4. Pętla z odcinków

```
1 glBegin(GL_LINE_LOOP);
      glVertex3f(-1.0, -1.0, 0.0);
3     glVertex3f(1.0, -1.0, 0.0);
      glVertex3f(1.0, 1.0, 0.0);
5 glEnd();
```

4.1.5. GL_TRIANGLES

Przypuszczalnie najczęściej wykorzystywanym prymitywem jest trójkąt. Wynika to z wielu różnych przesłanek - od tego, że każdy wielokąt można zdekomponować na trójkąty składowe, po fakt, że zawsze mamy gwarancję płaskości jego powierzchni.

Listing 4.5. Generacja trójkąta

```
1 glBegin(GL_TRIANGLES);
      glVertex3f(-1.0, -1.0, 0.0);
3     glVertex3f(1.0, -1.0, 0.0);
      glVertex3f(1.0, 1.0, 0.0);
5 glEnd();
```

4.1.6. GL_TRIANGLE_STRIP

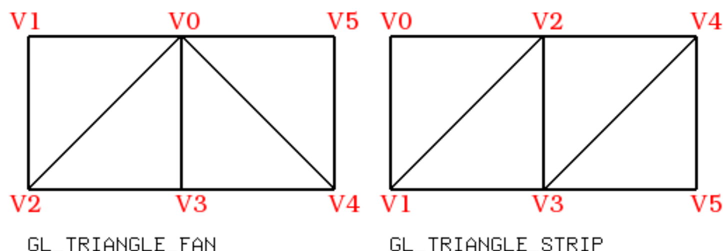
Tak pasek trójkątów, jak i wachlarz jest o tyle ważny, że nie tylko umożliwia zapisanie nam całej grupy z użyciem mniejszej liczby podawanych pozycji wierzchołków niż ma to miejsce w przypadku traktowania każdego z nich osobno, ale również przyczynia się do wzrostu wydajności, ponieważ karty graficzne mają mechanizmy optymalizujące taką organizację danych. Wynika to w znacznym stopniu z faktu, że kolejne trójkąty posiadają wspólne wierzchołki i krawędzie co znacznie redukuje liczbę koniecznych operacji przy przekształceniach geometrycznych.

Oczywiście podział złożonych obiektów na odpowiednie paski trójkątów nie jest banalny. Ważne przy tym jest odpowiednia kolejność wierzchołków w liście (patrz rysunek 4.4).

Listing 4.6. Generacja paska czterech trójkątów

```
1 glBegin(GL_TRIANGLE_STRIP);
      glVertex3f(-1.0, 1.0, 0.0);
3     glVertex3f(-1.0, -1.0, 0.0);
```

```
    glVertex3f(1.0, 1.0, 0.0);
5   glVertex3f(1.0, -1.0, 0.0);
    glVertex3f(3.0, 1.0, 0.0);
7   glVertex3f(3.0, -1.0, 0.0);
glEnd();
```



Rysunek 4.4. Kolejność wierzchołków dla prymitywów GL_TRIANGLE_FAN i GL_TRIANGLE_STRIP

4.1.7. Biblioteki Tri Stripper i NvTriStrip

Jeżeli operujecie na bardzo złożonych obiektach to ręczna organizacja trójkątów w paski może okazać się niewykonalna. Rozwiązaniem w tym wypadku jest skorzystanie z którejś z bibliotek oferujących optymalizację i przygotowujących cachea wierzchołków. Jedną z pośród kilku istniejących jest Tri Stripper, o którym możecie więcej poczytać na stronie `http://users.telenet.be/tfautre/softdev/tristripper/`. Warto też zapoznać się z dostępnym tam porównaniem wydajności z oferującą analogiczną funkcjonalność biblioteką NvTriStrip `http://developer.nvidia.com/object/nvtristrip_library.html`.

4.1.8. GL_TRIANGLE_FAN

Drugim typem organizacji grupy trójkątów jest wachlarz, w którym wszystkie elementy mają wspólny jeden z wierzchołków. Sytuacja taka jest znacznie rzadziej spotykana niż pasek i obejmuje w większości sytuacji mniejszą liczbę trójkątów stąd też jej mniejsza popularność. Kolejność wierzchołków na liście jest zaprezentowana na rysunku 4.4.

Listing 4.7. Generacja wachlarza czterech trójkątów

```
glBegin(GL_TRIANGLE_FAN);
2   glVertex3f(1.0, 1.0, 0.0);
    glVertex3f(-1.0, 1.0, 0.0);
```

```
4       glVertex3f(-1.0, -1.0, 0.0);
        glVertex3f(1.0, -1.0, 0.0);
6       glVertex3f(3.0, -1.0, 0.0);
        glVertex3f(3.0, 1.0, 0.0);
8   glEnd();
```

4.1.9. GL_QUADS

Kolejnymi prymitywami dostępnymi w ramach OpenGL są czworokąty. Mimo że wydają się atrakcyjne to bywa, że generują więcej problemów niż dają korzyści. Podstawową przyczyną takiego stanu rzeczy jest fakt, że wierzchołki w modelu trójwymiarowym mogą nie leżeć w jednej płaszczyźnie, co powoduje problemy nie tylko z wyliczaniem wartości normalnej do powierzchni figury, ale również nie ma gwarancji, że taki obiekt zostanie wyrenderowany poprawnie. o ile obejście pierwszego problemu jest możliwe, jeżeli mamy znajomości geometrii całego obiektu, to drugi jest beznadziejny.

Listing 4.8. Narysowanie czworokąta

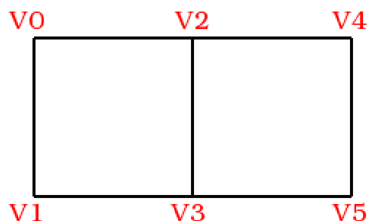
```
    glBegin(GL_QUADS);
2       glVertex3f(-1.0, -1.0, 0.0);
        glVertex3f(1.0, -1.0, 0.0);
4       glVertex3f(1.0, 1.0, 0.0);
        glVertex3f(-1.0, 1.0, 0.0);
6   glEnd();
```

4.1.10. GL_QUAD_STRIP

Podobnie jak trójkąty, czworokąty też mogą być układane w paski. Rysunek 4.5 prezentuje kolejność zapisu pozycji kolejnych wierzchołków w ramach listy. Nie jestem w stanie wskazać żadnej biblioteki, która dokonywałaby optymalizacji obiektów z nich złożonych pod kątem generacji pasków czworokątów. Być może przyczyna leży w fakcie, że czworokąt, zresztą tak jak i wielokąt, został wprowadzony do OpenGL trochę na siłę i z założeniem używania do generacji grafiki dwuwymiarowej, wspomniane już programy inżynierskie, niż modelowania trójwymiarowego.

Listing 4.9. Generacja paska złożonego z dwóch czworokątów

```
    glBegin(GL_QUAD_STRIP);
2       glVertex3f(-1.0, 1.0);
        glVertex3f(-1.0, -1.0);
4       glVertex3f(1.0, 1.0);
        glVertex3f(1.0, -1.0);
6       glVertex3f(3.0, 1.0);
```

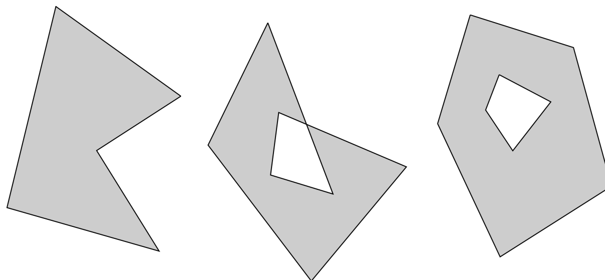


Rysunek 4.5. Kolejność wierzchołków dla prymitywu GL_QUAD_STRIP

```
      glVertex3f(3.0, -1.0);
8  glEnd();
```

4.1.11. GL_POLYGON

Listę prymitywów zamyka wielokąt. Ponieważ OpenGL jest zaprojektowany i implementowany pod kątem maksymalnej wydajności, konieczne stało się nałożenie pewnych ograniczeń na tę grupę elementów. Wymaga się, aby wielokąty, tyczy się to też czworokątów, były figurami wypukłymi i prostymi. Oznacza to, że nie mogą posiadać ani jednego kąta większego niż półpełny, ich boki nie mogą się przecinać, a powierzchnia zawierać dziur (patrz rysunek 4.6). Jednakże, ponieważ OpenGL nie sprawdza poprawności przekazywanych mu opisów obiektów, więc jeżeli jednak zdarzy się tak, że opis waszego wielokąta nie będzie spełniał tych warunków, to nie spowoduje to żadnego komunikatu o błędzie. Należy się tylko liczyć z tym, że sama scena nie zostanie wygenerowana w sposób poprawny.



Rysunek 4.6. Niedopuszczalne postaci wielokątów w OpenGL

Co jednak począć, jeżeli nasze obiekty składają się właśnie z niepoprawnych. według kryteriów nałożonych przez OpenGL, wielokątów? Przy prostych modelach można samemu dokonać zamiany na trójkąty, w przypadku

bardziej złożonych można skorzystać z mechanizmów tesselacji zaimplementowanych w postaci funkcji, które dostępne są w ramach biblioteki GLU. Ich użycie wykracza jednak znacznie poza zakres tej książki.

Listing 4.10. Lista wierzchołków dla wielokąta

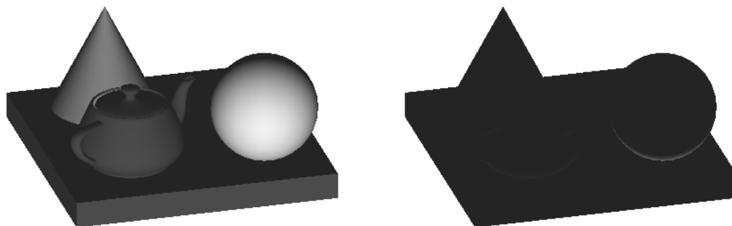
```
glBegin(GL_POLYGON);
    glVertex3f(-1.0, 1.0, 0.0);
    glVertex3f(1.0, 1.0, 0.0);
    glVertex3f(3.0, 0.0, 0.0);
    glVertex3f(1.0, -1.0, 0.0);
    glVertex3f(-1.0, -1.0, 0.0);
glEnd();
```

Ważnym aspektem przy OpenGL jest fakt, że wszystkie wielokąty są traktowane jako wycinki powierzchni zorientowanej i przekazywanie w liście wierzchołków powinny być w określonej kolejności. w praktyce przekłada się to na posiadanie przez nie dwóch stron - przodu i tyłu.

Przyjęto, że przód jest wtedy, gdy idąc po obwodzie figury kolejno według opisu poruszamy się w kierunku przeciwnym do wskazówek zegara. Jeżeli z jakiegoś powodu zależy nam na zmianie orientacji domyślnej to możemy skorzystać w tym celu z funkcji `glFrontFace()`, która jako parametr wywołania przyjmuje dwie wartości zdefiniowane jako stałe `GL_CCW` i `GL_CW`.

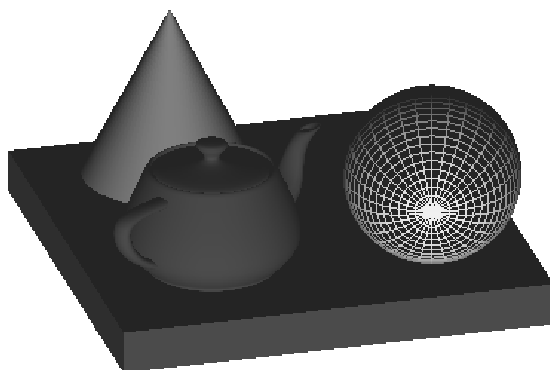
W tym miejscu wspomnimy jeszcze o dwóch zastosowaniach dla stron prymitywów. w przypadku poprawnie zbudowanych modeli brył wszystkie składające się na nie wielokąty powinny być zorientowane w jedną stronę i i ustawione przodem na zewnątrz. Oznacza to, że w możliwe jest zrezygnowanie z wyświetlania tyłu wszystkich tworzących ją prymitywów ponieważ nigdy nie będą one widoczne. Jednak prawdopodobnie ze względu na brak zaufania do użytkowników opcja ta jest domyślnie wyłączona. Zmiana tego stanu rzeczy ma miejsce gdy wywołamy funkcję `glEnable()` wraz z parametrem `GL_CULL_FACE`. Teraz z pomocą `glCullFace()` możemy ustawić, która strona wielokąta ma być ukrywana gdy nie jest zwrócona swoją powierzchnią do obserwatora. Dopuszczalne opcje to, domyślnie wybrane, `GL_BACK` dla tyłu, `GL_FRONT` dla przodu i `GL_FRONT_AND_BACK`. Ta ostatnia powoduje, jak można się domyślić, całkowite ukrycie wielokąta. Wyłączenie obsługi ukrywania odbywa się standardowo z użyciem `glDisable()`. Przykład działania widoczny jest na przykładowej scenie na rysunku 4.7.

W przypadku wielokątów OpenGL mamy możliwość zdecydowania czy mają być wyświetlane w postaci wypełnionej, samego obrysu, czy punktów na pozycji wierzchołków. Możemy osobno ustalić ten parametr w zależności od tego czy prymityw jest skierowany do nas przodem czy tyłem (patrz



Rysunek 4.7. Użycie `glCullFace()` z opcjami GL_CULL_BACK i GL_CULL_FRONT

rysunek 4.8). Używamy w tym celu funkcji `glPolygonMode()` z dwoma parametrami. Pierwszy mówi o stronie, której tyczy się ustawiany tryb przyjmuje takie wartości, jak miało to miejsce w przypadku `glCullFace()`. Drugi określa sam tryb i przyjmuje wartości `GL_FILL` dla wypełnienia, `GL_LINE` w przypadku linii i `GL_POINT` punktu.

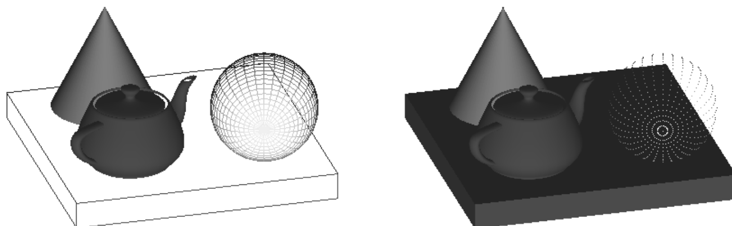


Rysunek 4.8. Użycie `glPolygonMode()` z różnymi opcjami dla wielokątów kuli zwróconych przodem (GL_LINE) i tyłem (GL_FILL)

Na rysunku 4.9 mamy wynik użycia różnych trybów wyświetlania wielokątów. Ponieważ włączone w tym wypadku było również ukrywanie powierzchni skierowanych tyłem do obserwatora, to mimo wyrysowania obiektów z użyciem linii, czy też punktów nie dostrzeżemy przez nie ich wnętrza, podczas gdy obiekty znajdujące się za nimi są doskonale widoczne.

Tak jak miało to miejsce w przypadku linii, tak i dla wielokąta możemy zdefiniować własny wzór wypełnienia. w tym wypadku schemat jest bitmapą i ma wymiar 32x32 piksele. Reprezentowany jest przez tablice zawierającą 128 wartości typu `GLubyte`. Listing 4.11) przedstawia definicję wzoru

szachownicy o jednostkowym wymiarze pojedynczego kafelka 8x8 pikseli. w celu użycia wypełnienia w ramach sceny konieczne jest włączenie jego obsługi z użyciem `glEnable()` wywołanej z parametrem `GL_POLYGON_STIPPLE`. Krokiem następnym jest wskazanie na bieżący wzór za pośrednictwem `glPolygonStipple()`.



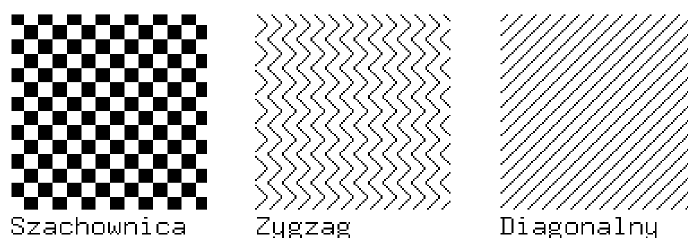
Rysunek 4.9. Użycie `glPolygonMode()` z opcjami GL_LINE dla prostopadłościanu i kuli oraz GL_POINT dla samej kuli

Tu dwie uwagi. Wzory wypełnienia nie powinny być mylone z teksturami mimo, że w obu wypadkach mamy mapę pikseli nanoszone na obiekt, to odbywa się to w całkowicie odmienny sposób i zasadniczo innym celu. Te pierwsze nie podlegają skalowaniu w zależności od odległości od obserwatora i mogą być, tak jak bliźniacze rozwiązanie w przypadku linii, użyte przy tworzeniu dwuwymiarowych systemów inżynierskich.

Listing 4.11. Definicja wzoru wypełnienia użytkownika

```
1 GLubyte check [] = {
  0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00,
3 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00,
  0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00,
5 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00,
  0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff,
7 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff,
  0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff,
9 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff,
  0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00,
11 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00,
  0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00,
13 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00,
  0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff,
15 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff,
  0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff,
17 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff
  };
```

Tekstury zaś są nierozerwalnie związane z geometrią obiektu i podlegają tym samym zasadom jak on jeżeli chodzi przekształcenia czy oświetlenie. o tym jednak szerzej będzie w rozdziale 9.



Rysunek 4.10. Wypełnienie czworokąta z użyciem wzoru zdefiniowanego przez użytkownika

Zarówno biblioteka GLU jak i GLUT zawierają szereg funkcji, które umożliwiają nam wygenerowanie złożonych z prymitywów modeli predefiniowanych. Składają się na nie takie obiekty, jak sześciany, stożki, bryły platońskie, kule, a nawet jeden czajniczek. Dodatkowo, wygląd części z nich może być kontrolowany w oparciu o wartości parametrów podanych przez użytkownika.

4.2. Predefiniowane modele GLU

GLU jest biblioteką, która jest dostępna wraz z OpenGL i zawiera szereg mechanizmów go wspierających, które z tych czy innych względu nie pasowały do przyjętego dla niego profilu. Wśród nich znalazła się obsługa NURBS, kwadryk, mipmappingu, operacji na macierzach widoku, czy wspomnianej już, tesselacji złożonych wielokątów[3]. Jej uniwersalność sprawia, że podobnie jak GLUT, będzie się pojawiać również w kolejnych rozdziałach.

Wśród szerokiej gamy różnych funkcji zawartych w GLU naszą uwagę w tym momencie przykują te, których zadaniem jest generacja kwadryk, czyli powierzchni, których opis jest możliwy przy użyciu równania drugiego stopnia. Oczywiście, istniejących jest dużo więcej niż oferuje nam biblioteka, która, de facto, ogranicza się tylko co cylindra, stożka, dysku i kuli.

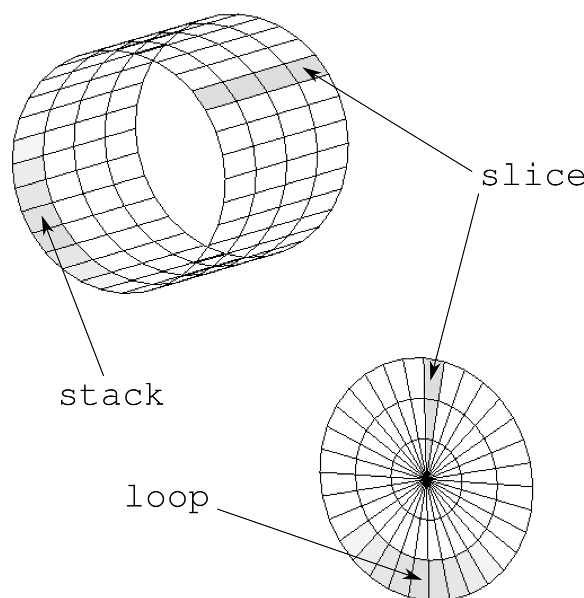
Wygenerowanie wspomnianych obiektów jest wykonywane w kilku kolejnych krokach. Pierwszym jest stworzenie wskaźnika typu GLUquadricObj, który pełni rolę uchwytu. Następnie zostaje wywołana funkcja `gluNewQuadric()`. Teraz możemy przystępować do wywołania funkcji generujących kwadrykę. Do każdej z nich przekazywany jest szereg parametrów,

które wpływają rozmiar, proporcje i precyzję odwzorowania dla gotowego modelu.

Tabela 4.1. Funkcje generujące kwadryki zawarte w GLU

Funkcja	Parametry
`gluCylinder()`	`quadobj` - uchwyt kwadryki `baseRadius` - promień podstawy `topRadius` - promień szczytu `height` - wysokość `slices` - liczba podziałów promieniowych `stacks` - liczba podziałów poziomych
`gluDisk()`	`quadobj` - uchwyt kwadryki `innerRadius` - promień otworu `outerRadius` - zewnętrzny promień dysku `slices` - liczba podziałów promieniowych `loops` - liczba podziałów koncentrycznych
`gluPartialDisk`	`quadobj` - uchwyt kwadryki `innerRadius` - promień otworu `outerRadius` - zewnętrzny promień dysku `slices` - liczba podziałów promieniowych `loops` - liczba podziałów koncentrycznych `startAngle` - kąt początkowy `sweepAngle` - rozmiar kątowy dysku
`gluSphere()`	`quadobj` - uchwyt kwadryki `radius` - promień `slices` - liczba podziałów promieniowych `stacks` - liczba podziałów poziomych

Do tabeli 4.1 zawierające zebrane funkcje generacji obiektów GLU konieczne dodanie jest kliku słów komentarza. Trzeba pamiętać o fakcie, że w przypadku `gluCylinder()` otrzymujemy modele, które posiadają tylko boczną powierzchnię. w sytuacji gdy wartość `baseRadius` jest równa `topRadius` otrzymamy walec, przy różnych zaś obcięty stożek. Jeżeli jeden z tych parametrów jest wyzerowany, wygenerowany zostanie stożek. Do generacji dysku służą aż dwie funkcje `gluDisk()` i `gluPartialDisk()`. Ta druga tworzy wycinek dysku w formie znanej z wykresów tortowych. Dla obu, jeżeli parametr `innerRadius` będzie większy od zera, otrzymamy w wygenerowanym obiekcie koncentrycznie umieszczony okrągły otwór (patrz rysunek 4.12).



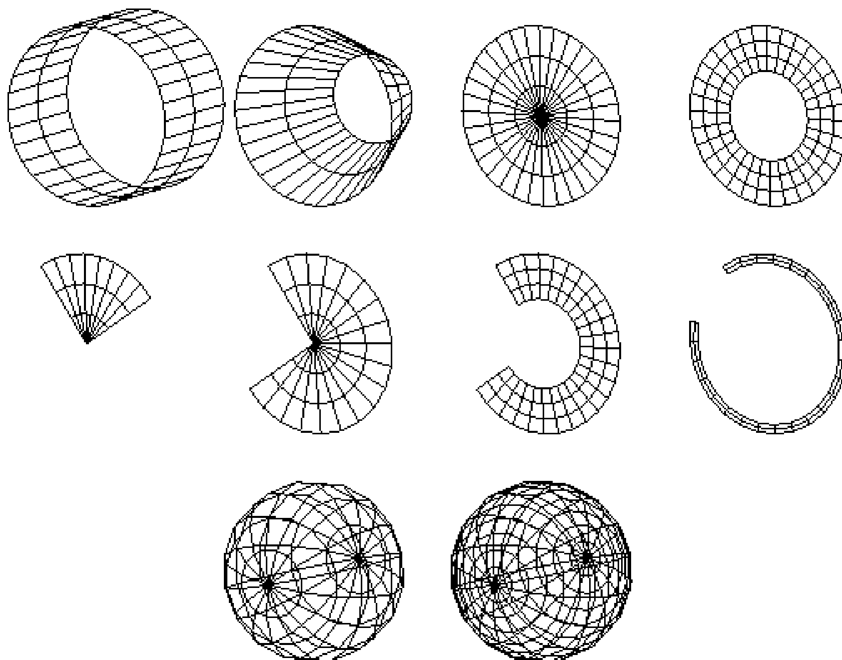
Rysunek 4.11. Nazwy używane przy opisie podziałów dla kwadryk w GLU

Dobór wartości dla gęstości podziałów przy generacji modeli ma wpływ na ostateczny ich wygląd. Szczególnie ma to duże znaczenie przy powierzchniach obłych, gdzie szczególnie mocno widać poczynione w tym względzie oszczędności. Oczywiście, każdy przypadek powinien być rozpatrywany osobno, można jednak przyjąć, że w cylindrach, kulach i dyskach by podziałów promieniowych (slice) było minimalnie dziesięć. Dodatkowo dla kuli liczba podziałów poziomych nie powinna być mniejsza niż osiem. Jeżeli obiekt obserwujemy na naszej scenie z bliska, wartości te należy podnieść.

Pozostałe parametry podziału mogą mieć wpływ na jakość generacji oświetlenia dla obiektu, ale ich wartości nie są krytyczne i mogą być niskie, czy nawet przyjmować jeden czyli dopuszczalne minimum.

Usunięcie kwadryki i zwolnienie zaalokowanej dla niej pamięci odbywa się z użyciem funkcji `gluDeleteQuadric()` podając jako jedyny parametr wywołania uchwyt.

Na sposób wyświetlania modeli predefiniowanych w ramach biblioteki GLU mamy bezpośredni wpływ między innymi za pośrednictwem funkcji `gluQuadricDrawStyle()`. Umożliwia nam ona na wybranie, czy będzie obiekt narysowany z użyciem wypełnionych wielokątów, linii czy punktów. Po kolei odpowiadają za to stałe `GLU_FILL`, `GLU_LINE`/`GLU_SILHOUETTE`, `GLU_POINT`, przekazane jako drugi parametr wywołania funkcji. Pierwszym parametrem jest, jak łatwo się domyślić, uchwyt do kwadryki.



Rysunek 4.12. Predefiniowane modele w GLU wygenerowane z różnymi wartościami parametrów

Zmiana orientacji wielokątów tworzących wygenerowane obiekty odbywa się przy użyciu `gluQuadricOrientation()` i skojarzone z nią dwie stałe: `GLU_OUTSIDE` i `GLU_INSIDE`. Pierwsza z nich jest wartością domyślną i oznacza wektory normalne skierowane na zewnątrz, co automatycznie wiąże się z orientacją przeciwną do ruchu wskazówek zegara.

Zagadnienia związane z nakładaniem tekstur na modele predefiniowane w GLU i wyborem sposobu wyliczania normalnej pojawią się w późniejszych rozdziałach.

Listing 4.12. Stworzenie kwadryki w postaci cylindra z użyciem GLU

```
    GLUquadricObj *kwadryka;
2   kwadryka = gluNewQuadric();
    gluCylinder(kwadryka, 1, 1, 2, 32, 4);
4   gluDeleteQuadric(kwadryka);
```

4.3. Predefiniowane modele GLUT

Również GLUT zawiera szereg możliwości jeżeli chodzi o generowanie predefiniowanych modeli. w tym wypadku procedura jest znacznie uproszczona względem kwadryk, które do zaoferowania ma biblioteka GLU i ogranicza się do wywołania w bloku opisu sceny pojedynczej funkcji. Oczywiście wiąże się to z mniejszymi możliwościami wpływania na właściwości tworzonego modelu.

Pierwszą grupą dostępnych metod są generatory brył platońskich. Jest ich, a jakże, pięć. Choć, będąc bardziej precyzyjnym, to w zasadzie dziesięć, ponieważ każda występuje w dwóch wariantach. Jednym tworzącym modele ze ścianami wypełnionymi i drugim zawierającym tylko krawędzie.

Małe przypomnienie dla tych, co przysnęli w szkole. Bryły platońskie to te, których wszystkie ściany to wielokąty foremne a do każdego wierzchołka schodzi się ta sama liczba krawędzi. Okazuje się, że można je budować tylko z trójkątów, kwadratów i pięciokątów i w sumie istnieje tylko pięć możliwych kombinacji w tym aż trzy dla pierwszej z wymienionych figur. Są nimi - *tetraedr*, czyli czworościan foremny,*oktaedr* zwany po polsku ośmiościanem i *ikosaedr* alias dwudziestościan. z kwadratów złożyć można *heksaedr*, czyli sześcian, a z pięciokąta *dodekaedr* - dwunastościan.

Funkcje, które w GLUT są odpowiedzialne za stworzenie brył platońskich, zawiera tabela 4.2 i wszystkie za wyjątkiem sześcianu są bezparametrowe. z nieznanych przyczyn do `glutSolidCube()` i `glutWireCube()` przekazywana jest wartość współczynnika skalowania.

Mimo że według Platona te pięć brył reprezentowało żywioły, to dla grafika komputerowego tylko reprezentant Ziemi, sześcian, jest przydatny. Reszta stanowi ciekawostkę, a w GLUT znalazła się jako ukłon w kierunku klasycznej filozofii i geometrii.

Tabela 4.2. Funkcje generujące bryły platońskie w GLUT

Funkcja	Generowana bryła
`glutSolidTetrahedron()` `glutWireTetrahedron()`	czworościan foremny
`glutSolidOctahedron()` `glutWireOctahedron()`	ośmiościan foremny
`glutSolidIcosahedron()` `glutWireIcosahedron()`	dwudziestościan foremny
`glutSolidCube()` `glutWireCube()`	sześcian
`glutSolidDodecahedron()` `glutWireDodecahedron()`	dwunastościan foremny

Do kolejnej grupy funkcji zaliczamy trzy możliwe do otrzymania modele. Jest to w identyczny sposób jak w bibliotece GLU budowana kula, stożek, tym razem z podstawą i pierścień. w pierwszym przypadku niewątpliwą przyczyną zdublowania bryły występującej również w bibliotece standardowej OpenGL jest chęć uproszczenia życia programistom.

Tabela 4.3. Funkcje generujące kulę, stożek i pierścień zawarte w GLU

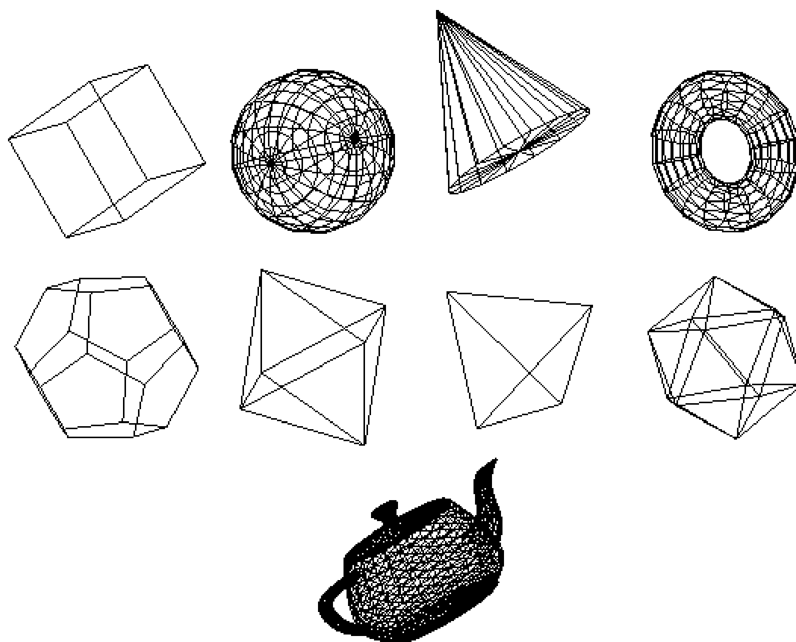
Funkcja	Parametry
`glutSolidSphere()` `glutWireSphere()`	`radius` - promień `slices` - liczna podziałów promieniowych `stacks` - liczba podziałów poziomych
`glutSolidCone()` `glutWireCone()`	`base` - szerokość podstawy `height` - wysokość `slices` - liczba podziałów promnienowych `stacks` - liczba podziałół poziomych
glutSolidTorus() glutWireTorus()	`innerRadius` - promień wewnętrzny `outerRadius` - promień zewnętrzny `sides` - liczba ścian w segmencie `rings` -liczba segmentów

Ostatnim dostępnym modelem, który można wygenerować za pomocą funkcji zawartej w bibliotece GLUT, jest czajniczek. Tak, dobrze czytacie, czajniczek, choć będąc bardziej precyzyjnym, jest to imbryk. Dla trójwymiarowej grafiki komputerowej ma on podobne znaczenie, jak bryły platońskie miały dla szkół filozofii w starożytności.

Teapotahaedr, jak niektórzy go zwą, to matematyczny opis imbryka firmy Melitta. Powstał w roku 1975 w celu testowania burzliwie rozwijających się w tamtym czasie metod renderingu modeli trójwymiarowych na czymś innym niż kule, sześciany i stożki. Okazał się strzałem w dziesiątkę i odgrywa podobną, referencyjną, rolę jak Lena w przetwarzaniu obrazu. Pierwowzór znajduje się obecnie w Computer History Museum w Mountain View w Kaliforni. Martin Newell tworząc go, popełnił mały błąd i nie skorygował nietypowych proporcji wyświetlacza, z którym pracował, przez co czajniczek z Utah jest bardziej pękaty niż jego materialny pierwowzór.

Do wygenerowania obiektu reprezentującego testowy imbryk w GLUT należy wywołać z przekazaniem jako parametru wartości współczynnika skalowania funkcję `glutSolidTeapot()` dla wersji z wypełnionymi wielokątami i `glutWireTeapot()` przy złożonej z samych krawędzi. Warto tu wspomnieć, że wersja znajdująca się w bibliotece ma wielokąty ją tworzące zorientowane odwrotnie niż w przypadku pozostałych modeli. w tej sytuacji, jeżeli

planujecie z niego korzystać, to warto przed jego generacją wywołać `glFrontFace(GL_CCW)`.



Rysunek 4.13. Dostępne w GLUT modele

Rozdział 5

Macierze i przekształcenia geometryczne

5.1. Współrzędne jednorodne **58**
5.2. Macierz modelu-widoku **59**
5.3. Macierze przekształceń **59**
5.4. Złożenie przekształceń **60**
5.5. Przesunięcie **61**
5.6. Obrót **61**
5.7. Skalowanie **63**
5.8. Stos macierzy modelu-widoku **64**
5.9. Odwzorowanie hierarchii **66**
5.10. Normalne **71**

Rysunek 5.1. Vernon Evans i rodzina Lemonów, Południowa Dakota, niedaleko Missouli, Montana, Droga numer 10. [–] Pokonują około 200 mill dziennie w Fordzie model T. *Arthur Rothstein, Library of Congress, Prints & Photographs Division, FSA-OWI Collection, 1936*

.

Ten rozdział w całości poświęcimy niezwykle ważkiej tematyce związanej z programowaniem z użyciem OpenGL. Pojawi się też trochę matematyki, dokładniej zaś rachunek macierzowy. Jeżeli nie czujecie się w nim zbyt mocno, to nie przejmujcie się tym bardzo, nawet bez jego znajomości będziecie w stanie wynieść z wywodu całkiem sporo. Możemy się umówić, że będzie to trochę jak w przypadku kierowcy, który potrafi prowadzić samochód, ale nie zna cyklu Otta. Zresztą, rozdział w pewnym sensie stał będzie pod znakiem motoryzacji, bo w jego drugiej części zmierzymy się tak ze zdobytą do tej pory wiedzą, jak i legendą motoryzacji.

5.1. Współrzędne jednorodne

Okazuje się, że znane Wam ze szkoły współrzędne kartezjańskie nie są optymalne w przypadku ich użycia do przekształceń obiektów w przestrzeni trójwymiarowej. Szczególnie problematyczne jest to, gdy mamy do czynienia

z operacjami rzutowania, co jest jednym ze standardowych etapów przetwarzania również w OpenGL. Problem leży również w fakcie, że w geomtrii euklidesowej dwie proste równoległe nigdy się nie przetną podczas, gdy intuicyjnie wiemy, że w przypadku rzutu tych prostych taka sytuacja może mieć miejsce w nieskończoności. Przykładem z życia wziętym są tory kolejowe, na które patrząc, widzimy, że schodzą się do jednego punktu. Tutaj z pomocą przychodzi nam geometria rzutowa i zaproponowane przez Augusta Ferdinanda Möbiusa w 1827 współrzędne jednorodne. Dodają one do zestawu składowych współrzędnych kartezjańskich x, y, z dodatkową składową w. Wartość w w sytuacji gdy mamy do czynienia z operacjami w przestrzeni euklidesowej jest równa *1*.

5.2. Macierz modelu-widoku

W przypadku pracy z programami do modelowania trójwymiarowego zarówno obiekty, jak i kamera mają skojarzone lokalne układy współrzędnych, a precyzyjniej macierze opisujące ich położenie i orientację względem układu globalnego sceny. Pozwala to na łatwą modyfikację każdego z elementów osobno.

Ze względu na specyficzną konstrukcję OpenGL charakterystyczny dla niego jest fakt, że nie ma osobnych macierzy służących do opisu położenia i ruchu wirtualnej kamery i modelu. Zamiast tego wprowadzono jedną wspólną macierz `GL_MODELVIEW`. Co więcej, jeżeli chcemy dla sceny składającej się z wielu modeli na każdym z nich dokonać innych przekształceń konieczne będzie użycie swego rodzaju technik specjalnych. Takie rozwiązanie ma tak swoje dobre jak i złe strony, ale jeżeli zależy Wam na opanowaniu umiejętności programowania w OpenGL, to jedynie pozostaje poznać je, zrozumieć i polubić.

$$M_{mv} = \begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}$$

5.3. Macierze przekształceń

Okazuje się, że korzystanie ze współrzędnych jednorodnych jest korzystne jeszcze z jednego powodu. Umożliwia nam opis przekształceń geometrycznych obiektów z wykorzystaniem macierzy, a złożenie wielu operacji jako ich mnożenie. Rachunek macierzowy miedzy innymi dzięki swojej prostocie zapisu nawet skomplikowanych operacji bardzo dobrze poddaje się

optymalizacji. z tego powodu współczesne procesory graficzne, pomijając szeroki wachlarz funkcji dodatkowych, to w zasadzie specjalizowane koprocesory macierzowe.

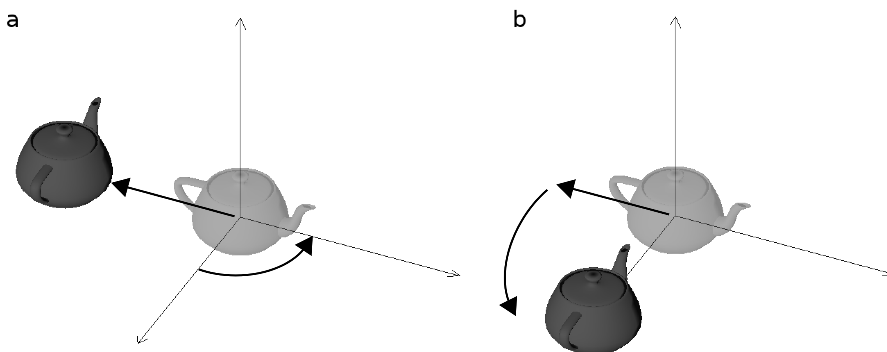
W chwili gdy ustawiamy maszynę stanów tryb pracy z macierzą modelu-widoku z wykorzystaniem funkcji `glMatrixMode()` standardową procedurą jest również zresetowanie aktualnej macierzy modelu, a precyzyjniej sceny, do postaci jednostkowej. Wykonywane jest za pomocą funkcji `glLoadIdentity()`.

OpenGL, w momencie kiedy stosujemy do sceny jedną z trzech dostępnych operacji przekształceń geometrycznych, dodaje ona do mnożenia kolejną macierz. w momencie, kiedy do maszyny stanów zostają przekazane współrzędne wierzchołków obiektu, dokonywane jest ich przemnożenie przez kolejne zgromadzone macierze. Jeżeli założymy, że przekształcenia (T_1 i T_2) zostały zdefiniowane przez kolejne wywołania funkcji OpenGL, a wierzchołek opisuje wektor v, to w wyniku otrzymamy następujące mnożenie:

$$v' = I(T_1(T_2(v)))$$

5.4. Złożenie przekształceń

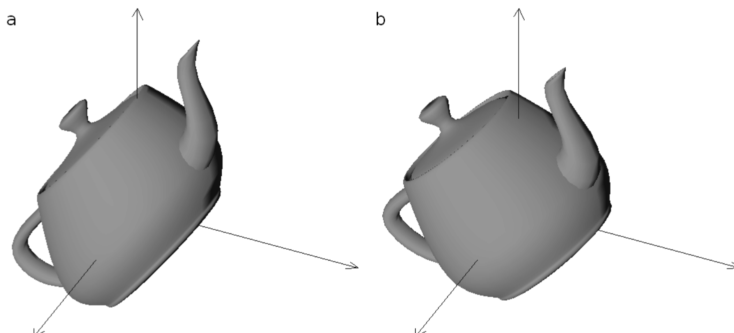
Co dla nas z tych informacji wynika? Rzecz dość zasadnicza dla programisty OpenGL. w przypadku tworzenia aplikacji z wykorzystaniem aktywnej macierzy model-widok przekształcenia modelu mają miejsce w kolejności odwrotnej niż zostały wyspecyfikowane w kodzie, a sama kolejność nie jest przypadkowa i wynika z faktu, że mnożenia macierzy nie są przemienne.



Rysunek 5.2. Wynik operacji

Dlatego bardzo ważne jest dokładne zaplanowanie, jak obiekt ma być przekształcany (patrz rysunek 5.2). Szczególnie istotne jest to w sytuacji,

gdy dokonujemy skalowania o różne współczynniki dla poszczególnych kierunków i obroty może to zaowocować skoszeniem geometrii obiektu. Przykładowa deformacja tego typy zaprezentowana została na rysunku 5.3.



Rysunek 5.3. Deformacja będąca wynik złej kolejności operacji obrotu i skalowania względem jednej z osi (a) oraz rezultat kolejności poprawnej (b).

5.5. Przesunięcie

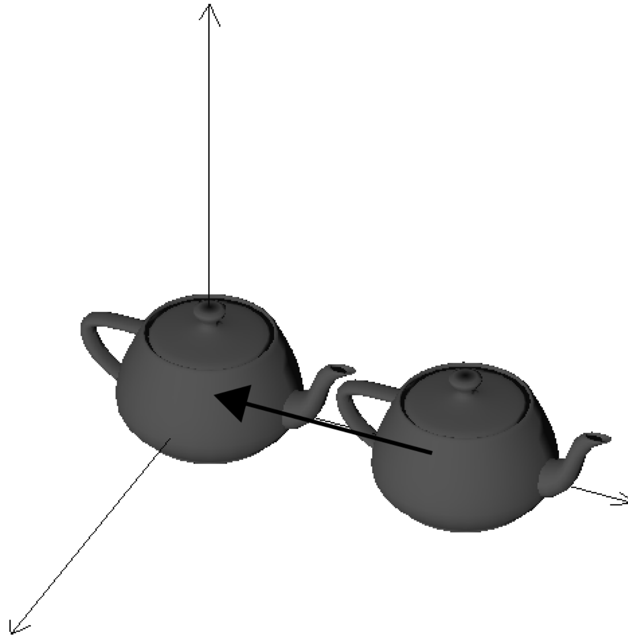
Przesunięcie, które sprowadza się w przypadku współrzędnych euklidesowych do dodania wartości poszczególnych składowych wektora opisującego zmianę położenia do odpowiadających im składowych przemieszczanego punktu, w zapisie macierzowym dla współrzędnych jednorodnych wyraża się w następujący sposób:

$$M_T = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

W OpenGL za przesunięcie odpowiedzialna jest funkcja `glTranslatef()`, która ma w wywołaniu trzy argumenty opisujące wektor zmiany położenia obiektu.

5.6. Obrót

W przypadku gdy obrót o kąt θ ma miejsce względem dowolnego wektora znormalizowanego o współrzędnych x, y i z to macierz obrotu jest opisana w następujący sposób:



Rysunek 5.4. Przesunięcie o wektor

$$M = \begin{bmatrix} r_1 & r_4 & r_7 & 0 \\ r_2 & r_5 & r_8 & 0 \\ r_3 & r_6 & r_9 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

gdzie wartości r_1 do r_9 opisują wyrażenia:

$$r_1 = x^2(1 - cos(\theta)) + cos(\theta)$$

$$r_2 = yx(1 - cos(\theta)) + zsin(\theta)$$

$$r_3 = xz(1 - cos(\theta)) - ysin(\theta)$$

$$r_4 = xy(1 - cos(\theta)) - zsin(\theta)$$

$$r_5 = y^2(1 - cos(\theta)) + cos(\theta)$$

$$r_6 = yz(1 - cos(\theta)) + xsin(\theta)$$

$$r_7 = xz(1 - cos(\theta)) + ysin(\theta)$$

$$r_8 = yz(1 - cos(\theta)) - xsin(\theta)$$

$$r_9 = z^2(1 - cos(\theta)) + cos(\theta)$$

W przypadku wykonywania obrotów względem osi układu współrzędnych, co w większości wypadków jest łatwiejsze i bardziej intuicyjne, macierz upraszcza się do klasycznych, postaci. i tak dla obrotu względem osi *0X* jest to:

$$M_{Rx} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) & 0 \\ 0 & sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

osi *0Y*:

$$M_{Ry} = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

oraz *0Z*:

$$M_{Rz} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

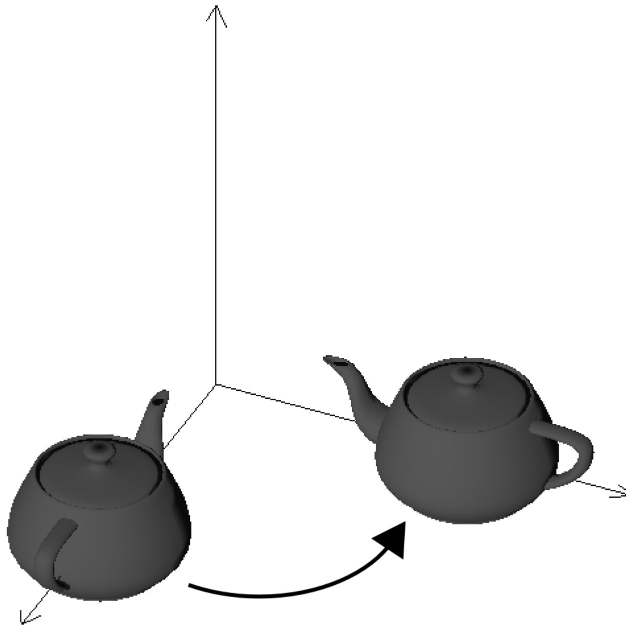
W OpenGL funkcja, która definiuje obrót według zadanego wektora to *glRotatef()* i przyjmuje cztery argumenty. Są to po kolei wartość kąta obrotu wyrażona w stopniach i współrzędne wektora, który, o ile nie będzie znormalizowany, to przed wygenerowaniem macierzy zostanie automatycznie sprowadzony do długości jednostkowej.

5.7. Skalowanie

Skalowanie obiektów w przestrzeni trójwymiarowej odbywa się poprzez podanie współczynników, które następnie zostają przemnożone przez wartości poszczególnych współrzędnych. w reprezentacji macierzowej jest to opisywane za pomocą:

$$M_S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Skalowanie może być niejednorodne, czyli wartości poszczególnych współczynników mogą różne. Użycie współczynników o wartości -1 spowoduje wprowadzenie symetrii względem płaszczyzny. Dla wartości ujemnych, ale różnych od jeden możemy przyjąć, że mamy do czynienia ze złożeniem operacji skalowania i odbicia.

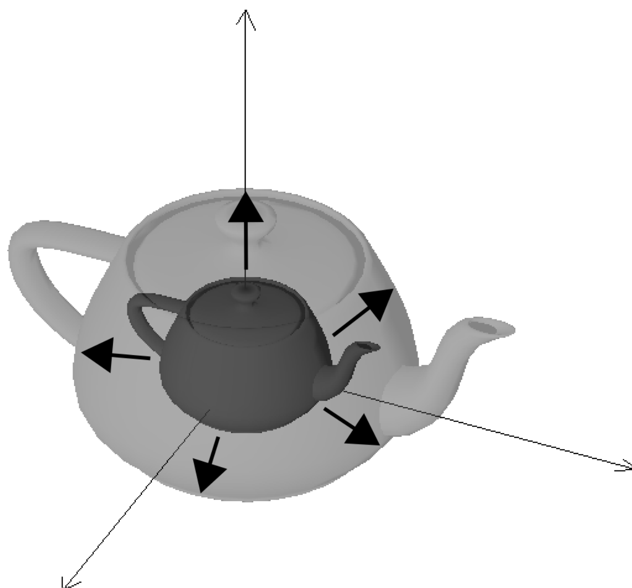


Rysunek 5.5. Obrót względem osi *0X* o zadany kąt.

W przypadku OpenGL za operację skalowania odpowiada `glScalef()`, a jej argumentami są wartości współczynników skalowania kierunków wyznaczanych przez poszczególne osi współrzędnych.

5.8. Stos macierzy modelu-widoku

Za każdym razem, gdy chcemy dodać kolejny obiekt, na którym planujemy dokonać odmiennych przekształceń niż dotychczasowe używamy `glLoadIdentity()`. Takie postępowanie nie jest rozwiązaniem ani optymalnym, ani elastycznym. Co więcej, duży problem pojawia się w sytuacji, gdy mamy do czynienia z modelami złożonymi z wielu obiektów, które są względem siebie powiązane hierarchią. Często pozycja poszczególnych elementów w takim wypadku może zależeć od położenia węzłów wyższego rzędu. Taki właśnie przypadek będziemy za chwilę rozpatrywali. Równie skomplikowane jest nadanie grupie obiektów wspólnego przekształcenia jeżeli w pomiędzy nimi występuje przywrócenie macierzy jednostkowej, ponieważ wymagane jest po tej operacji powtórzenie wszystkich interesujących nas przekształceń.



Rysunek 5.6. Skalowanie o zadany współczynnik

Jedak w OpenGL jest rozwiązanie, które znacznie ułatwia tego typu działania. Jest nim stos macierzy. w przypadku nas w tej chwili interesującym, czyli macierzy modelu-widoku, możliwe jest odłożenie na nim do 32 elementów. Przekłada się to w praktyce na przykład na 32 poziomy zagłębienia w modelu hierarchicznym.

OpenGL w danym momencie pracuje z macierzą znajdującą się na wierzchu stosu. w momencie, gdy dokonywane jest odkładanie na stos macierzy modelu-widoku przy pomocy funkcji `glPushMatrix()`, polega to na wykonaniu kopii bieżącego stanu, czyli wszystkich przekształceń, które zostały zdefiniowane od momentu załadowania postaci jednostkowej, i położeniu go jej na wierzchu stosu.

Zdjęcie ze stosu za pomocą `glPopMatrix()` przywraca zapamiętany stan na warstwie niższej i jednocześnie niszczy zdejmowaną macierz. Zerknijmy na kod z listingu 5.1. Na imbryczek działać będą, w kolejności wykonania, operacje obrotu i przesunięcia, zaś na sześcian tylko przesunięcie.

Listing 5.1. Użycie `glPushMatrix()` i `glPopMatrix()`.

```
  glLoadIdentity();
2 glTranslate(0, 1.0, 0);

4 glPushMatrix();
```

```
    glRotate(45.0, 1, 0, 0);
6   glutSolidTeapot(1.0);
    glPopMatrix();
8
    glutSolidCube();
```

5.9. Odwzorowanie hierarchii

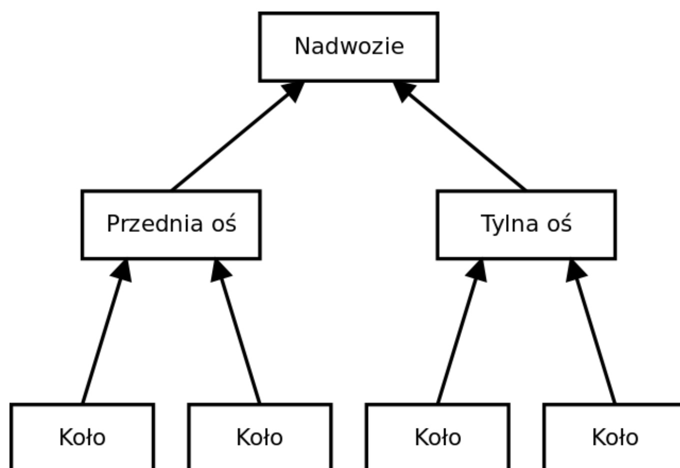
Na koniec wracamy do tematyki motoryzacyjnej. Otóż w roku 1908 z taśm montażowych fabryk Henriego Forda zaczął zjeżdżać pojazd, który miał przyczynić się do sposobu patrzenia i myślenia o samochodzie. Potocznie zwany Blaszaną Elżunią model T był prosty, łatwo naprawialny, radził sobie świetnie na bezdrożach i kosztował grosze. Mimo że był dostępny tylko w jednym, czarnym kolorze, to sprzedawał się świetnie i w ciągu dwudziestu lat wyprodukowano go w niebanalnej liczbie piętnastu milionów egzemplarzy.

Zadaniem, jakie sobie stawiamy, będzie, korzystając z posiadanej już wiedzy, zamodelowanie, rzecz jasna uproszczonego, modelu T. Przy okazji będzie to świetna okazja do przyjrzenia się w praktyce działaniu stosu macierzy modelu-widoku.

Należy tu nadmienić, że tworząc poważniejsze oprogramowanie, nikt przy zdrowych zmysłach nie buduje modeli w przedstawiony sposób, tylko korzysta z przeznaczonego do tego celu oprogramowania. Jednak, obrana tu droga niewątpliwie przyczyni się do głębszego zrozumienia zagadnienia, a przecież o to tu chodzi.

Korzystając z papieru w kratkę, wykonujemy przybliżony projekt modelu. w przypadku, gdy ktoś się brzydzi ołówka, albo uważa to narzędzie za archaiczne, to może skorzystać z jakiegoś programu graficznego. Zasadniczo minimalistyczny samochodzik będzie się składał z kabiny, maski, dwóch świateł, 4 kół i dwóch osi.

Dobrym rozwiązaniem jest rozrysowanie diagramu przedstawiającego hierarchię modelu. Możliwe są tutaj dwa różne podejścia funkcjonalne lub geometryczne. Nas interesuje to pierwsze i jest ono wynikiem analizy składowych obiektów pod kątem tworzenia grup funkcjonalnych i związanych z nimi zależności względem transformacji. Nasz Ford T jadąc porusza się jako całość, ale również w swoim własnym ruchu w tej sytuacji będzie oś, przynajmniej przednia, symulując pokonywanie zakrętów jak i koła, udające toczenie się po drodze. Ruch kół nie wpływa na oś, ale ruch osi na koła tak. Analogicznie ruch osi nie wpływa na nadwozie, ale ruch nadwozia wpływa na cały pojazd. Wyraźnie więc jest widoczna tu hierarchia, którą reprezentuje rynuek 5.7. Takie podejście jest analogiczne z kinematyką prostą.



Rysunek 5.7. Hierarchiczny opis modelu.

Z zaprezentowanej hierarchii wynikać będą też tworzone przez nas funkcje opisujące geometrię poszczególnych elementów modelu, jak i miejsca, w których mieć miejsce będzie korzystanie z funkcji obsługi stosu macierzy stanu model-widok.

Zanim jednak zajmiemy się pracą nad opisem modelu, wprowadzimy dwie funkcje upraszczające nam korzystanie z dwóch kwadryk GLU, które będą nam potrzebne. Ich nazwy `myCylinder()` i `myDisk()` odpowiadają swoim pierwowzorom w podstawowej bibliotece narzędziowej OpenGL, a lista parametrów w ich wywołaniu jest prawie identyczna. Różnica leży w braku wskazania na uchwyt kwadryki i istnieniu dodatkowego parametru określającego orientację generowanych wielokątów (patrz listing 5.2). Jeżeli zaistnieje taka potrzeba nic nie stoi na przeszkodzie, żeby takie funkcje w sposób analogiczny stworzyć dla pozostałych kwadryk.

Listing 5.2. Definicje funkcji `myCylinder()` i `myDisk()`.

```
1 void myCylinder(
          GLdouble base, GLdouble top, GLdouble height,
3         GLint slices, GLint stacks,
          GLenum orientation=GLU_OUTSIDE) {
5
      GLUquadricObj *q;
7     q = gluNewQuadric();
      gluQuadricOrientation(q, orientation);
9     gluCylinder(q, base, top, height, slices, stacks);
      gluDeleteQuadric(q);
11 }

13 void myDisk(
```

```
            GLdouble inner, GLdouble outer,
15          GLint slices, GLint loops,
            GLenum orientation=GLU_OUTSIDE) {
17
        GLUquadricObj *q;
19      q = gluNewQuadric();
        gluQuadricOrientation(q, orientation);
21      gluDisk(q, inner, outer, slices, loops);
        gluDeleteQuadric(q);
23  }
```

W tej chwili możemy przejść do pracy nad naszym modelem. Zaczniemy od przygotowania samego koła, na które składać się będzie osiem szprych, obręcz, piasta oraz opona, co widoczne jest na rysunku 5.8. Całość kodu ubierzemy w funkcję, co pozwoli nam na wykorzystanie gotowego modelu konieczną ilość razy.

Listing 5.3. Opis modelu koła

```
1   void drawWheel(void) {

3       /* Szprychy */
        for(int i=0; i<8; i++) {
5           glPushMatrix();
            glRotatef(45*i,0,1,0);
7           glTranslatef(0,0,0.5);
            myCylinder(0.1, 0.1, 1.5, 8, 4);
9           glPopMatrix();
        }
11
        /* Obrecz */
13      glPushMatrix();
        glRotatef(90, 1, 0, 0);
15      glTranslatef(0,0,0.25);
        myDisk(1.8, 2, 64, 4);
17      glPopMatrix();

19      glPushMatrix();
        glRotatef(90, 1, 0, 0);
21      glTranslatef(0,0,-0.25);
        myCylinder(1.8, 1.8, 0.5, 64, 4, GLU_INSIDE);
23      glPopMatrix();

25      glPushMatrix();
        glRotatef(90, 1, 0, 0);
27      glTranslatef(0,0,-0.25);
        myDisk(1.8, 2, 64, 4, GLU_INSIDE);
29      glPopMatrix();

31      /* Opona */
```

```
    glPushMatrix();
33  glRotatef(90, 1, 0, 0);
    glutSolidTorus(.25, 2.05, 16, 64);
35  glPopMatrix();

37  /* Piasta */
    glPushMatrix();
39  glRotatef(90, 1, 0, 0);
    glTranslatef(0,0,0.5);
41  myDisk(0, 0.5, 16, 4);
    glPopMatrix();
43
    glPushMatrix();
45  glRotatef(90, 1, 0, 0);
    glTranslatef(-0.0,0,-0.5);
47  myCylinder(0.5, 0.5, 1, 64, 4);
    glPopMatrix();
49
    glPushMatrix();
51  glRotatef(90, 1, 0, 0);
    glTranslatef(0,0,-0.5);
53  myDisk(0, 0.5, 16, 4, GLU_INSIDE);
    glPopMatrix();
55 }
```

Przeanalizujmy pokrótce, co dzieje się w kodzie zamieszczonym w listingu 5.8. w linijkach 4-10 mamy pętlę, której zadaniem jest stworzenie w modelu koła szprych. Każda z nich jest pojedynczym walcem, który zostaje obrócony o kąt wyliczony w oparciu o wartość licznika pętli i założoną odległość kątową między kolejnymi szprychami, co w naszym przypadku oznacza 45 stopni.

Wydawać by się mogło, że w tym przypadku można skorzystać z faktu, że w maszynie stanów OpenGL mamy multiplikację kolejnych obrotów z macierzą modelu-widoku i można równie dobrze rozmieścić kolejne szprychy, bez konieczności odkładania dla każdego elementu stanu na stos. w tej sytuacji nasza funkcja mogłaby wyglądać tak jak na listingu 5.4 i o ile jest poprawna z punktu widzenia składni to okazuje się, że wiąże się z nią pewne zagrożenie. Kąt pomiędzy szprychą ostatnią a pierwszą może mieć inną wartość niż 45 stopni, co spowodowane będzie błędami zaookrąglenia. Oczywiście, niewielkie odchyłki w przypadku prostego modelu forda T są akceptowalne, ale jeżeli pisalibyśmy oprogramowanie wyświetlające zegar i ruch wskazówek zaimplementowany byłby w analogiczny do przedstawionego sposób szybko przekonalibyśmy, że wskazuje inną godzinę niż powinien.

Obręcz i piasta składa się z dwóch dysków i cylindra. Dla pierwszego z wymienionych elementów cylinder ma odwróconą orientację, ze względu

na fakt, że w gotowym modelu widoczne będzie jego wnętrze. w ten sam sposób potraktowana została połowa użytych dysków, chociaż w tym wypadku możliwe byłoby ustawienie ich z wielokątami zwróconymi w odpowiednią stronę poprzez wykonanie obrotu o kąt półpełny.

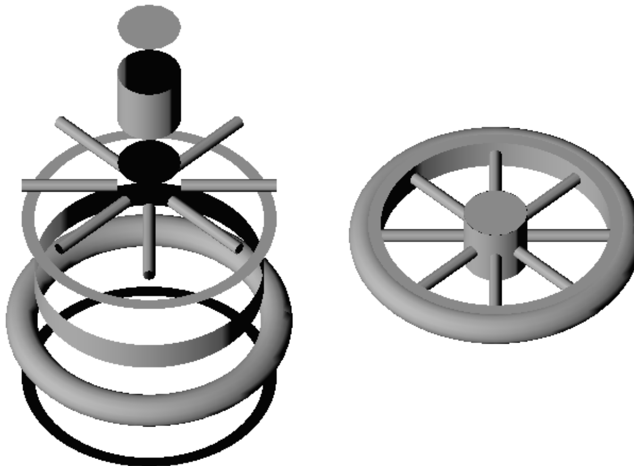
Przy okazji warto zerknąć na kolejność operacji wykonywanych na poszczególnych składowych koła i samemu przekonać się, że przekształcenia wykonywane są w kolejności od ostatniej.

Reszta zaprezentowanego kodu powinna być w pełni czytelna bez dodatkowego komentarza.

Listing 5.4. Błędnie skonstruowana pętla generująca szprychy koła

```
1     glPushMatrix();
      for(int i=0; i<8; i++) {
3         glRotatef(45,0,1,0);
          myCylinder(0.1, 0.1, 2, 8, 4);
5     }
      glPopMatrix();
```

Kolejnym naszym krokiem wynikającym z analizy hierarchii modelu, będzie stworzenie grupy, na którą składać się będą dwa koła i łącząca je oś. Jest stosunkowo krótka (listing 5.5) i powinna być łatwo zrozumiała, ponieważ działania w niej zawarte są analogiczne do tych, które zaprezentowane zostały przy okazji kodu opisującego model koła.

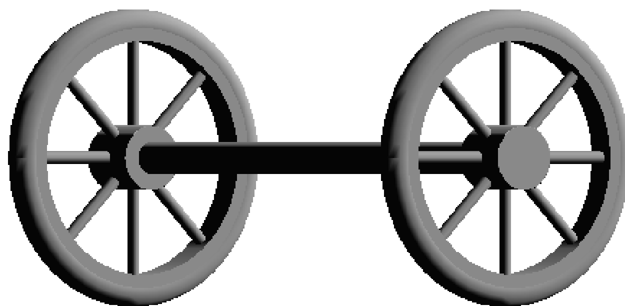


Rysunek 5.8. Elementy koła i złożony model.

Listing 5.5. Opis modelu osi.

```
void drawAxle(void) {
    glPushMatrix();
    glRotatef(90, 1, 0, 0);
    glTranslatef(0.0, 0.0, -5.0);
    myCylinder(0.25, 0.25, 10, 64, 4);
    glPopMatrix();

    glPushMatrix();
    glTranslatef(0.0, -5.0, 0.0);
    drawWheel();
    glPopMatrix();

    glPushMatrix();
    glTranslatef(0.0, 5.0, 0.0);
    drawWheel();
    glPopMatrix();
}
```

Teraz zajmiemy się stworzeniem nadwozia. Będzie ono składało się z kilku elementów zbudowanych z trójkątów. Ponieważ będą to relatywnie proste i płaskie segmenty tesselacja dla nich odbędzie się na papierze.



Rysunek 5.9. Oś wraz z kołami.

5.10. Normalne

Ponieważ chcemy mieć w przyszłości prawidłowo oświetlony nasz model, konieczne jest zadbanie o prawidłowo zdefiniowana normalne dla trójkątów, które składają się na kabinę i maskę pojazdu. Do momentu, kiedy korzystaliśmy z obiektów predefiniowanych w GLU, lub GLUT, nie musieliśmy się tym, zajmować ponieważ odpowiednie obliczenia były wykonywane w tym celu wewnątrz funkcji generujących je.

W przypadku modelowania obiektów trójwymiarowych mamy dwie możliwości przypisywania normalnej. Pierwszy, którym zajmiemy się w tym momencie to wyznaczenie jej względem powierzchni. Drugi, pozwalający na lepsze jakościowo wygenerowanie rozkładu świateł w modelach obiektów o łagodnych, łukowatych kształtach, przypisuje normalną do poszczególnych wierzchołków.

Normalna jest definiowana jako wektor jednostkowy prostopadły do powierzchni. Jego zwrot w przypadku OpenGL skierowany powinien być na zewnątrz modelowanego obiektu i wynika jednoznacznie z orientacji wielokąta, dla którego jest wyliczana.

Dla trójkąta, lub płaskiego wielokąta, dla którego wyznaczymy trzy kolejne wierzchołki ułożone nieliniowo, normalna do powierzchni jest zdefiniowana jako następujący wektor:

$$N_T = (v_1 - v_2) \times (v_3 - v_2)$$

poddany następnie normalizacji:

$$\hat{N}_T = \frac{N_T}{|N_T|}$$

Pamiętacie jeszcze klasy, które zaczęliśmy tworzyć w rozdziale 2.3? Teraz zajmiemy się zadeklarowaną, ale nie zdefiniowaną do tej pory metodą `calculateNormal()`. Widoczna jest ona na listingu 5.6 w formie absolutnie minimalistycznej. w typowej sytuacji dla wyliczania długości i mnożenia wektorów wskazane byłoby stworzenie osobnych metod, jednak niepotrzebnie dokonałoby to zaciemnienia kodu w naszym przypadku.

Listing 5.6. Metoda `calculateNormal()` klasy `MyGLtriangle`.

```
1  void MYGLtriangle::calculateNormal(void) {
       MYGLvertex a;
3      MYGLvertex b;
       GLdouble len;
5
       a.x = v1->x - v2->x;
7      a.y = v1->y - v2->y;
       a.z = v1->z - v2->z;
9
       b.x = v3->x - v2->x;
11     b.y = v3->y - v2->y;
       b.z = v3->z - v2->z;
13
       n.x = a.y * b.z - a.z * b.y;
15     n.y = a.z * b.x - a.x * b.z;
       n.z = a.x * b.y - a.y * b.x;
17
```

```
    len = sqrt((n.x * n.x) + (n.y * n.y) + (n.z * n.z));
19  if(len!=0.0) {
        n.x = n.x / len;
21      n.y = n.y / len;
        n.z = n.z / len;
23  }
}
```

Normalna będąc wektorem skojarzonym z wierzchołkiem modelu podlega wszystkim przekształceniom tyczącym się modelu w skład, którego wchodzi. Przesunięcie i obrót zmieniają wartość współrzędnych opisujących normalną, ale nie mają wpływu na jej długość. Gorzej jest w sytuacji, gdy obiekt zostanie poddany skalowaniu. Wynikowy wektor może mieć długość niejednostkową co wpłynie negatywnie między innymi na wynik modelowania oświetlenia.

Rozwiązaniem jest wymuszenie na OpenGL ponownego znormalizowania wektora normalnego. Jeżeli mamy pewność, że wszystkie skalowania są jednorodne możemy włączyć prosty mechanizm skalowania wektora za pomocą funkcji `glEnable()` z przekazanym do niej parametrem w postaci stałej `GL_RESCALE_NORMAL`. w przeciwnym razie konieczne jest użycie stałej `GL_NORMALIZE`. Należy pamiętać, że obie metody wymagają dodatkowych obliczeń związanych z operacjami przekształceń obiektu i o ile nie jest to konieczne, nie powinny być używane.

Zanim przejdziemy do kolejnego etapu budowy modelu Forda T do klasy `MYGLtriangle` dodamy jeszcze metodę, która będzie generować wywołania funkcji `glNormal()` i `glVertex()` dla wierzchołków wskazywanych w jej intensji (listing 5.7).

Listing 5.7. Metoda `getGLDefinition()` klasy `MyGLtriangle`.

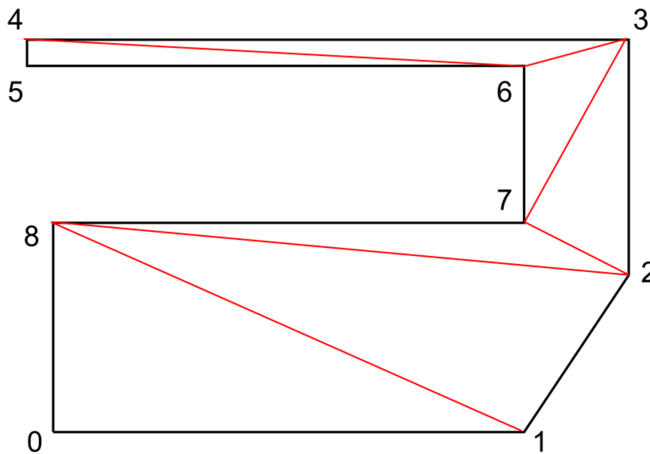
```
  void MYGLtriangle::getGLDefinition(void) {
2     glNormal3f(n.x, n.y, n.z);
      glVertex3f(v1->x, v1->y, v1->z);
4     glVertex3f(v2->x, v2->y, v2->z);
      glVertex3f(v3->x, v3->y, v3->z);
6 }
```

Wracamy do opisu modelu. Zakładamy, że obie boczne ściany będą identyczne, płaskie, stąd wniosek, że trójkąty, z których je zbudujemy różnić tylko się będą orientacją, ponieważ chcemy zachować poprawne oświetlenie na zewnątrz. Połączy jest podłoga, ściana przednia, tylna i dach, które będą wymodelowane jako jeden, ciągły pas.

Planowana tesselacja przedstawiona jest na rysunku 5.10 wraz z numerami poszczególnych wierzchołków. Jeżeli przyjrzymy się teraz kodowi

funkcji z listingu 5.8, to znajdziemy tam tablicę v2d, która zawiera opis współrzędnych kolejnych wierzchołków, które zostaną wykorzystane do rozpięcia trójkątów ścian bocznych. Składowe współrzędnych są dwie, trzecia będzie dodawana w momencie definiowania konkretnych ścian (patrz pętla w liniach od 14 do 17).

Tablica ts2d zawiera indeksy wierzchołków dla kolejnych trójkątów ściany, a tr2d analogicznie dla podłogi i dachu. w oparciu o te dane dokonana zostanie generacja odpowiednich wartości dla zbiorczych tablic wierzchołków i trójkątów. Jest to jedna z możliwości, ponieważ analogiczny efekt można otrzymać bez użycia prezentowanych klas, co oczywiście komplikuje wyliczanie normalnych, jak i opierając na rozbudowanych klasach i listach standardowej biblioteki szablonów C++ (STL).



Rysunek 5.10. Indeksy wierzchołków i tesselacja ściany bocznej nadwozia modelu.

Listing 5.8. Funkcja opisując kabinę Forda T.

```
    void drawBody(void) {
 2      GLdouble v2d[9][2] =
            { {0.0, 0.0}, {9.0, 0.0}, {11.0, 3.0},
 4          {11.0, 7.5}, {-0.5, 7.5}, {-0.5, 7.0},
            {9.0, 7.0}, {9.0, 4.0}, {0.0, 4.0} };
 6      GLbyte ts2d[7][3] = { {0, 1, 8}, {1, 2, 8},
            {2, 7, 8}, {2, 3, 7}, {3, 6, 7},
 8          {3, 4, 6}, {4, 5, 6} };
        GLbyte tr2d[7] = {8, 0, 1, 2, 3, 4, 5};
10
        MYGLvertex v[18];
12      MYGLtriangle t[26];
```

```
14     for(int i=0; i<9; i++) {
           v[i].setVertex(v2d[i][0], v2d[i][1], 3.5);
16         v[i+9].setVertex(v2d[i][0], v2d[i][1], -3.5);
       }
18
       for(int i=0; i<7; i++) {
20         t[i].setTriangle(&v[ts2d[i][0]],
               &v[ts2d[i][1]], &v[ts2d[i][2]]);
22         t[i+7].setTriangle(&v[ts2d[i][2]+9],
               &v[ts2d[i][1]+9], &v[ts2d[i][0]+9]);
24     }

26     for(int i=0; i<6; i++) {
           t[i+14].setTriangle(&v[tr2d[i]+9],
28             &v[tr2d[i+1]+9], &v[tr2d[i]]);
           t[i+20].setTriangle(&v[tr2d[i]],
30             &v[tr2d[i+1]+9], &v[tr2d[i+1]]);
       }
32
       glPushMatrix();
34     glBegin(GL_TRIANGLES);
       for(int i=0; i<26; i++)
36         t[i].getGLDefinition();
       glEnd();
38     glPopMatrix();
   }
```

Do kabiny należy dodać jeszcze cztery słupki, co jest banalne i sprowadza się do odpowiednio przetransformowanych sześcianów wziętych prosto z GLUT (listing 5.9).

Listing 5.9. Funkcja opisująca słupki w kabinie Forda T.

```
 1 void drawSticks(void) {
       glPushMatrix();
 3     glTranslatef(0.25, 3.25, 5.5);
       glScalef(0.5, 0.5, 3.0);
 5     glutSolidCube(1.0);
       glPopMatrix();
 7
       glPushMatrix();
 9     glTranslatef(0.25, -3.25, 5.5);
       glScalef(0.5, 0.5, 3.0);
11     glutSolidCube(1.0);
       glPopMatrix();
13
       glPushMatrix();
15     glTranslatef(5.25, 3.25, 5.5);
       glScalef(0.5, 0.25, 3.0);
17     glutSolidCube(1.0);
```

```
        glPopMatrix();
19
        glPushMatrix();
21      glTranslatef(5.25, −3.25, 5.5);
        glScalef(0.5, 0.25, 3.0);
23      glutSolidCube(1.0);
        glPopMatrix();
25 }
```

Maska silnika składa się z dwóch odpowiednio przekształconych sześcianów, przy czym jej górna część została najpierw obrócona o 45 stopni po czym przeskalowana z różnymi wartościami współczynników dla każdego z kierunków, co spowodowało, że jedna ze ścian stała się rombem. Jeżeli chcemy, aby dla tych elementów prawidłowo były obliczane normalne, konieczne jest użycie `glEnable(GL_NORMALIZE)`.

Listing 5.10. Funkcja opisująca maskę Forda T.

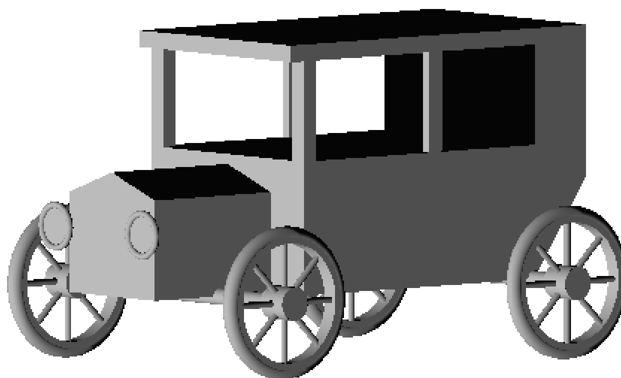
```
 1 \void drawEngine(void) {
        glPushMatrix();
 3      glTranslatef(−2.0, 0.0, 1.5);
        glScalef(5.0, 4.0, 3.0);
 5      glutSolidCube(1.0);
        glPopMatrix();
 7
        glPushMatrix();
 9      glTranslatef(−2.0, 0.0, 1.5);
        glScalef(5.0, 4.0, 3.0);
11      glutSolidCube(1.0);
        glPopMatrix();
13
        glPushMatrix();
15      glTranslatef(−2.0, 0.0, 3.0);
        glScalef(5.0, 2.8, 1.0);
17      glRotatef(45,1,0,0);
        glutSolidCube(1.0);
19      glPopMatrix();
   }
```

Do kompletu pozostało zamodelowanie świateł, co można praktycznie zostawić bez komentarza, bo kod za to odpowiedzialny jest prosty i, o ile uważaliście do tej pory, powinien być całkiem czytelny (patrz listing 5.11).

Listing 5.11. Funkcja opisując światło w Fordzie T.

```
   void drawLamp() {
 2      glPushMatrix();
        glRotatef(−90.0, 0, 1, 0);
 4      myDisk(0.0, 0.7, 16, 16);
```

```
        glutSolidTorus(.1, 0.65, 16, 16);
6       glRotatef(180.0, 0, 1, 0);
        myCylinder(0.7, 0.0, 0.5, 16, 4);
8       glPopMatrix();
  }
```



Rysunek 5.11. Gotowy model.

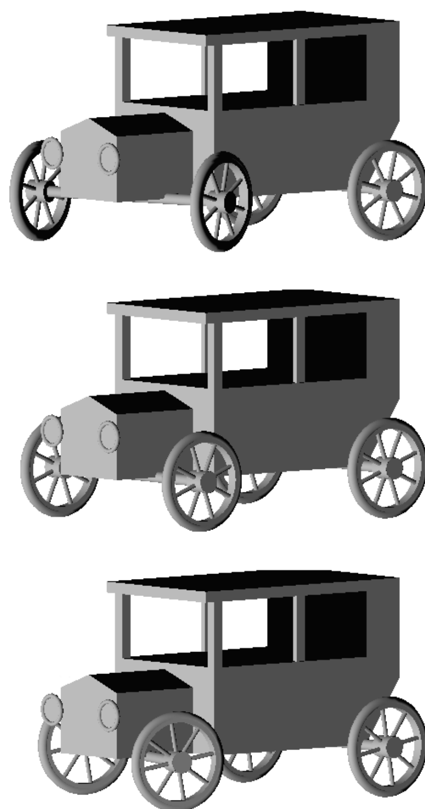
Jedyne, co nam pozostało, to złożyć przygotowane elementy w całość. Dodatkowo wykorzystamy fakt, że wprowadziliśmy hierarchię do modelu i w wywołaniu funkcji odpowiedzialnej za całość wirtualnego Forda T damy dwa argumenty. Pierwszy z nich opisuje skręt przedniej osi o zadany kąt. Oczywiście prawdziwy samochód przy pokonywaniu zakrętów dokonuje poprzez odpowiednie ustawienie kół, a nie całej osi, ale w dziecięcych zabawkach przedstawione rozwiązanie jest dość popularne, a nasz model jest, równie jak one, niepoważny. Drugi z argumentów odpowiada za obrót kół i osi symulując toczenie się po nawierzchni. w podobny sposób opisywane są modele na potrzeby gier komputerowych, oczywiście, ze znacznie rozbudowanymi zależnościami.

Listing 5.12. Kompletny opis modelu Ford T.

```
1       drawBody();
        drawSticks();
3       drawEngine();

5       glPushMatrix();
        glTranslatef(-5.0, 2.0, 2.0);
7       drawLamp();
        glPopMatrix();
9
```

```
        glPushMatrix();
11      glTranslatef(-5.0, -2.0, 2.0);
        drawLamp();
13      glPopMatrix();

15      /* tylna os */
        glPushMatrix();
17      glTranslatef(9,0,0);
        glRotatef(rotation, 0, 1, 0);
19      drawAxle();
        glPopMatrix();
21
        /* przednia os */
23      glPushMatrix();
        glTranslatef(-2,0,0);
25      glRotatef(direction, 0, 0, 1);
        glRotatef(rotation, 0, 1, 0);
27      drawAxle();
        glPopMatrix();
29 }
```

W jednym z kolejnych rozdziałów będzie okazja do wykorzystania stworzonego właśnie modelu, przy okazji zobaczymy, jak pewne jego elementy można opisać inaczej lub bardziej optymalnie.



Rysunek 5.12. Ford T w ruchu.

Rozdział 6

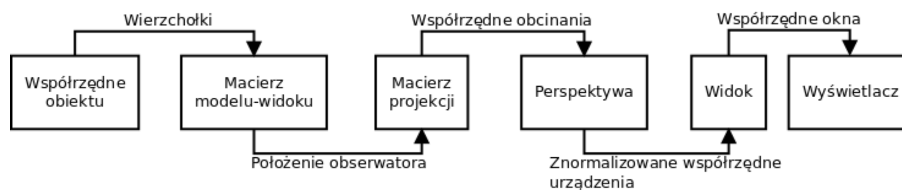
Rzutowania

6.1. Macierz projekcji **83**
6.2. Rzutowanie ortogonalne **84**
6.3. Rzutowanie perspektywiczne **84**
6.4. Obsługa rzutowań w OpenGL i GLU **86**
6.5. Pozycja i parametry kamery **88**

Rysunek 6.1. Satyra na fałszywą perspektywę. *William Hogarth 1753 r.*

W tym rozdziale zajmiemy się opisem sposobu, w jaki OpenGL wraz z bibliotekami pomocniczymi radzi sobie z symulacją kamery i takim przekształcaniem geometrii obiektów na potrzeby ich przyszłego wyświetlenia. Poznamy też kolejną macierz, tym razem projekcji i związany z nią stos. Będzie też okazja na przekazanie podstawowych informacji na temat perspektywy.

Wprowadzone podczas opisu sceny współrzędne wierzchołków wszystkich znajdujących się na niej obiektów w momencie renderowania widoku, który zostanie wyświetlony na ekranie użytkownika poddawane są szeregowi przekształceń. w pierwszym kroku są one wymnażane przez macierz modelu-widoku wygenerowaną dla każdego z nich w oparciu o zdefiniowane operacje przekształceń geometrycznych i operacje odkładania i zdejmowania ze stosu. w ramach tej samej macierzy definiowane położenie obserwatora.



Rysunek 6.2. Etapy przetwarzania współrzędnych wierzchołków obiektu na współrzędne ekranowe.

Kolejny krok korzysta już z macierzy projekcji i opierając się na niej dokonywana jest operacja rzutowania z przestrzeni trój do dwuwymiarowej. z parametrów tworzących tą macierz wynikają też współrzędne bryły obcinania, która przy rzucie ortogonalnym jest prostopadłościanem, a przy perspektywicznym ściętym ostrosłupem. Następnie uzyskane współrzędne zostają znormalizowane i przeskalowane do rozmiarów okna, w którym ma miejsce wyświetlanie.

6.1. Macierz projekcji

Obok wspomnianej już macierzy modelu-widoku, w OpenGL używana jest jeszcze macierz projekcji. o ile korzystając z tej pierwszej, mieliśmy możliwość zmiany rozmiaru i położenia elementów na scenie, to użycie macierzy projekcji umożliwia zdefiniowanie sposobu rzutowania sceny w procesie jej przygotowania do wyświetlenia na ekranie.

Przy okazji wprowadzania pojęcia współrzędnych jednorodnych wspomniane zostało, że ich użycie w grafice trójwymiarowej wynika z konieczności przeliczenia współrzędnych wierzchołków obiektów sceny na reprezentujące je współrzędne na płaszczyźnie. OpenGL standardowo oferuje tylko dwie metody rzutowania. Ortogonalną, zwaną też prostopadłą oraz perspektywiczną. Jeżeli konieczne jest użycie specjalnych sposobów rzutowania jest to możliwe poprzez zdefiniowanie własnych macierzy z wykorzystaniem funkcji `glLoadMatrix()`, do której przekazuje się jako parametr tablicę 4x4.

Analogicznie do macierzy modelu-widoku, tu też mamy do dyspozycji stos, jednak o mocno ograniczonym, bo dwuelementowym, rozmiarze. Umożliwia to chwilową zmianę sposobu lub parametrów projekcji sceny. Przykładowo w oprogramowaniu architektonicznym może być on użyty do wyświetlenia wizualizacji wraz z podglądem prostopadłym ścian budynku czy jego dachu.

Przełączenie pomiędzy aktywnym stosem i macierzą przekształceń dokonuje się z wykorzystaniem `glMatrixMode()`, podając jako parametr w jej

wywołaniu stałą GL_MODELVIEW gdy zależy nam na pracy w trybie model-widok i GL_PROJECTION przy trybie projekcji.

6.2. Rzutowanie ortogonalne

Najprostszy i historycznie pierwszy sposób przedstawiania widzianej przez artystę rzeczywistości, bardzo mocno przypomina efekt uzyskiwany w procesie rzutowania ortogonalnego. Wszystkie osoby i obiekty znajdujące się na takich przedstawieniach mają rozmiary do siebie proporcjonalne niezależnie od odległości od obserwatora. Głębie sceny starano się oddać poprzez umieszczenie dalszych planów wyżej i jako częściowo przysłonięte.

Ten sposób rzutowania jest wykorzystywany po dzień dzisiejszy przez inżynierów. Zarówno w architekturze, jak i przy projektowaniu przemysłowym, obiekty trójwymiarowe są przedstawiane na rozrysach technicznych jako rzuty boczne. Podobnie jeżeli przyjrzycie się dokumentacjom serwisowym tam też wszystkie ilustracje przedstawiające sposób montażu urządzenia są pozbawione perspektywy i rzutowane aksjometrycznie.

Macierz definiująca to przekształcenie od strony matematycznej przedstawia się ona następująco:

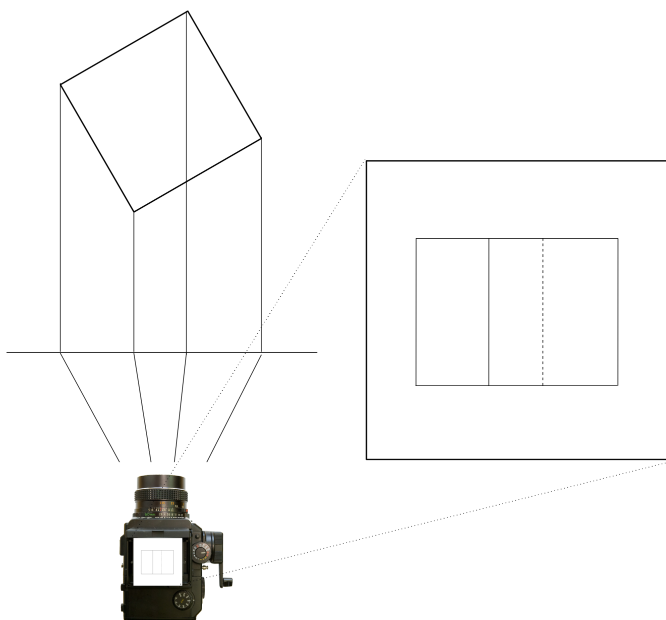
$$M_p = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

gdzie $left$, $right$, top, $bottom$, $near$ i far, są parametrami opisującymi współrzędne płaszczyzn obcinania.

Schematycznie proces rzutowania ortogonalnego prezentuje rysunek 6.3. Przyjmijmy, że obiektem jest sześcian, którego górną ścianę widzimy. Rzutowanie odbywa się na ekran, na który patrzy obserwator. Wszystkie linie biegnące od wierzchołków są prostopadłe do ekranu, wynika z tego też ich równoległość względem siebie. Na tak uzyskanym rzucie wszystkie linie równoległe do ekranu zachowują swoją długość.

6.3. Rzutowanie perspektywiczne

Na temat momentu, w którym perspektywa w pełni zaistniała w sztuce, spór pomiędzy historykami sztuki trwa od bardzo dawna. Wiadomo, że już



Rysunek 6.3. Ortogonalny rzut sześcianu.

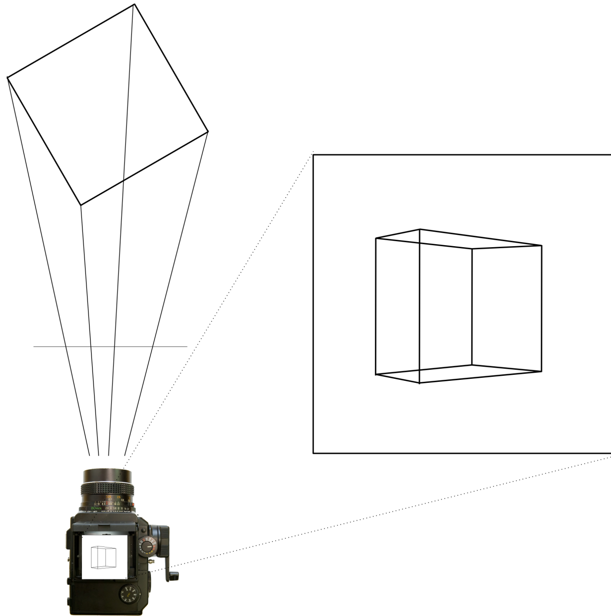
w okresie starożytnym były próby przedstawiania scen, w których poszczególne ich elementy były w zależnościach geometrycznych wynikających z ich wzajemnego położenia. Jednak dopiero renesans uważany jest za okres, kiedy dokonano formalnego opisu podstawowych typów perspektywy i zaczęto z nich w sposób w pełni uświadomiony korzystać. w sztuce zwykło się korzystać z podziału na trzy podstawowe typy, ale z opisu matematycznego wynika, że są to szczególne przypadki perspektywy trzypunktowej.

OpenGL ma zdefiniowaną macierz projekcji perspektywicznej w sposób następujący:

$$M_p = \begin{bmatrix} \frac{2n}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2n}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2f}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

gdzie, jak miało to miejsce w przypadku rzutowania ortogonalnego, $left$, $right$, top, $bottom$, $near$ i far, są opisem płaszczyzn obcinania.

Rysunek 6.4 w sposób schematyczny przedstawia powstanie obrazu sześcianu w rzucie perspektywicznym.



Rysunek 6.4. Perspektywiczny rzut sześcianu.

6.4. Obsługa rzutowań w OpenGL i GLU

Będąc w trybie pracy z macierzą projekcji w OpenGL, mamy możliwość ustalić parametry i sposób rzutowania sceny na płaszczyznę za pomocą odpowiednich funkcji. Dokonujemy tego, korzystając z trybu macierzy projekcji.

W przypadku rzutu prostopadłego stosuje się w tym celu funkcję `glOrtho()`, której zadaniem jest ustawienie skopiowanie odpowiedniej macierzy rzutowania. Jako parametry dla argumentów jej wywołania podaje się kolejno wartości współrzędnych odpowiedzialnych za prostopadłościan rzutu kolejno po dwie wartości dla współrzędnych x, y i z. Czyli, tak jak miało to miejsce w formalnym opisie, daje nam to szóstkę liczb wyrażonych w jednostkach świata w kolejności `lewa`, `prawa`, `góra`, `dół`, `blisko` i `daleko`.

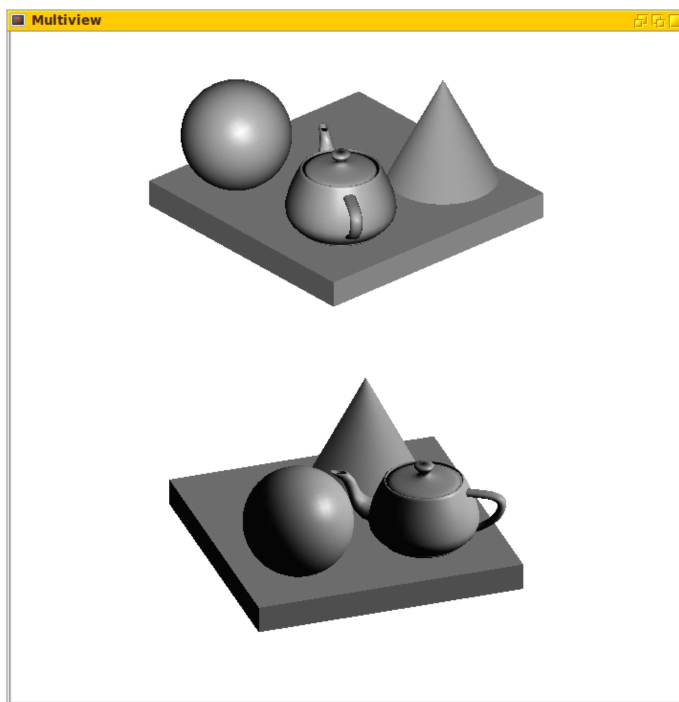
Na listingu 6.2 widoczne jest, że dwa argumenty maja wartość wyliczaną z wykorzystaniem wartości zmiennych `w` i `h`. Stosunek wysokości i szerokości wyświetlanego obrazu to proporcja, używa się też terminu *aspekt*, obrazu.

Takie działanie daje nam pewność zachowania poprawności geometrycznej obrazu niezależnie od zmian wymiarów okna, w którym ma miejsce wyświetlanie.

`glViewport()` jest funkcją definiującą nam powierzchnię obszaru wyświetlania kontekstu renderowania OpenGL wyrażona w pikselach. w przedstawionym przypadku zajmuje ona całą powierzchnię okna, ale nic nie stoi na przeszkodzie, aby był to jego fragment, co daje nam możliwość definiowania wielu różnych widoków dla jednej sceny jak na rysunku 6.5.

Listing 6.1. Ustawienie rzutu ortogonalnego.

```
1 glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
3 glViewport(0, 0, w, h);
  if(w<h)
5     glOrtho(-1.0f, 1.0f, -1.0f*h/w, 1.0f*h/w, 1.0f, 40.0f);
  else
7     glOrtho(-1.0f*w/h, 1.0f*w/h, -1.0f, 1.0f, 1.0f, 40.0f);
  glMatrixMode(GL_MODELVIEW);
```



Rysunek 6.5. Program wykorzystujący dwa różne widoki zdefiniowane z użyciem `glViewport()`.

Odpowiednikiem `glOrtho()` dla rzutu perspektywicznego jest funkcja `glFrustrum()`, która definiuje ostrosłup widzenia. Jak widać, na rysunku 6.6 jest on opisany analogicznymi parametrami.

Listing 6.2. Ustawienie rzutu perspektywicznego.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glViewport(0, 0, w, h);
if(w<h) {
    glFrustum(-1.1, 1.1, -1.1f*h/w, 1.1f*h/w, 1.0f, 20.0f);
} else {
    glFrustum(-1.1f*w/h, 1.1f*w/h, -1.1f, 1.1f, 1.0f, 20.0f);
}
glMatrixMode(GL_MODELVIEW);
```

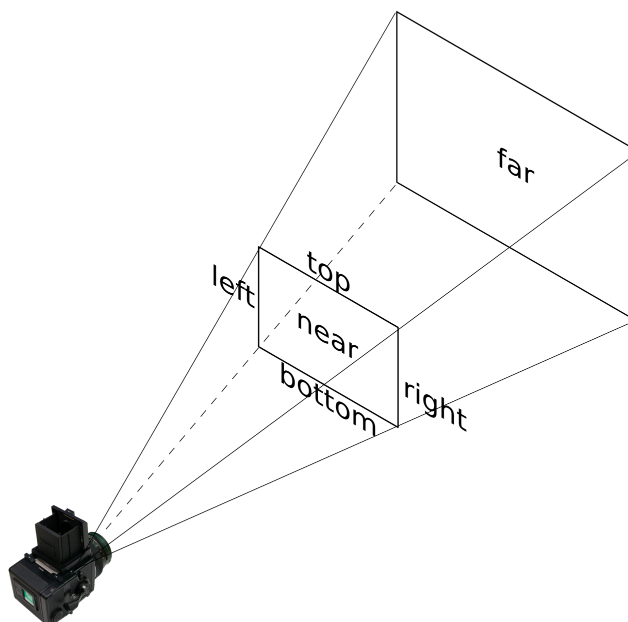
Mimo że parametry wywołania `glOrtho()` i `glFrustrum()` są identyczne, to podmienienie w kodzie jednej z tych funkcji na drugą jest pierwszym krokiem do zmiany używanego sposobu wyświetlania. Konieczne będzie też uwzględnienie konieczności modyfikacji położenia kamery, czy kąta patrzenia. Niezalecane jest dokonywanie korekcji poprzez skalowanie całej sceny, ponieważ ma to negatywny skutek wynikający ze zmiany wartości normalnych.

Dużo wygodniejszym, a na pewno bardziej intuicyjnym, sposobem zmiany parametrów rzutowania perspektywicznego, jest użycie funkcji, znajdującej się w bibliotece GLU, `gluPerspective()`. Przyjmuje ona w swoim wywołaniu cztery argumenty i kolejno jest to kąt widzenia wirtualnej kamery, proporcje obrazu i bliskie oraz dalekie współrzędne płaszczyzn obcinania. Zmiana kąta widzenia w fotografii odpowiada użyciu obiektywu o innej ogniskowej.

Należy pamiętać, że używamy jednej z wymienionych funkcji co wynika z faktu, że każda z nich dokonuje przemnożenia aktualnie używanej macierzy projekcji przez określone wartości wynikające z wybranego sposobu rzutowania.

6.5. Pozycja i parametry kamery

W odróżnieniu od rzutowania informacje tak o pozycji jak i kierunku patrzenia kamery ustawiane są w ramach przekształceń macierzy model-widok. Startowo kamera jest umiejscowiona w początku układu współrzędnych i zmiana jej pozycji względem sceny możliwa jest poprzez użycie operacji przekształceń, które już poznaliśmy.



Rysunek 6.6. Ostrosłup widzenia.

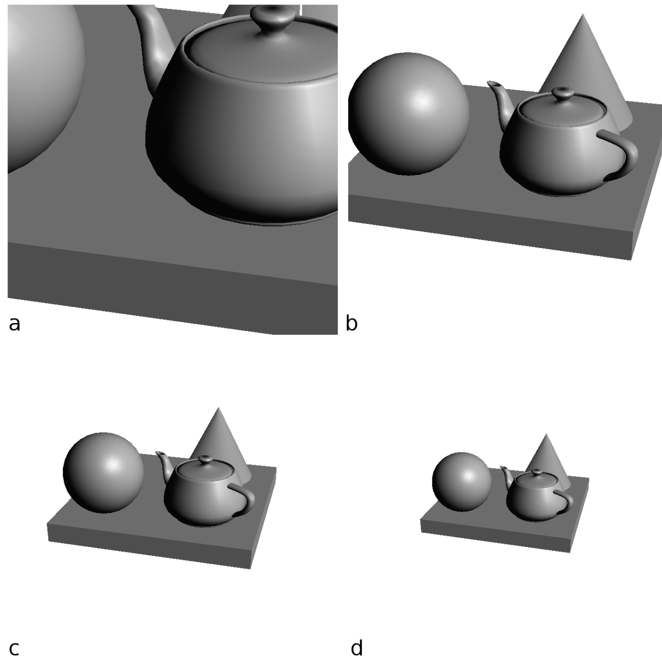
W większości przypadków znaczenie wygodniejsze jest w tym celu skorzystanie z funkcji biblioteki GLU `gluLookAt()`. Jej argumenty to trzy grupy współrzędnych. Pierwsza opisuje pozycję kamery, druga punkt, na który jest skierowana i ostatnia wyznacza wektor orientujący górę obrazu.

Listing 6.3. Użycie `gluLookAt()`.

```
1 glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
3 gluLookAt(0.0, 1.0, −3.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

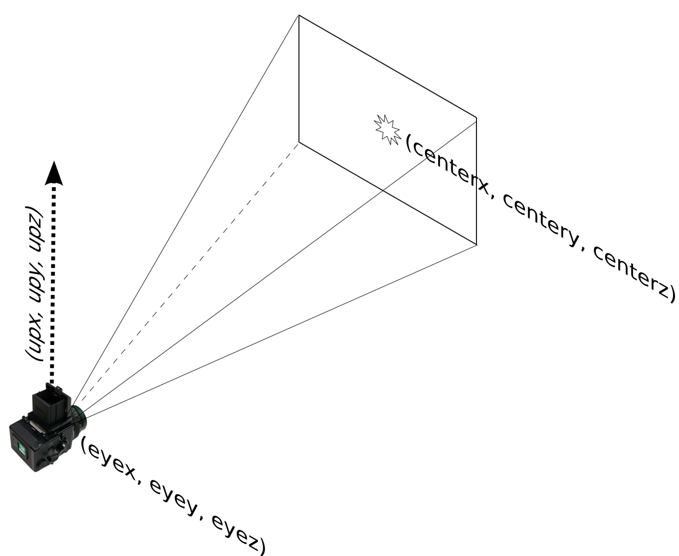
Ponieważ w OpenGL nie ma osobnych macierzy przekształceń dla kamery, to `gluLookAt()`, de facto, jest funkcją wykonującą określone przekształcenia na macierzy modelu-widoku. Wynika z tego konieczność umiejscowienia jej w odpowiednim miejscu kodu, tak, aby wszystkie operacje obrotu i przesunięcia dotyczyły się każdego z umieszczonych na scenie obiektów. z reguły wywołuje się ją zaraz po wejściu w tryb pracy z macierzą sceny i załadowaniu do niej macierzy jednostkowej (patrz listing 6.3).

Podsumowując. Ustawiając rzutowanie, parametry i pozycję kamery, należy pamiętać o wybraniu odpowiedniej macierzy, na której wykonywane



Rysunek 6.7. Wynik generacji obrazu dla różnych wartości kąta widzenia kamery, a) 10, b) 20, c) 30 i d) 40 stopni.

będą operacje. w przypadku funkcji ustalających sposób rzutowania, czyli `glOrtho()`, `glFrustrum()`, `gluPerspective()`, a także `glVieport()`, zawsze jest to macierz projekcji. Pozycjonowanie kamery na scenie odbywa się zaś w ramach macierzy model-widok.



Rysunek 6.8. Poszczególne współrzędne związane z funkcją `gluLookAt()`.

Rozdział 7

Przygotowanie sceny, listy wyświetlania i tablice wierzchołków

7.1. Czyszczenie sceny **94**
7.2. Tryb cieniowania **96**
7.3. Wielokrotne próbkowanie **97**
7.4. Listy wyświetlania **97**
7.5. Tablice wierzchołków **100**

Rysunek 7.1. Teatr jarmarczny, fragment. *Piotr Bruegel młodszy*

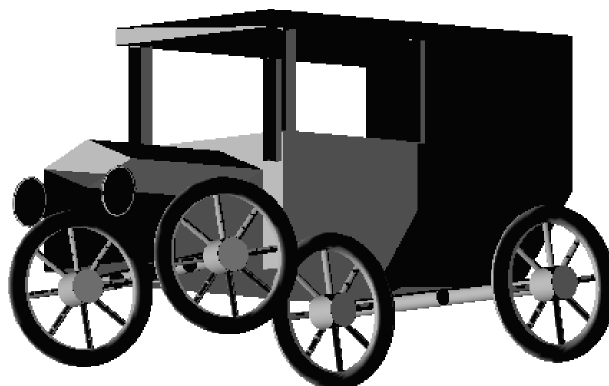
.

Ten rozdział zostanie poświęcony pozostałym aspektom związanym z modelami i ich umieszczaniem na naszej wirtualnej scenie. Pokazane zostanie, w jaki sposób można mieć wpływ na jakość generowanego obrazu. w drugiej części zostaną przedstawione alternatywne metody przechowywania informacji o naszych modelach.

7.1. Czyszczenie sceny

Proces przygotowania sceny przed umieszczeniem na niej modeli rozpoczyna się od wielkiego sprzątania. Dal uzyskanie pustej przestrzeni do pracy konieczne jest wyczyszczenie używanych buforów. OpenGL ma możliwość konfiguracji, które z nich są używane. Warto pamiętać, że jeżeli z jakiejś funkcjonalności nie korzystamy, to ze względu na wydajność powinna być ona wyłączona. Ponieważ taka filozofia jest też skojarzona z samą biblioteką to obsługa buforów, poza buforem ramki, jest domyślnie wyłączona. Ponieważ nasze sceny są trójwymiarowe i chcemy, żeby poprawnie były generowane, konieczne jest włączenie dodatkowo obsługi głębi w przeciwnym razie obiekty na scenie będą rysowane w kolejności ich opisywania, a nie

położeniem w przestrzeni (patrz rysunek 7.2). Dokonujemy tego, korzystając z funkcji `glEnable()`, która już się pojawiła w poprzednich rozdziałach co najmniej dwukrotnie, ale gwoli przypomnienia wspomnę, że przyjmuje ona jako argument wywołania stałą definiującą typ włączanej funkcjonalności. w przypadku buforów głębi i szablonów są one zdefiniowane jako `GL_DEPTH_TEST` i `GL_STENCIL_TEST`. Jeżeli używamy GLUT, to konieczne będzie również ustawienie dla trybu wyświetlania opcji `GLUT_DEPTH`.



Rysunek 7.2. Model narysowany z wyłączoną obsługą bufora głębokości.

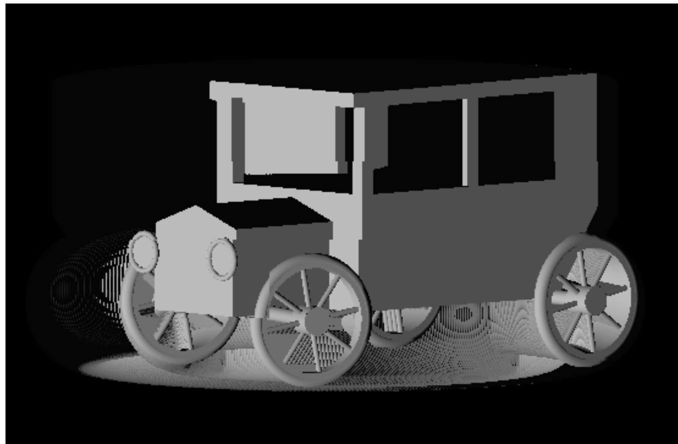
.

Możemy zająć się mechanizmem czyszczenia buforów. Wywoływany jest on za pośrednictwem funkcji `glClear()`, która jednoznacznie wskazuje na te z nich, które mają zostać uwzględnione podczas operacji. Używamy w tym celu zdefiniowanych flag bitowych reprezentowanych przez stałe `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT` i `GL_STENCIL_BUFFER_BIT`, które odpowiadają kolejno za bufory ramki, głębi i szablonu. w wywołaniu podaje się ich sumę logiczną.

OpenGL daje nam możliwość zdefiniowania wartości, którą zostają bufory wypełnione, z użyciem funkcji `glClearColor()`, `glClearDepth()` i `glClearStencil()`. Nas na ten moment interesuje tylko pierwsza z nich, ponieważ pozwala na zdefiniowanie koloru tła niż domyślne czarne.

Listing 7.1. Włączenie obsługi i czyszczenie bufora ramki i głębokości w OpenGL.

```
1 void initOpenGL{
      glClearColor(1.0, 1.0, 1.0, 0.0);
3     glEnable(GL_DEPTH_TEST);
```



Rysunek 7.3. Obracający się model narysowany bez czyszczenia bufora ramki.

.

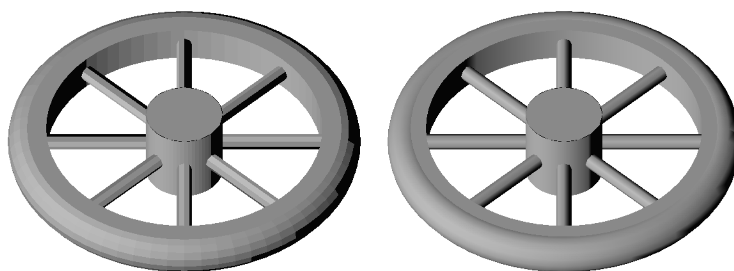
```
    /* ... */
5 }

7 void drawScene(void){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
9   /* ... */
  }
```

Tutaj ważna uwaga, jeżeli jakiś tryb lub opcja maszyny stanów OpenGL ma być permanentny dla całego czasu działania aplikacji to powinien zostać ustawiony raz w funkcji odpowiedzialnej za przygotowanie środowiska, tak jak ma to miejsce na listingu 7.1. Dużym błędem jest konfigurowanie koloru tła, czy włączanie obsługi bufora głębokości w wywoływanej najczęściej kilkadziesiąt razy na sekundę głównej pętli OpenGL ponieważ w ten sposób niepotrzebnie marnujemy czas procesora.

7.2. Tryb cieniowania

OpenGL oferuje w przypadku standardowego potoku dwa różne tryby cieniowania obiektów. Mamy możliwość przełączania się pomiędzy nimi z wykorzystaniem funkcji `glShadeModel()`. Domyślny tryb `GL_SMOOTH` oblicza kolor dla każdego z fragmentów, interpolując informację dla poszczególnych wierzchołków. w przypadku użycia `GL_FLAT` każdy z fragmentów danego trójkąta lub wielokąta ma ten sam kolor wynikający z uśrednienia wartości przypisanych dla wszystkich jego wierzchołków. Przykład użycia

obu sposobów cieniowania przedstawia rysunek 7.4. Warto też wspomnieć, że za pomocą programów fragmentów możliwe jest zdefiniowanie również własnych sposobów cieniowania, jednak wykracza to poza zakres niniejszej pozycji.



Rysunek 7.4. Cieniowanie płaskie i wygładzone.

.

7.3. Wielokrotne próbkowanie

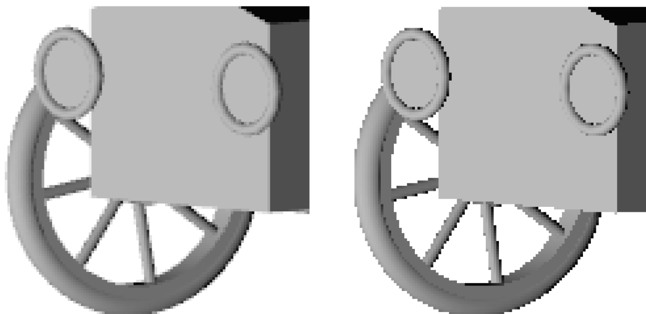
Typowo obraz na poziomie bufora ramki jest generowany z fragmentów, które są kojarzone jeden do jednego z poszczególnymi pikselami obrazu wynikowego. Powoduje to powstanie ostrych przejść na krawędziach wszystkich prymitywów. Część implementacji OpenGL obsługuje tryb wielokrotnego próbkowania, gdzie fragmenty są kojarzone z subpikselami. Rozwiązanie to znacznie wpływa na poprawę wizualną wygenerowanej sceny, ale zwiększa też obciążenie karty.

Włączenie obsługi tej opcji jest stosunkowo proste. Wymaga, jeżeli korzystamy z biblioteki GLUT, dodania w trakcie inicjacji trybu okna flagi `GLUT_MULTISAMPLE` i włączenie jego obsługi na maszynie stanów poprzez wywołanie `glEnable()` z parametrem `GL_MULTISAMPLE`.

Grafika 7.5 przedstawia powiększenie wycinka ekranu z obrazem generowanym z użyciem wielokrotnego, dokładnie czterokrotnego, próbkowania i bez.

7.4. Listy wyświetlania

Jeżeli spodziewacie się, że w opisie waszej sceny wielokrotnie zostanie użyty ten sam obiekt, to obok możliwości zamknięcia go w funkcję istnieją inne rozwiązania dostępne jako mechanizmy OpenGL. w wielu wypadkach mają tę zaletę, że w zależności do złożoności tak sceny jak i obiektów,



Rysunek 7.5. Ośmiokrotne powiększenie obrazu wygenerowanego z włączoną obsługą wielokrotnego próbkowania i bez.

.

których się tyczą, mogą w sposób znaczący wpłynąć na wydajność całego systemu.

Jedną z takich metod jest tworzenie list wyświetlania, które zawierają skompilowany opis modelu. Dodatkowo w przypadku, gdy korzystamy z OpenGL w architekturze klient-serwer, rozwiązanie to znacznie poprawia komunikację sieciową pomiędzy końcówkami uczestniczącymi w transmisji. Wynika to z faktu, że lista jest przechowywana po stronie serwera, a klient wysyła jedynie żądanie jej użycia.

Pierwszym krokiem przy tworzeniu listy wyświetlania jest zadeklarowanie zmiennej typu `GLuitn`, która będzie pełniła funkcję uchwytu. Następnie dokonujemy generacji pustych list i przypisani ich do uchwytu za pomocą funkcji `glGenLists()`. Jako argument jej wywołania przekazujemy liczbę list, które chcemy, aby zostały przygotowane. Wynik zwracany to nazwa. w postaci liczby, pierwszej z list, kolejne są opisywane wartościami o zwiększającymi się o jeden.

Teraz przechodzimy do opisu naszego modelu poprzedzając go funkcją `glNewList()` tworząca nową listę. w jej wywołaniu podajemy dwa argumenty, spośród których pierwszy jest uprzednio zaalokowanym uchwytem do listy. Drugi informuje w jakim trybie lista ma zostać stworzona. Mamy w tym wypadku następujące możliwości: `GL_COMPILE` dokonujące dopisanie do listy poszczególnych jej składowych w postaci opisu obiektu i `GL_COMPILE_AND_EXECUTE`, gdzie dodatkowo ma miejsce automatyczne wywołanie listy na maszynie stanów co spowoduje dodanie do sceny modelu w niej opisanego. Miejscem, w którym lista się kończy, jest wywołanie funkcji `glEndList()`.

Wywołanie takiej listy odbywa się z wykorzystaniem `glCallList()`, której jedyny argument to uchwyt do listy.

W tym miejscu skorzystamy ponownie z modelu Forda T, który został przedstawiony w rozdziale 5. Znajduje się tam funkcja `drawWheel()`, która jest wywoływana w sumie cztery razy za każdym razem, gdy konieczne jest utworzenie kół modelu. Mamy dwie możliwości. Pierwszą jest jej modyfikacja tak, aby proces tworzenia listy wyświetlania był w niej zawarty. Druga, z której skorzystamy i wynika z faktu, że funkcja ta zawiera tylko dozwolone operacje sterujące maszyny stanów OpenGL, będzie jej wywołaniem wewnątrz bloku definiowania listy (patrz listing 7.2). `genWheelAsList()` powinno być wykonane w czasie działania aplikacji raz, więc wskazane byłoby umieszczenie jej wywołania w fragmencie kodu odpowiedzialnego za inicjację maszyny stanów.

Listing 7.2. Utworzenie listy wyświetlania dla koła.

```
   GLuint wheel;
 2 /* ... */
   void genWheelAsList(void) {
 4     wheel = glGenLists(1);
       glPushAttrib(GL_ALL_ATTRIB_BITS);
 6     glNewList(wheel, GL_COMPILE);
       drawWheel();
 8     glEndList();
       glPopAttrib();
10 }
```

W kodzie dodatkowo pojawiają się funkcje `glPushAttrib()` i `glPopAttrib()`, których działanie jest analogiczne do `glPushMatrix()` i `glPopMatrix()` z tą różnicą, że odkładają wartości ustawionych atrybutów maszyny stanów. To, które z nich, kolor, parametry tekstury, tryby i wiele innych, mają być w ten sposób chronione przed zmianą występującą w liście wyświetlania decyduje przekazana flaga bitowa. w zaprezentowanym przypadku stała `GL_ALL_ATTRIB_BITS` odnosi się do wszystkich możliwych atrybutów.

Przykładowe wywołanie listy jest zaprezentowane na listingu 7.3 w zmodyfikowanej funkcji z rozdziału 5.

Listing 7.3. Wykorzystanie listy wyświetlania.

```
   GLuint wheel;
 2 void drawAxle(void) {
       glPushMatrix();
 4     glRotatef(90, 1, 0, 0);
       glTranslatef(0.0, 0.0, -5.0);
 6     myCylinder(0.25, 0.25, 10, 64, 4);
       glPopMatrix();
 8
       glPushMatrix();
```

```
10      glTranslatef(0.0, -5.0, 0.0);
        glCallList(wheel);
12      glPopMatrix();

14      glPushMatrix();
        glTranslatef(0.0, 5.0, 0.0);
16      glCallList(wheel);
        glPopMatrix();
18  }
```

7.5. Tablice wierzchołków

Istnieje jeszcze jedna metoda przechowywania i modyfikacji skomplikowanych modeli. w ramach OpenGL są do dyspozycji mechanizmy obsługi tablic zawierających opis geometrii i właściwości charakterystycznych dla poszczególnych wierzchołków obiektu. w tym celu bazując na opisie nadwozia Forda T dokonamy pewnych modyfikacji w reprezentacji, tak aby móc z tych rozwiązań skorzystać.

Obsługa tablic jest możliwa po ich aktywacji z użyciem funkcji `glEnableClientState()` wywołanej z parametrem wskazującym na typ. Niestety ubocznym efektem użycia tego rozwiązania jest fakt przechowywania danych po stronie klienta, stąd pochodzi historycznie nazwa funkcji. Wyłączenie obsługi odbywa się z użyciem `glDiableClientState()` z identycznymi argumentami.

W naszym przypadku włączymy obsługę użycia tablic dla wierzchołków i normalnych. Nie jest konieczne zmienienie organizacji tablicy wierzchołków na postać jednowymiarową, ponieważ do maszyny stanów jest przekazywany wskaźnik do danych, a ich układ w pamięci w obu wypadkach jest jednakowy. Wskazanie na tablicę zawierającą dane obywa się z użyciem specjalizowanych funkcji. Dla wierzchołków jest to `glVertexPointer()`, a dla normalnych `glNormalPointer()`. Zebrana informacja dla wszystkich typów danych oraz przyjmowanych argumentach zawiera tabela 7.2. Przy wywołaniu `size` informuje o liczbie składowych wektorów, `type` opisuje typ zmiennych, w którym przechowywane są dane i może przyjąć następujące wartości `GL_SHORT`, `GL_INT`, `GL_FLOAT`, or `GL_DOUBLE`. `Stride` domyślnie ma wartość zero, co oznacza, że kolejne wartości w tablicy są ułożone jedna po drugiej. Inne wartości przyjmuje ten argument w sytuacji, gdy korzystamy z tablic z przeplotem, w których znajdują się zapisane dane nie tylko dla pozycji wierzchołka, ale również jego normalnej, czy koloru.`Pointer` to oczywiście wskaźnik do tablicy zawierającej dane.

Tabela 7.1. Zdefiniowane stałe przyjmowane jako argument funkcji `glEnableClientState()` określające typ aktywowanej tablicy.

Nazwa typu	Przechowywana informacja
GL_VERTEX_ARRAY	Tablica wierzchołków.
GL_COLOR_ARRAY	Tablica koloru wierzchołka.
GL_SECONDARY_COLOR_ARRAY	Tablica koloru drugorzędowego wierzchołka dla OpenGL w wersji 1.4 lub nowszej.
GL_INDEX_ARRAY	Tablica indeksów kolorów w trybie ze zdefiniowaną paletą.
GL_NORMAL_ARRAY	Tablica wartości normalnych dla wierzchołka.
GL_FOG_COORD_ARRAY	Tablica współrzędnych z dla mgły skojarzonej z obiektem.
GL_TEXTURE_COORD_ARRAY	Tablica współrzędnych mapowania tekstury.
GL_EDGE_FLAG_ARRAY	Tablica definiująca ukrywanie krawędzi.

Obsługa danych opisanych w ten sposób może się odbywać na kilka sposobów. Trzeba tu zaznaczyć, że przedstawione za chwilę funkcje w momencie wywołania przy pracy w architekturze klient-serwer powodują przesłanie informacji do maszyny, która jest odpowiedzialna za rendering sceny.

Najprostszą metodą jest odwołanie się do danych poszczególnych wierzchołków poprzez ich indeks, który jest przekazywany jako argument wywołania funkcji `glArrayElement()`. Widać to na listingu 7.4. Gdybyśmy zajrzeli, do tego co się dzieje wewnątrz, okaże się, że pojedyncze wywołanie wspomnianej funkcji owocuje faktycznie wywołaniem `glVertex()` i `glNormal()`. Jeżeli uruchomiona będzie obsługa większej ilości typów tablic, wywołanie będzie adekwatnie bardziej rozbudowane. Już w tym momencie widać, że taka organizacja danych daje duże oszczędności związane z samą konstrukcją kodu aplikacji i sprzyja mniejszemu prawdopodobieństwu popełnienia błędów. Warto pamiętać, że standardowo dane na temat wierzchołków, tekstur czy normalnych w większości przypadków pochodzić będą z plików zewnętrznych, a nie jak w przykładzie, ze statycznie zdefiniowanych tablic.

Listing 7.4. Opis ścian nadwozia modelu Forda T oparty na tablicach wierzchołków i normalnych.

```
  void drawBody(void) {
2     GLdouble v2d[18] =
          { 0.0, 0.0, 9.0, 0.0, 11.0, 3.0,
```

```
4              11.0, 7.5, -0.5, 7.5, -0.5, 7.0,
               9.0, 7.0, 9.0, 4.0, 0.0, 4.0 };
6       GLdouble n[54];
        GLbyte ts2d[7][3] = { {0, 1, 8}, {1, 2, 8},
8           {2, 7, 8}, {2, 3, 7}, {3, 6, 7},
            {3, 4, 6}, {4, 5, 6} };
10      GLbyte tr2d[7] = {8, 0, 1, 2, 3, 4, 5};

12      glEnableClientState(GL_NORMAL_ARRAY);
        glEnableClientState(GL_VERTEX_ARRAY);
14
        glVertexPointer(2, GL_DOUBLE, 0, &v2d);
16      glNormalPointer(GL_DOUBLE, 0, &n);

18      for(int i=0;i<18;i++){
            n[i*3]=0.0;
20          n[i*3+1]=0.0;
            n[i*3+2]=-1.0;
22      }

24      glPushMatrix();
        glRotatef(90,1,0,0);
26      glTranslatef(0.0, 0.0, -3.5);
        glBegin(GL_TRIANGLES);
28      for(int i=0;i<7;i++)
            for(int j=0;j<3;j++)
30              glArrayElement(ts2d[i][j]);
        glEnd();
32      glPopMatrix();
        /* ... */
34  }
```

Drugą z funkcji, które umożliwiają odwołania do danych wierzchołkowych jest `glDrawElements()`. Różni się tym od `glArrayElement()`, że daje kompleksowe odwołanie do grupy kolejnych wierzchołków składających się na konkretny prymityw. Do dyspozycji są tu znane już z rozdziału 4 zdefiniowane stałe `GL_POINTS`, `GL_LINES`, `GL_TRIANGLES` i pozostałe z listy prymitywów, które przekazywane są do funkcji jako pierwszy jej argument. Drugim jest liczba wskazań indeksowych określona dla jednego wystąpienia prymitywu. Trzecim parametrem wykorzystywanym przy wywołaniu jest informacja o typie całkowitym zawartym w tablicy indeksów przekazywanej jako ostatni z argumentów. Wartości, które są dopuszczone w tym wypadku, to `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`.

Na listingu 7.5 przedstawiony jest fragment odpowiedzialny za tworzenie ścian bocznych nadwozia modelu Forda korzystający z opisywanej funkcji. Trzeba zwrócić tu uwagę, że korzystając listy nie ma miejsca wywołanie `glBegin()glEnd()`.

Listing 7.5. Użycie `glDrawElements()`.

```
void drawBody(void) {
    /* ... */
    glPushMatrix();
    glRotatef(90,1,0,0);
    glTranslatef(0.0, 0.0, -3.5);
    for(int i=0;i<7;i++)
        glDrawElements(GL_TRIANGLES, 3,
            GL_UNSIGNED_BYTE, &ts2d[i]);
    glPopMatrix();
    /* ... */
}
```

Można pójść dalej, pominąć pętlę dodającą kolejne trójkąty i przekazać do funkcji kompletną listę wszystkicj prymitywów. w naszym przypadku będzie to wyglądało w sposób następujący:

```
glDrawElements(GL_TRIANGLES, 21, GL_UNSIGNED_BYTE, &ts2d);
```

Warto tu zaznaczyć, że istnieje jeszcze wariant tej funkcji pozwalający na odwołania do wielu tablic wierzchołków za jednym wywołaniem. Nazywa się ona `glMultiDrawElements()`.

Tabela 7.2. Funkcje wskazujące na tablicowe dane modelu.

Funkcje	Argumenty
`glColorPointer()`	GLint size GLenum type GLsizei stride, const GLvoid *pointer
`glEdgeFlagPointer()`	GLsizei stride, const GLvoid *pointer
`glFogPointer()`	GLenum type GLsizei stride, const GLvoid *pointer
`glIndexPointer()`	GLenum type GLsizei stride, const GLvoid *pointer
`glNormalPointer()`	GLenum type GLsizei stride, const GLvoid *pointer
`glSecondaryColorPointer()`	GLint size GLenum type GLsizei stride, const GLvoid *pointer
`glTexCoordPointer()`	GLint size GLenum type GLsizei stride, const GLvoid *pointer
`glVertexPointer()`	GLint size GLenum type GLsizei stride, const GLvoid *pointer

Rozdział 8

Model oświetlenia

8.1. Światło w OpenGL **107**
8.1.1. Światło otoczenia 109
8.1.2. Światło rozproszenia 109
8.1.3. Światło rozbłysku 109
8.2. Modyfikacje świateł **110**
8.2.1. Kontrola spadku intensywności świecenia . . . 110
8.2.2. Reflektor 111
8.2.3. Wiele źródeł światła 112
8.3. Materiał **112**
8.3.1. Emisyjność obiektów 113
8.3.2. Rozbłysk na powierzchni obiektów 113
8.3.3. Pozostałe parametry 114
8.3.4. Alternatywna zmiana parametrów materiału . . 114

Rysunek 8.1. Filozof medytujący. *Rembrandt van Rijn 1632r.*

Kolejny nasz rozdział poświęcimy w całości modelowaniu oświetlenia. Jest to zagadnienie niezwykle istotne dla każdego, kto pragnie w sposób kontrolowany tworzyć, w ramach budowanego przez siebie świata, wrażenie rzeczywistości. Obserwacja sposobu rozkładu światła i cienia, analiza zależności mających miejsce między nimi i ich znaczenia dla budowy ogólnego nastroju od wieków było istotnym elementem procesu tworzenia sztuki. Tak malarze, jak i współcześnie fotograficy, dużą wagę przykładają do oświetlenia, z którym przyszło im pracować. Często też sami świadomie kształtują je.

Model oświetlenia przyjęty w OpenGL ze względu na nastawienie na wydajność musiał być stosunkowo prosty i mimo, że wywiedziony z fizyki to daleki jest od wiernego symulowania rozkładu światła w świecie rzeczywistym. Wprowadzone uproszczenia i ograniczenia nie powinny jednak tyczyć się wyobraźni osób korzystających z biblioteki. Przy odrobinie pomysłowości i korzystając z dodatkowych rozwiązań możliwe jest uzyskanie zaskakująco dobrych rezultatów.

Jeżeli jednak z jakichś powodów istnieje konieczność symulacji rozkładów świateł w sposób wierny rzeczywistości, pozostaje skorzystać z systemów opartych na technikach takiej jak *radiosity*, a wyniki wykorzystać przy wyświetlaniu scen OpenGL. To jednak wykracza poza zakres zagadnień, którymi się tu zajmujemy. Zainteresowanym polecam zapoznanie się z RadiosGL, pod adresem `http://hcsoftware.sourceforge.net/RadiosGL/RadiosGL.html`, generatorem modelu oświetlenia off-line i przeglądarką stworzoną w OpenGL.

8.1. Światło w OpenGL

Na kolor i jasność każdego fragmentu wyświetlonego modelu składa się kilka czynników. Najważniejszymi z nich jest padające światło, cechy materiału i tekstura, którą został pokryty. Obecnie skupimy się na dwóch pierwszych, z wymienionych, aspektach.

OpenGL domyślnie obsługę świateł dla sceny ma wyłączoną. w tej sytuacji o kolorze poszczególnych powierzchni decydują wartości kolorów ustawione dla poszczególnych wierzchołków. Uruchomienie obliczeń dla oświetlenia i generacji gotowego obrazu z uwzględnieniem ich dokonujemy podając do `glEnable()` stałą zdefiniowana jako `GL_LIGHTING`. Następnie włączamy, w sposób analogiczny, przynajmniej jedno źródło światła podając parametr `GL_LIGHT0`.

W bibliotece domyślnie jest dostępnych tylko osiem niezależnych źródeł światła. Nazwane zostały jako `GL_LIGHT0` do `GL_LIGHT7`. Światło zerowe, jako jedyne, domyślnie po aktywowaniu jest włączone z ustawionym kolorem świecenia tak dla rozproszonego jak i rozbłyskowego na biały i maksymalną intensywnością.

Ustawiając parametry każdego ze świateł, mamy możliwość podjęcia decyzji o jego charakterystyce. Czy ma być światłem dookolnym, czy kierunkowym reflektorem, jaki ma być spadek jego intensywności wraz z odległością. Mamy też możliwość wpływu do koloru świecenia.

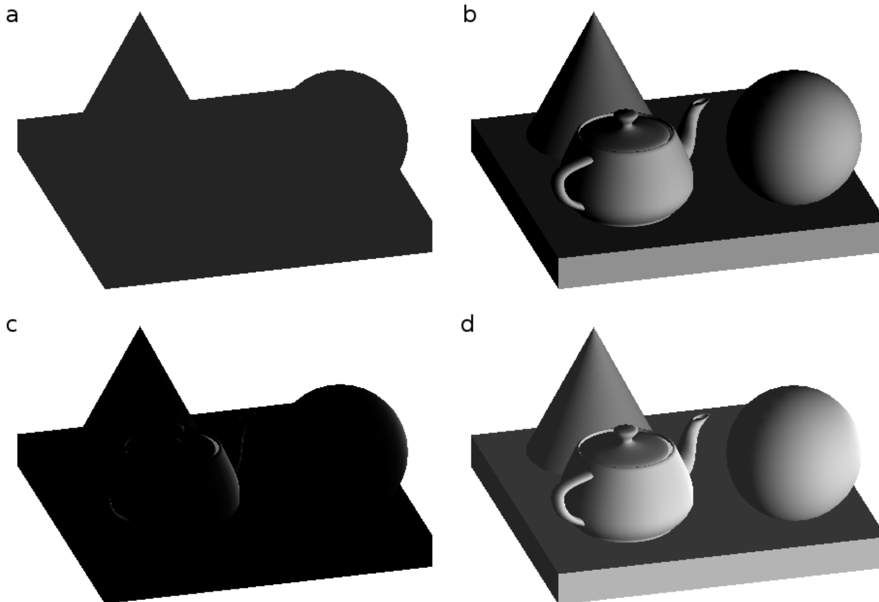
W czasie ustalania sposobu oświetlenia naszej sceny istotne jest też wybranie, jakie zachowanie modelu zostanie skonfigurowane. Dokonujemy tego, korzystając z funkcji `glLightModel()` z odpowiednimi parametrami. Domyślnie OpenGL korzysta z włączonego oświetlenia otoczenia o kolorze i intensywności opisanym wektorem w przestrzeni RGBA o wartości (0.2, 0.2, 0.2, 1.0).

Jeżeli zależy nam na poprawieniu wyświetlania rozbłysków, ale licząc się ze znacznym spadkiem wydajności, możliwe jest włączenie wyliczania ich względem pozycji obserwatora, a nie punktu wirtualnie nieskończenie odległego. w tym celu należy jako parametry wywołania, wspomnianej funkcji

konfiguracyjnej, podać stałą GL_LIGHT_MODEL_LOCAL_VIEWER wskazującą na zmieniane zachowanie i GL_TRUE. Warto tutaj wspomnieć, że rozwiązanie to jest kłopotliwe z jeszcze jednego powodu. Przy zmianie położenia światła konieczne jest wywołanie ponownego przeliczenia przekształceń projekcji, mimo, że pozycja obserwatora nie uległa zmianie.

Rozbłyski mogą być też traktowane w sposób specjalny jeżeli pracujemy z teksturami. Domyślnie OpenGL wylicza jasność poszczególnych fragmentów powierzchni biorąc pod uwagę zarówno składową pochodzącą od rozbłysków jak i światła otoczenia i rozproszenia, a następnie nakłada teksturę. Możemy zmienić ten bieg rzeczy i zmusić do nałożenia rozbłysków już po etapie teksturowania, co podnosi jakość generowanego oświetlenia, ale jak można się spodziewać wpływa na wydajność. Osiągnąć to można wywołując glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR).

W specyficznych sytuacjach możliwe jest też włączenie oświetlania tylnej strony wielokątów budujących model poprzez przekazanie jako argumentu zdefiniowanych stałych GL_LIGHT_MODEL_TWO_SIDE i GL_TRUE.



Rysunek 8.2. Poszczególne składniki, otoczenia (a), rozproszenia (b) i rozbłysku (c) dla modelu światła oraz wynik ich złożenia (d).

8.1.1. Światło otoczenia

Światło otoczenia (*ambient*) jest syntetycznie wprowadzonym do modelu tworem, który ma za zadanie symulować wkład w jasność poszczególnych obiektów wynikającą tak z odbicia światła od innych obiektów znajdujących się na scenie, jak i rozproszenie na cząstkach atmosfery. Nie pada ono pod żadnym określonym kątem i obiekt z każdej strony zostaje oświetlony w jednakowym stopniu.

Tabela 8.1. Wartości dla drugiego argumentu dla `glLight()`

Nazwa stałej	Działanie
`GL_AMBIENT`	intensywność i kolor światła otoczenia
`GL_CONSTANT_ATTENUATION`	składowa stała spadku mocy świecenia
`GL_DIFFUSE`	intensywność i kolor światła rozproszonego
`GL_LINEAR_ATTENUATION`	składowa liniowa spadku mocy świecenia
`GL_POSITION`	współrzędne pozycji światła
`GL_QUADRATIC_ATTENUATION`	składowa wykładnicza spadku mocy świecenia
`GL_SPECULAR`	intensywność i kolor światła rozbłysku
`GL_SPOT_CUTOFF`	kąt światła reflektorowego
`GL_SPOT_DIRECTION`	kierunek światła reflektorowego
`GL_SPOT_EXPONENT`	rozkład intensywności światła reflektorowego

8.1.2. Światło rozproszenia

Pada kierunkowo ze źródła światła i zostaje rozproszone w różnych kierunkach i niezależnie od kierunku, z którego patrzymy jasność danego miejsca na powierzchni obiektu jest taka sama. Ilość światła rozproszonego jest zależna od jego kąta padania względem normalnej.

8.1.3. Światło rozbłysku

Modelując każde ze świateł, podajemy jego pozycję i wkład w oświetlenie sceny korzystając z funkcji `glLight()`. Przyjmuje ona trzy argumenty

w swoim wywołaniu. Pierwszym jest to numer źródła światła, którego tyczy się ustawiany parametr, drugi wskazuje na stałą reprezentującą jego nazwę, a trzeci odpowiada za ustawiane wartości liczbowe. Lista wszystkich dostępnych parametrów zawarta została w tabeli 8.1. Przykładowe ich użycie przedstawia listing 8.1. Jak widać w kodzie położenie i wartości koloru-intensywności zostały zamknięte w postać wektorów.

Listing 8.1. Opis pojedynczego źródła światła i włączenie jego obsługi.

```
1 GLfloat light_position[] = { 1.0, 1.0, -10.0, 0.0 };
  GLfloat light_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
3 GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat light_specular[] = { 0.8, 0.8, 0.8, 1.0 };
5
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
7 glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
9 glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

11 glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);

13 glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
```

Rysunek 8.3 przedstawia wpływ na końcowe oświetlenie sceny poszczególnych składowych rodzajów światła.

8.2. Modyfikacje świateł

Jak już było wspomniane, model światła w OpenGL jest skonstruowany pod kątem maksymalnej wydajności. Światła powinny być włączane z rozwagą, w ilości minimalnej. Tak samo opisane modyfikacje, które możliwe są do wprowadzenia dla poszczególnych lamp należy używać tylko w sytuacji, gdy jest to uzasadnione. Istnieje szereg metod, w tym mapy oświetlenia, będące odpowiednio spreparowaną teksturą, które przyjdą z pomocą.

8.2.1. Kontrola spadku intensywności świecenia

Za pomocą parametrów nazywających się `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION` i `GL_QUADRATIC_ATTENUATION` możemy wpływać na spadek intensywności świecenia w zależności od odległości pomiędzy źródłem światła, a obiektem. Przy domyślnych ustawieniach zmiana położenia modelu nie wpływa na jego iluminację, możliwa jest jednak zmiana domyślnego zachowania poprzez wprowadzenie do sposobu obliczania przez

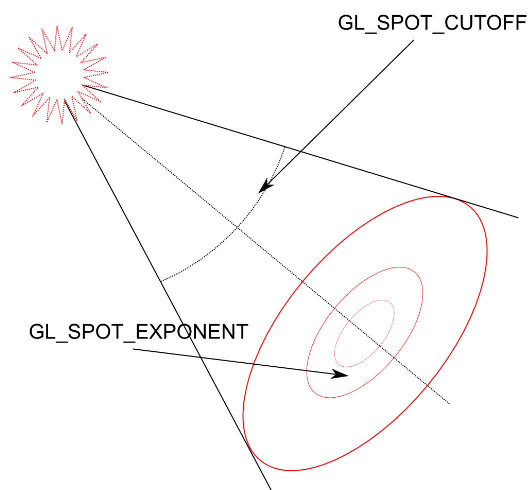
OpenGL aktualnej intensywności nietypowych wartości współczynników. Dobór ich wynika ze wzoru:

$$I_l = \frac{1}{a_c + a_l d + a_q{}^2 d}$$

gdzie a_c, a_l i a_q to wartości wymienionych wcześniej parametrów.

8.2.2. Reflektor

Typowe źródło światła w OpenGL ma dookolną charakterystykę świecenia. Czasami przydałoby się mieć jednak możliwość użycia źródła kierunkowego. Jest to możliwe, a na dokładkę istnieje szereg dodatkowych parametrów umożliwiających wpływanie na generowany wirtualny strumień świetlny.



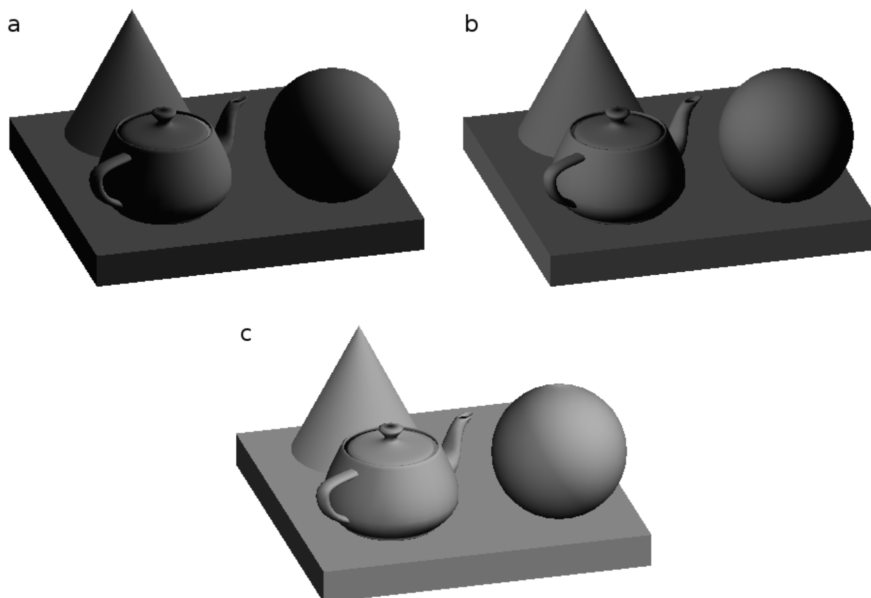
Rysunek 8.3. Parametry wpływające na konfigurację reflektora.

Wprowadzenie światła w tryb reflektora oznacza konieczność ustawienia `GL_SPOT_CUTOFF` czyli parametru opisującego szerokość kątową stożka świecenia względem osi padającego światła. Następnie trzeba wskazać kierunek w jakim chcemy skierować reflektor. Tutaj konieczne jest trochę matematyki zarówno gdy oświetlamy określony punkt sceny, jak również jeżeli ma on podążać za określonym obiektem. Odpowiada za to `GL_SPOT_DIRECTION`.

Ostatnim krokiem może być zmiana sposobu rozkładu światła na oświetlanej powierzchni. Domyślnie jest to dystrybucja równomierna, ale zmieniając wartość dla parametru GL_SPOT_EXPONENT możemy skupić większość światła bliżej osi stożka.

Tu warto pamiętać, że przy scenach składających się z obiektów o małej gęstości sieci trójkątów uzyskane za pomocą reflektora efekty będą rozczarowujące.

8.2.3. Wiele źródeł światła



Rysunek 8.4. Wynik oświetlenia światłem tylko z prawej strony sceny (*a*), tylko z lewej (*b*) i jednoczesny z obu (*c*).

8.3. Materiał

Przejdziemy teraz do drugiej części zagadnień związanych z podstawowym modelowaniem świateł w ramach OpenGL. Jeżeli przyjrzymy się formule matematycznej, którą używa biblioteka do ustalenia wartości oświetlenia, okaże się, że na wynik wpływ ma tak położenie i parametry lamp jak i materiał, z którego wygenerowany jest obiekt. Przyjmując następujące oznaczenia dla globalnego oświetlenia otoczenia zdefiniowanego dla modelu

m_a. Współczynników oświetlenia otoczenia a_i, rozproszonego d_i i rozbłysku s_i dla kolejnych, indeksowanych po i, zródeł światła. Współczynników materiałowych obiektu dla światła otoczenia m_a, rozproszonego m_d i dwóch od rozbłysku m_s i m_{sh}. Wektorów normalnego **n**, skierowanego od wierzchołka do źródła światła $\mathbf{L_i}$ i rozbłysku **s**, który liczony jest różnie w zależności od tego czy używana jest pozycja obserwatora, czy nie. Kolor wierzchołka wyraża się wzorem:

$$V_c = M_a + \sum_{i=0}^{n-1} I_{l_i} \left[a_i m_a + (max\{\mathbf{L_i} \cdot \mathbf{n}, 0\}) d_i m_d + (max\{\mathbf{s} \cdot \mathbf{n}, 0)\}^{m_s h} s_i m_s \right]$$

gdzie

$$M_a = m_e + m_a * a_l$$

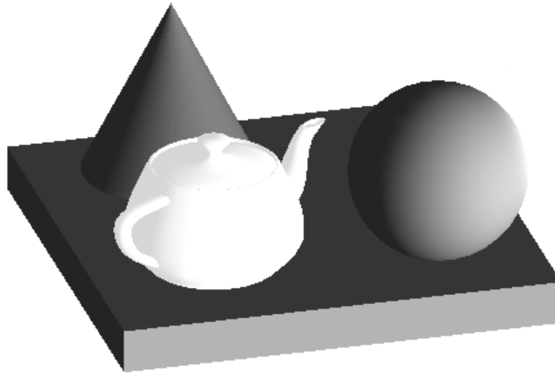
Ustawienie wartości współczynników parametrów charakteryzujących powierzchnię obiektów odbywa się z użyciem funkcji `glMaterial()`. w jej wywołaniu przekazywane są trzy argumenty. Pierwszy z nich wskazuje na stronę wielokąta, określonej przez kierunek normalnej, której tyczyć się będzie parametr. Tu możliwe są trzy wartości zdefiniowane jako stałe `GL_FRONT` dla przodu, `GL_BACK` dla tyłu i `GL_FRONT_AND_BACK` dla obu stron. Drugi argument to nazwa parametru zbiorczo przedstawiona w tabeli 8.2. Ostatnim zaś jest wartość parametru, która może być wektorem lub zmienną liczbową.

8.3.1. Emisyjność obiektów

Mimo że uzyskiwany efekt kojarzy się, ze źródłem światła, emisyjność jest cechą materiałową w OpenGL. Fakt, że obiekt na scenie wygląda po użyciu tego parametru na rozświetlony od wewnątrz, to nie generuje światła wpływającego na wygląd sceny. Widoczne jest to na rysunku 8.5, gdzie kula będąca w bezpośrednim sąsiedztwie czajniczka zwrócona jest do niego ciemniejszą stroną.

8.3.2. Rozbłysk na powierzchni obiektów

W przypadku definicji materiału dwa parametry są odpowiedzialne za opis światła rozbłysku dla powierzchni. Pierwszy z nich, analog parametru omawianego przy okazji światła, odpowiada za intensywność i kolor lśnienia. Drugi z parametrów odpowiedzialny jest za wymodelowanie połysku powierzchni. Jeżeli jego wartość jest mała, to uzyskujemy mocne rozbłyski przypominające te znane z powierzchni polerowanego metalu, czy kamienia, wraz ze wzrostem jego powierzchnie sprawiają wrażenie bardziej matowych.



Rysunek 8.5. Obiekt z ustawioną emisyjnością.

Należy jednak pamiętać, że lustrzany połysk nie jest w żaden sposób odpowiedzialny za powstawanie odbić innych elementów sceny od takich powierzchni, taki efekt możliwy jest do uzyskania tylko za pomocą mapowania środowiskowego.

8.3.3. Pozostałe parametry

Opisując powierzchnię obiektu, używane są jeszcze parametry odpowiedzialne za sposób zachowania na padające światło otoczenia, zarówno zdefiniowane w modelu, jak i pochodzące od aktywnych lamp oraz światła rozproszenia.

Pozostałe atrybuty materiałowe, które pojawiają się w tabeli 8.2, możemy bez szkody dla spójności wywodu, przemilczeć.

8.3.4. Alternatywna zmiana parametrów materiału

Istnieje jeszcze jedna, dość specyficzna metoda zmiany parametrów opisujących materiał dla obiektu. Jeżeli na maszynie stanów włączymy obsługę opcji `GL_COLOR_MATERIAL` za pomocą `glEnable()`, to korzystając z funkcji `glColorMaterial()` z odpowiednimi wywołanej z dwoma argumentami, gdzie pierwszy wskazuje czy dane ustawienia tyczą się, analogicznie jak w `glMaterial()`, przodu, tyłu czy obu stron wielokąta oraz drugiej wybierającej obsługę jednego z parametrów materiału, pozwalamy na zmianę współczynników wskazanego parametru przy użyciu funkcji `glColor()`.

Tabela 8.2. Wartości dla drugiego argumentu dla `glMaterial()`

Nazwa stałej	Działanie
`GL_AMBIENT`	współczynnik intensywności i koloru materiału dla padającego światła otoczenia
`GL_AMBIENT_AND_DIFFUSE`	współczynnik intensywności i koloru materiału jednakowy dla padającego światła otoczenia i rozproszonego
`GL_DIFFUSE`	współczynnik intensywności i kolor materiału dla padającego światła rozproszonego
`GL_EMISSION`	emisyjność materiału
`GL_SHINESS`	współczynnik ostrości rozkładu powierzchniowego światła rozbłysku dla materiału
`GL_SPECULAR`	współczynnik intensywności i koloru materiału dla padającego światła rozbłysku
`GL_COLOR_INDEXES`	indeksy do pozycji w palecie dla światła otoczenia, rozproszonego i rozbłysku

Rozdział 9

Teksturowanie

9.1. Biblioteka GLPNG **118**
9.2. Obsługa teksturowania **119**
9.2.1. Utworzenie identyfikatora tekstury 120
9.2.2. Określenie parametrów tekstury 122
9.2.3. Definicja tekstury 125
9.2.4. Skojarzenie tekstury z obiektem 126
9.3. Mipmapy **127**
9.4. Tekstury trójwymiarowe **127**

W tym rozdziale zajmiemy się niezwykle ważnym elementem każdej atrakcyjnej pod względem wizualnym sceny. Mowa będzie o teksturach, których znaczenie we współczesnej grafice jest trudne do przecenienia. Dzięki nim możliwe jest nie tylko odwzorowanie nawet niezwykle skomplikowanych układów barwnych na powierzchni obiektów, ale również ich faktury, odbić lustrzanych czy skomplikowanej gry świateł. Tematyka jest na tyle szeroka, że uda nam się zapoznać tylko z najważniejszymi jej aspektami.

9.1. Biblioteka GLPNG

Zanim zaczniemy zabawy z teksturami, musimy zapoznać się z narzędzie, które umożliwi bezproblemowe wczytanie nam odpowiednich bitmap. Sama biblioteka OpenGL nie posiada żadnych mechanizmów umożliwiających wczytywanie danych z zasobów plikowych. To jest zadanie zrzucone w całości na barki programisty. w przykładach krążących w Internecie obrazy tekstur z reguły zapisane są w formacie BMP. Nie widzę jednak powodów, żeby kurczowo trzymać się tego rozwiązania, gdy do dyspozycji mamy dużo lepsze rozwiązanie. Jest nim PNG - otwarty format i standard zapisu obrazów kompresowanych bezstratnie wraz z informacją o kanale przezroczystości. Do naszych zastosowań jest to rozwiązanie idealne.

Skorzystamy w tym celu z biblioteki GLPNG, która jest dostępna pod adresem `http://www.fifi.org/doc/libglpng-dev/glpng.html` oraz w repozytoriach wielu systemów operacyjnych z rodziny GNU.

Składa się na nią dosłownie dziesięć funkcji umożliwiających tak wczytanie obrazu, jak i stworzenie w oparciu o niego *mipmap*, a nawet jednoczesne dobindowanie bitmapy w ramach maszyny stanów OpenGL. Co więcej, zadba o to, aby rozmiar tekstury nie przekraczał nałożonych na nią ograniczeń dokonując odpowiedniego jej przeskalowania.

Na początek interesuje nas tylko funkcja `pngLoad()`, w dalszej części pojawi się też `pngBind()`. Przyjmuje ona cztery argumenty: nazwę pliku, dwie stałe, gdzie pierwsza określa sposób generacji *mipmap*, druga rodzaj obsługi przezroczystości. Ostatnim z listy argumentów jest wskaźnik do struktury zawierającej informacje o pliku (patrz tabela A.1). Funkcja zwraca wartość różną od zera jeżeli otwarcie pliku powiodło się.

Przykład użycia zaprezentowanej funkcji zawarty został w listingu 9.1. Obecnie część operacji, które towarzyszą zamiany wczytanej bitmapy na teksturę pozostaną chwilowo nie czytelne, ale musicie zaufać, że w ten sposób się to odbywa.

Listing 9.1. Wczytanie tekstury z użyciem biblioteki GLPNG.

```
GLuint tex1;
```

```
2
  int LoadTextures(void) {
4     glGenTextures(1, &tex1);
      glBindTexture(GL_TEXTURE_2D, tex);
6
      glTexParameteri(GL_TEXTURE_2D,
8         GL_TEXTURE_WRAP_S, GL_REPEAT);
      glTexParameteri(GL_TEXTURE_2D,
10        GL_TEXTURE_WRAP_T, GL_REPEAT);
      glTexParameteri(GL_TEXTURE_2D,
12        GL_TEXTURE_MAG_FILTER, GL_LINEAR);
      glTexParameteri(GL_TEXTURE_2D,
14        GL_TEXTURE_MIN_FILTER, GL_LINEAR);

16    if (!pngLoad("tex1.png",
          PNG_NOMIPMAP, PNG_SOLID, NULL))
18            return 1;
  }
```

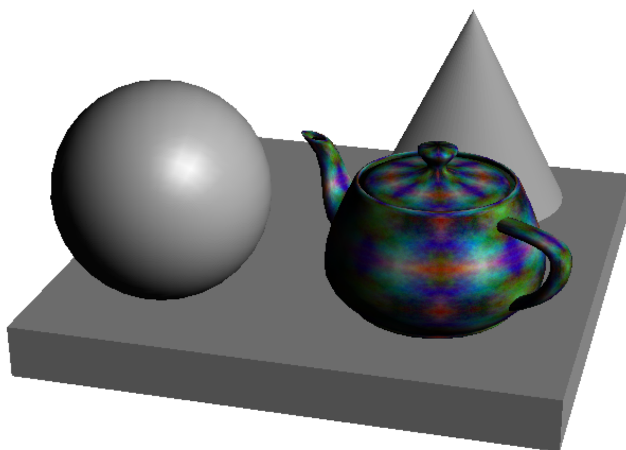
Nałożenie tekstury na imbryczek zaprezentowane zostało zaś w listingu 9.2, a wynik przedstawiony jest na rysunku 9.1.

Listing 9.2. Użycie tekstury.

```
1 glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D, id);
3
  glPushMatrix();
5 glScalef(0.2,0.2,0.2);
  glRotatef(-45,0,1,0);
7 glTranslatef(-1.5,-1.25,-0.4);
  glutSolidTeapot(1.0);
9 glPopMatrix();

11 glDisable(GL_TEXTURE_2D);
```

9.2. Obsługa teksturowania

OpenGL daje możliwość tworzenia tekstur zarówno jedno, dwu jak i trójwymiarowych. My zajmiemy się w naszych rozważaniach tylko tym drugim przypadkiem. Dodatkowo, ponieważ w bibliotece zostały zdefiniowane kilka jednostek teksturowania możliwe jest nałożenie na każdy z fragmentów do wielu tekstur jednocześnie. Ile, możemy sprawdzić, korzystając z funkcji `glGetIntegerv()` z argumentem pierwszym w postaci stałej `GL_MAX_TEXTURE_UNITS` i drugim w postaci wskaźnika do zmiennej przechowania wyników.



Rysunek 9.1. Imbryk pokryty teksturą.

Tekstury wykorzystywane w OpenGL nie we wszystkich implementacjach mogą mieć rozmiary dowolne. Bezpieczniej jest przyjąć założenie, że zarówno wysokość jak i szerokość jest wyrażona wzorem:

$$S = 2^n + 2b$$

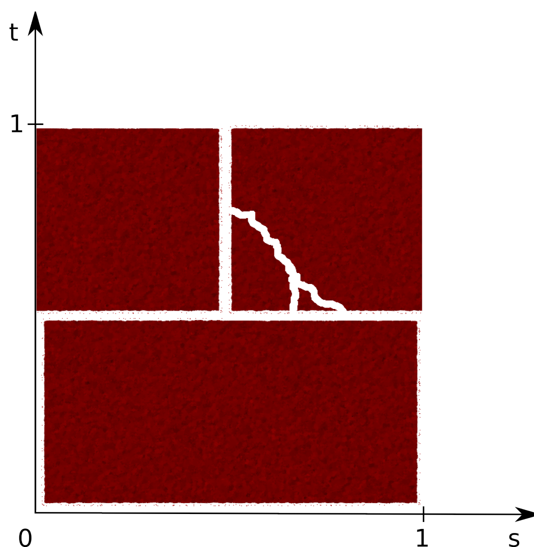
gdzie n jest liczbą całkowitą, a b to szerokość brzegu, która maksymalnie może mieć rozmiar jednego piksela. Dodatkowo warto zwrócić uwagę, że maksymalny wymiar tekstury dla każdej implementacji nie powinien być mniejszy niż 64 piksele. Tą wartość możemy sprawdzić, analogicznie jak miało to miejsce dla maksymalnej liczy jednostek teksturowana, za pomocą `glGetIntegerv()`, tym razem przekazując do funkcji stałą `GL_MAX_TEXTURE_SIZE`.

Tekstura ma przypisany własny, lokalny układ współrzędnych z użyciem współrzędnych s, t i r, które odpowiadają współrzędnym w układzie kartezjańskim, ale wszystkie możliwe ich wartości są zawarte w przedziale od 0.0 do 1.0.

9.2.1. Utworzenie identyfikatora tekstury

W przypadku teksturowania przyjęło się operować dla pojedynczego elementu tekstury terminem teksel. w najprostszym przypadku może on mieć bezpośrednie przełożenie na poszczególne bitmapy tekstury bazowej, ale przy bardziej zaawansowanych technikach jego znaczenie jest szersze.

W procesie nanoszenia tekstury na obiekt mamy zawsze możliwość wpływania w jaki sposób wyglądać będzie wpływ koloru poszczególnych tekseli na wynikowa wartość koloru poszczególnych fragmentów.



Rysunek 9.2. Dwuwymiarowa tekstura w układzie współrzędnych *st*.

Warto też pamiętać przed przystąpieniem do teksturowania, aby w ramach maszyny stanów OpenGL został ustawiona preferencja co do najlepszego stopnia wyliczania korekcji perspektywy za pomocą funkcji `glHint()` wywołanej z wartością dla parametru `GL_PERSPECTIVE_CORRECTION_HINT` ustawioną na `GL_NICEST`.

Pierwszym krokiem w tworzeniu tekstury na potrzeby obiektów sceny jest zaalokowanie identyfikatorów jednoznacznie opisujących je w systemie. Wykorzystywana w tym celu jest funkcja `glGenTextures()`, w wywołaniu której podajemy dwa argumenty, liczbę alokowanych nazw i wskaźnik do tablicy, w której będą przechowywane. Mimo że w większości implementacji będą to kolejne liczby, nie należy zakładać, że zawsze tak jest.

Kolejnym krokiem jest utworzenie tekstury określonego typu i skojarzenie jej z nazwą. Funkcja `glBindTexture()`, która jest za to odpowiedzialna w sytuacji, gdy wykryje, że skojarzenie z podaną nazwą danego typu ma już miejsce ustawia teksturę jako bieżącą dla maszyny stanów. Przyjmuje ona dwa argumenty, gdzie pierwszy to typ i może on przyjmować jedną z czterech wartości wyrażonych w postaci stałych: `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D` dla tekstur jedno, dwu i trójwymiarowych i `GL_TEXTURE_CUBE_MAP` dla map sześciennych. Te ostatnie dostępne są dopiero od specyfikacji w wersji 1.3.

9.2.2. Określenie parametrów tekstury

Kolejnym krokiem jest ustawienie dla skojarzonej aktualnie tekstury parametrów jej obsługi. Dokonywane jest z użyciem trójargumentowej funkcji `glTexParameter()`. w jej wywołaniu podajemy typ tekstury taki jak miało to miejsce w `glBindTexture()`, rodzaj ustawianego parametru i odpowiadająca mu wartość. w tym trzecim przypadku może to być zdefiniowana stała, lub wartość liczbowa.

Istnieje bardzo małe prawdopodobieństwo, że w typowej scenie przy jej generacji będziemy mieli sytuację, gdzie pojedynczy piksel ekranu pokrywa się rozmiarem z tekselem. w większości obiekty widzimy z pewnej odległości i konieczne jest dokonania redukcji rozmiaru tekstury. w jaki sposób ma to być dokonane mówi nam wartość przekazywana dla parametru `GL_TEXTURE_MIN_FILTER`. Tabela 9.1 zawiera zestawienie możliwych sposobów znajdowania wartości koloru tekstury dla danego fragmentu. Najszybsze rozwiązanie `GL_NEAREST` korzysta tylko z najbliższego, w metryce Manhattan, teksela, najwolniejsze `GL_LINEAR_MIPMAP_LINEAR`, ale również najlepsze jakościowo opiera się na mipmapach i interpolacji liniowej. Konieczne jest też zadbanie o odpowiednie zachowanie się środowiska jeżeli do obiektu zbliżymy się nadmiernie i poszczególne teksele będą mapowane na obszary większe niż pojedynczy piksel. Preferowane zachowanie definiuje nam w tym wypadku `GL_TEXTURE_MAG_FILTER`. Przy powiększaniu nie ma do dyspozycji metod opierających się na mipmapach, z tej prostszej przyczyny, że działamy powyżej rozdzielczości tekstury bazowej, która jest największa z nich.

Kolejnym istotnym parametrem dla tekstury jest zachowanie w sytuacji gdy definiujemy mapowanie tesktury na współrzędne lokalne obiektu. Operacja ta jest wyjątkowo istotna dla poprawnej generacji sceny. Korzystamy w tym celu z funkcji `glTexCoord()`, podając w niej współrzędne teksturowe przed wystąpieniem kolejnego wierzchołka obiektu. w przypadku przedstawionym w listingu 9.3 mamy bezpośrednie przełożenie kwadratowej tekstury na kwadratowy obiekt. Widać, że poszczególne wierzchołki są ze sobą skorelowane. Wynik działania tego kodu przedstawia rysunek 9.3.

Listing 9.3. Użycie tekstury.

```
1  glEnable(GL_TEXTURE_2D);
   glBindTexture(GL_TEXTURE_2D, id);
3
   glBegin(GL_QUADS);
5
   glNormal3f(0,0,-1);
7
   glTexCoord2f(0.0, 0.0);
9  glVertex3f(-1.0, -1.0, 0.0);
   glTexCoord2f(1.0, 0.0);
```

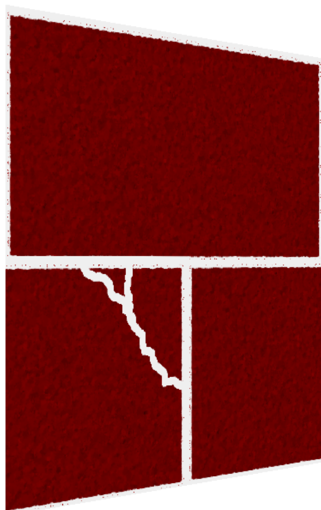
Tabela 9.1. Możliwe wartości dla funkcji `glTexParameter()` i parametrów `GL_TEXTURE_MIN_FILTER` oraz `GL_TEXTURE_MAG_FILTER` (tylko szare).

Wartość	Opis
`GL_NEAREST`	Skopiuj wartość teksela najbliższego sąsiada.
`GL_LINEAR`	Wylicz wartość średnią z tekseli czterech najbliższych sąsiadów.
`GL_NEAREST_MIPMAP_NEAREST`	Wybierz mipmapę o najbliższym rozmiarze i skopiuj wartość teksela najbliższego sąsiada.
`GL_LINEAR_MIPMAP_NEAREST`	Wybierz mipmapę o najbliższym rozmiarze i wylicz wartość średnią z tekseli czterech najbliższych sąsiadów.
`GL_NEAREST_MIPMAP_LINEAR`	Wybierz dwie mipmapy o najbliższym rozmiarze i oblicz średnią wartość teksela z najbliższych sąsiadów każdej z nich.
`GL_LINEAR_MIPMAP_LINEAR`	Wybierz dwie mipmapy o najbliższym rozmiarze i oblicz średnią wartość teksela z najbliższych czterech sąsiadów każdej z nich.

```
11 glVertex3f(1.0, -1.0, 0.0);
   glTexCoord2f(1.0, 1.0);
13 glVertex3f(1.0, 1.0, 0.0);
   glTexCoord2f(0.0, 1.0);
15 glVertex3f(-1.0, 1.0, 0.0);

17 glEnd();

19 glDisable(GL_TEXTURE_2D);
```

Co się jednak stanie, jeżeli definiując współrzędne mapowania tekstury podamy wartości inne niż z zakresu od 0.0 do 1.0? Okazuje się, że zachowanie które będzie miało miejsce zależne jest od tego jaki tryb przypiszemy parametrom zawijania tekstur `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T`, `GL_TEXTURE_WRAP_R` z użyciem funkcji `glTexParameter()`. Wszystkie możliwe wartości zawiera tabela 9.2.

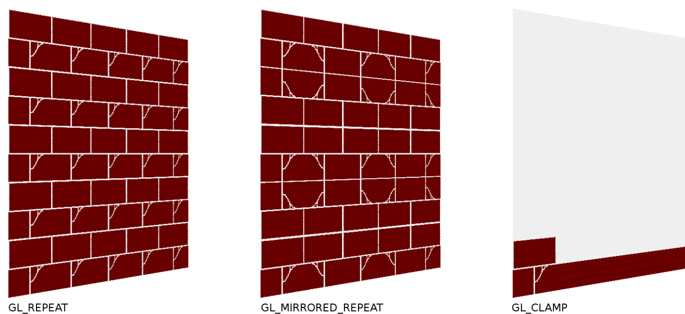
Wynik użycia różnych wartości parametru `GL_TEXTURE_WRAP` przedstawia rysunek 9.4.



Rysunek 9.3. Wynik teksturowania.

Wśród parametrów, które możemy ustawić w odniesieniu do tekstury pojawia się jeszcze możliwość zdefiniowania koloru brzegu. Dokonujemy tego analogicznie tym razem używając parametru `GL_TEXTURE_BORDER_COLOR`, jako jego wartość podając tablicę ze zdefiniowanymi składowymi koloru.

Wszystkie pozostałe zmiany, które można dokonać w stosunku do sposobu przetwarzania tekstur w OpenGL, wykraczają poza zakres tej książki i z tego powodu nie zostaną wspomniane.



Rysunek 9.4. Wynik teksturowania z użyciem różnych typów zawijania.

Tabela 9.2. Możliwe sposoby zawijania tekstury.

Wartość	Opis
GL_CLAMP	Jeżeli jest zdefiniowany brzeg tekstury zostaje on powielony na całym obszarze w danym kierunku w przeciwnym razie duplikowane są brzegowe teksele.
GL_CLAMP_TO_EDGE	Brzeg tekstury nawet jak jest zdefiniowany zostaje zignorowany. Powielone zostają teksele.
GL_CLAMP_TO_BORDER	Kolor zdefiniowanego brzegu zostaje powielony.
GL_REPEAT	Powtarza w danym kierunku wzór tekstury. Jeżeli tekstura ma zdefiniowany brzeg to jest on ignorowany.
GL_MIRRORED_REPEAT	Powtarza w danym kierunku wzór dokonując jednoczesnego jego odbicia osiowego. Dla tekstury ze zdefiniowanym brzegiem jest on ignorowany.

9.2.3. Definicja tekstury

Jak już zostało wspomniane, sam OpenGL nie ma możliwości wczytywania danych z plików. Dlatego sama definicja tekstury zawierająca wartości kolejnych tekseli do maszyny stanów przekazywana jest bezpośrednio z pamięci poprzez wskaźnik. w zależności od ilości wymiarów dla danej tekstury korzysta się w tym celu z jednej z trzech funkcji `glTexImage1D()`, `glTexImage2D()` i `glTexImage3D()`.

Listing 9.4. Definicja tekstury w oparciu o tablicę image.

```
1  GLubate image[64][64][4];
   GLuint tex1;
3
   funkcja_odczytu_obrazu_z_pliku(&image);
5
   glGenTextures(1, &tex1);
7  glBindTexture(GL_TEXTURE_2D, id);

9  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                                               GL_LINEAR);
11 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                                               GL_LINEAR);
13
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 64, 64, 0,
15     GL_RGBA, GL_UNSIGNED_BYTE, image);
```

Tabela 9.3. Kolejne argumenty funkcji `glTexImage*D()` i ich możliwe wartości.

Argument	Opis
`target`	Definiuje typ tekstury.
`level`	Dla 0 obrazek bazowy, kolejne wartości to n-ta generacja mipmapy
`internalFormat`	Określa wewnętrzny sposób reprezentacji składowych koloru dla tekstury.
`width`	Szerokość tekstury
`height`	Wysokość tekstury. Tylko dla dwu i trójwymiarowych.
`depth`	Głębokość tekstury. Tylko dla trójwymiarowych.
`border`	Szerokość brzegu. Możliwe tylko 0 i 1.
`width`	Szerokość tekstury
`format`	Określa reprezentację składowych koloru w przekazywanych danych.
`data`	Wskaźnik do obszaru pamięci zawierającego teksturę.

Sposób reprezentacji wewnętrznej poszczególnych tekseli zdefiniowanej tekstury może być inny niż ich format w danych wejściowych. OpenGL w takiej sytuacji dokona wewnętrznej konwersji.

9.2.4. Skojarzenie tekstury z obiektem

Przed skojarzeniem tekstury z konkretnym obiektem konieczne jest wykonanie kilku kroków bezpośrednio je poprzedzających. Pierwszym jest uruchomienie na maszynie stanów obsługi teksturowania za pomocą `glEnable()`. w zależności od wymiarowości podaje się w wywołaniu wartość argumentu w postaci stałej takiej samej jak miało to miejsce w procesie kojarzenia z nazwą. Ponieważ same tekstury zostają wczytane w standardowym przypadku poza główną pętlą renderowania sceny OpenGL, konieczne jest ponowane wskazanie, która z posiadanych w systemie tekstur będzie bieżącą dla maszyny stanów. Jak już wcześniej wspomnieliśmy, odpowiedzialna za to jest funckja `glBindTexture()`. w tym momencie może pojawić się opis geometryczny obiektu rozszerzony o skojarzone z konkretnymi wierzchołkami współrzędne tekstury. Takie rozwiązanie pozwala na poprawne nanoszenie tekseli nawet na bardzo złożone obiekty. Całość zamyka wyłączenie obsługi

teksturowania, skorzystanie z bieżącej tekstury dla kolejnych modeli, lub podmiana aktualnie używanej.

9.3. Mipmapy

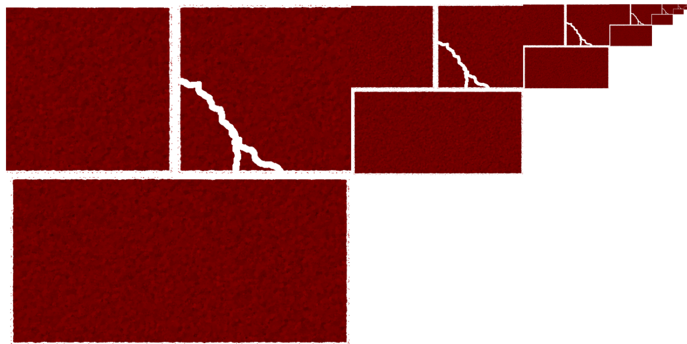
W sytuacji, gdy obiekty znajdujące się na scenie są oddalone od obserwatora nie ma żadnych specjalnych przesłanek, żeby pokrywać je teksturami w pełnej możliwej rozdzielczości ponieważ jest to czasochłonny proces, który w zaistniałej sytuacji nie podnosi w widzialny sposób jakości generowanego obrazu. Dużo lepszym rozwiązaniem jest skorzystanie z tekstur o odpowiednio zmniejszonej rozdzielczości wybieranych w zależności od stosunku rozmiaru piksela ekranowego i teksela. Technika ta, określana mianem mimpappingu, pozwala na zwiększenie złożoności i ilości używanych tekstur bez niekorzystnego wpływu na ogólną wydajność wizualizacji.

W mipmappingu kolejne warianty tekstur generowane są przez zmniejszenie rozdzielczości tekstury bazowej z dzielnikiem będącym kolejnymi potęgami liczby dwa. Może ten proces zostać przygotowany w oprogramowaniu zewnętrznym i zdefiniowany jako kolejny poziom dla bieżącej tekstury z użyciem `glTexImage*D()`, lub automatycznie przeprowadzony przez OpenGL.

Automatyczne generowanie mipmap jest wykonywane dla bieżącej tekstury w momencie gdy wywołana zostanie funkcja `glGenerateMipmap()`. Jako jej argument podajemy standardowo typ tekstury z jaką obecnie mamy do czynienia. Ile poziomów redukcji zostanie wygenerowane, może zostać ustawione z użyciem `glTexParameter()` i parametrów `GL_TEXTURE_BASE_LEVEL` i `GL_TEXTURE_MAX_LEVEL`.

9.4. Tekstury trójwymiarowe

Warto tu wspomnieć, że współczesne karty graficzne, które obsługują OpenGL w większości wypadków świetnie radzą sobie z obsługą danych wolumetrycznych za pośrednictwem tekstur trójwymiarowych. Najczęstszym źródłem danych tego pochodzą z akwizycji współczesnymi aparatami do diagnostyki obrazowej typu tomografów komputerowych lub rezonansu magnetycznego. Jeszcze kilka lat temu możliwość korzystania z renderingu wolumetrycznego zarezerwowana była dla posiadaczy specjalizowanych kart graficznych, których ceny sięgały wielu tysięcy dolarów. Obecnie podobne wydajności są dostępne dla użytkowników kart graficznych 3D zgodnych z OpenGL w wersji 2.0 i wyższej. Tutaj jednak informacja ta pojawia się jako ciekawostka, ponieważ pełna kontrola nad procesem renderowania tego typu obiektów na scenie wymaga również użycia dodatkowo programów cieniowania.



Rysunek 9.5. Tekstura podstawowa i wygenerowane dla niej mipmapy.

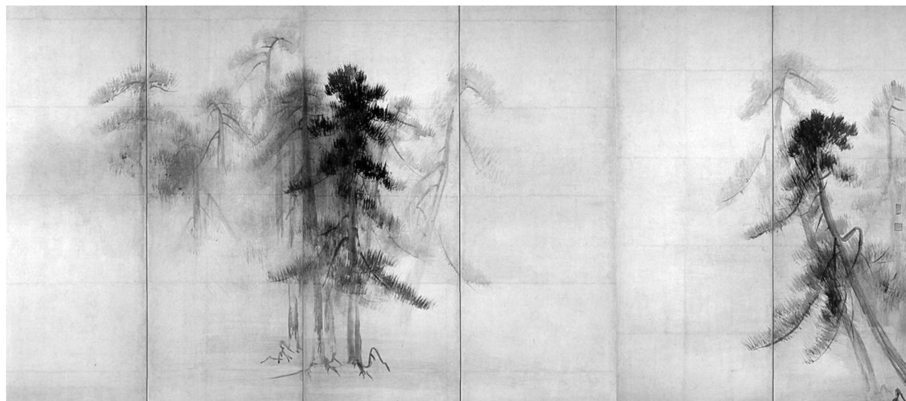
Tabela 9.4. Kolejne argumenty funkcji `pngLoad()` i ich możliwe wartości.

Argument	Wartość	Opis
`filename`		Nazwa pliku PNG wraz z rozszerzeniem.
`mipmap`	liczby całkowite	Głębokość generacji *mipmap*.
	`PNG_BUILDMIPMAPS`	Generacja *mipmap* analogiczna do zaimplementowanej w *gluBuild2DMipmaps()*.
	`PNG_NOMIPMAP`	Brak generacji *mipmap* dla bitmapy.
	`PNG_SIMPLEMIPMAPS`	Generacja każdego kolejnego poziomu *mipmap* z użyciem wartości pochodzących z górnego lewego piksela bloku 2x2.
`trans`	`PNG_ALPHA`	Użycie kanału przezroczystości pliku.
	`PNG_BLEND1..8`	Wygenerowanie kanału przezroczystości w oparciu o pozostałe kanały z wykorzystaniem ośmiu predefiniowanych funkcji.
	`PNG_CALLBACK`	Wygenerowanie kanału przezroczystości z wykorzystaniem funkcji użytkowanika.
	`PNG_SOLID`	Brak przezroczystości
	`PNG_STENCIL`	Zdefiniowanie przezroczystości przez użytkownika w oparciu o wartość składowych koloru.
`info`		Wskaźnik do struktury przechowującej informacje o szerokości, wysokości, głębi kolorów i kanale przezroczystości obrazu zawartego w pliku.

Rozdział 10

Mgła

10.1. Podstawowe parametry mgły **132**
10.2. Konfiguracja zaawansowana **133**



Rysunek 10.1. Las sosnowy we mgle. *Hasegawa Tohaku, ca. 1600.*

Od wieków inspirowała poetów, pisarzy i malarzy. Potem przyszedł czas na filmowców i twórców gier komputerowych. Nie jeden raz skrywała przed naszymi oczami krajobraz, z miejsc znanych czyniła nieznane i pełne zagadek. Odzierała otaczający świat z koloru. Mgła. Od delikatnej mgiełki, wynikiem której jest perspektywa powietrzna pozwalająca na subtelne, ale przemawiające do obserwatora zaprezentowanie głębi po nieprzeniknioną, gęstą mgłę, w której obserwator może odczuwać niepewność i zagubienie.

W OpenGL jest to jeden z niewielu efektów specjalnych dostępnych w standardowym zestawie metod oferowanych użytkownikowi. Współcześnie, gdy mamy do dyspozycji programy cieniowania można, ją uzyskać bez większych problemów. Gdy standard się rodził był, to rodzaj kaprysu twórców. Możliwość kreowania mgły na naszej scenie jest bardzo rozbudowana. Do dyspozycji otrzymaliśmy nie tylko cały szereg metod umożliwiających nam elastyczne konfigurowanie wyglądu mgły, ale również funkcje pozwalające na modyfikację sposobu jej generacji.

10.1. Podstawowe parametry mgły

W celu włączenia obsługi mgły w OpenGL konieczne jest wywołanie funkcji `glEnable()` z podanym parametrem `GL_FOG`. Musi być też włączona obsługa bufora głębokości. Następnym krokiem jest zdefiniowanie koloru zamglenia. Najczęściej przyjmuje się wartość szarości powyżej 50% jasności dla standardowej mgły. Ciemniejszą, czy wręcz czarną, możemy użyć do zasymulowania głębokiej nocy rozświetlanej tylko źródłem światła znajdującym się na pozycji obserwatora. Oczywiście, w takim przypadku konieczne jest też odpowiednie umiejscowienie oświetlenia.

Warto też ustawić aktywny kolor czyszczenia bufora ramki na taki jak przyjęty został kolor mgły w przeciwnym razie tło widoczne w miejscach nie przesłoniętych przez obiekt będzie psuć cały efekt.

Funkcja `glFog()` wraz z odpowiednimi parametrami pozwala na skonfigurowanie wszystkich ustawień dla mgły. Zestawienie możliwych zdefiniowanych stałych zawiera tabela 10.1.

W najprostszym przypadku mgła będzie miała rozkład gęstości liniowy w zakresie podanym przy użyciu `glFog()` z argumentem określającym początek `GL_FOG_START` i koniec `GL_FOG_END` z odpowiednimi wartościami odległości od obserwatora w jednostkach opisywanego świata. Jeżeli nie zostanie przez nas jawnie zdefiniowany ten zakres, to domyślnie przyjmuje się, że zawiera się on w przedziale $< 0.0, 1.0 >$.

Wynikowy kolor fragmentu zależny jest od koloru oryginalnego i wkładu pochodzącego z rozkładu gęstości mgły dla danego miejsca w przestrzeni. Wyrażone to jest w następujący sposób:

$$C = I_f * C_i + (1 - I_f)C_f$$

I_f to współczynnik wkładu koloru mgły i jest obliczany w przypadku rozkładu liniowego z wykorzystaniem następującej formuły:

$$I_f = \frac{end - d}{end - start}$$

gdzie d jest odległością od obserwatora, *start* to początek zakresu zdefiniowanej mgły, a *end* jego koniec.

Listing 10.1. Podstawowa obsługa mgły.

```
1 glEnable(GL_FOG);
  GLfloat fogColor[4] = {0.7, 0.7, 0.7, 1.0};
3 glFogfv(GL_FOG_COLOR, fogColor);
  glFogf(GL_FOG_START, 1.0f);
5 glFogf(GL_FOG_END, 7.0f);
  glFogi(GL_FOG_MODE, GL_LINEAR);
```

10.2. Konfiguracja zaawansowana

W OpenGL możliwe jest również użycie rozkładów gęstości mgły w przestrzeni niż liniowy. Są to dwa typy wykładnicze ze współczynnikiem składowej koloru mgły opisanym w następujący sposób:

$$I_f = e^{-c_e d}$$

i

$$I_f = e^{-(c_e d)^2}$$

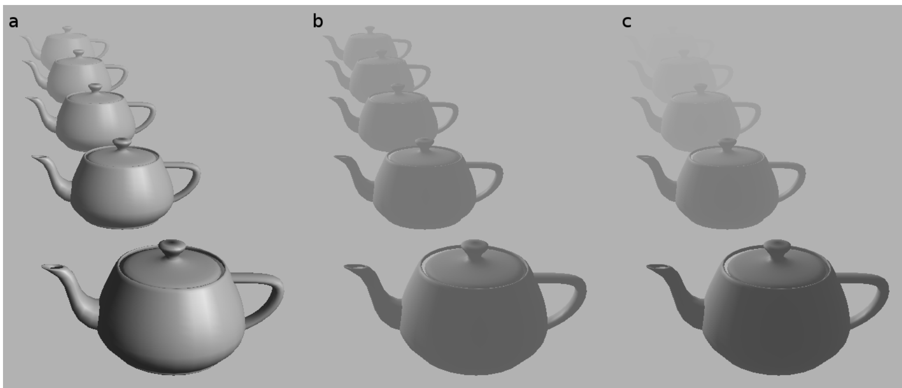
gdzie c_e jest współczynnikiem rozkładu.

Dodatkowo możemy zatroszczyć się o szybkość kalkulacji mgły. Standardowo jest współczynnik składowej wyliczany dla każdego fragmentu obrazu, co nie jest rozwiązaniem wydajnym, ale dającym efekty wysokiej jakości. Możliwe jest wyliczanie jej tylko dla wierzchołków, co znacznie przyspiesza całą operację, ale kosztem jakości, co może być szczególnie widoczne przy obiektach zbudowanych z małej liczby wielokątów. w tym celu korzystamy z funkcji `glHint()` z argumentem pierwszym w postaci parametru `GL_FOG_HINT` i drugim jego wartością ustawioną na `GL_DONT_CARE`, lub `GL_FASTEST`.

Listing 10.2. Rozszerzona obsługa mgły.

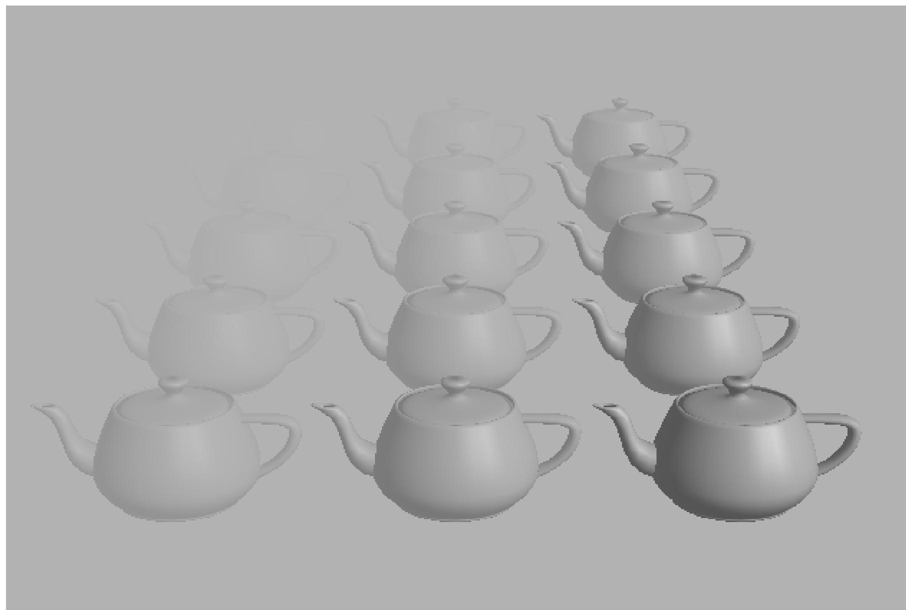
```
glEnable(GL_FOG);
GLfloat fogColor[4] = {0.7, 0.7, 0.7, 1.0};
glFogfv(GL_FOG_COLOR, fogColor);
glFogf(GL_FOG_DENSITY, 0.35);
glFogi(GL_FOG_MODE, GL_EXP);
glHint(GL_FOG_HINT, GL_DONT_CARE);
```

W implementacjach OpenGL od 1.4 wzwyż istnieje również możliwość definiowania mgły o całkowicie odmiennej charakterystyce, o gęstości nie związanej z odległością od obserwatora, a od powierzchni obiektu. Dzięki temu możliwe jest symulowanie mgły ścielącej się po ziemi, albo osnuwającej obiekty na powierzchni zbiorników wodnych.



Rysunek 10.2. Mgła w różnych trybach a) `GL_LINEAR`, b) `GL_EXP` i c) `GL_EXP2`.

W tym celu należy przełączyć tryb generacji na oparty o wartość współczynnika odległości d definiowany dla każdego z wierzchołków za pomocą `glFog()` z parametrem `GL_FOG_COORD_SRC` i jego wartością `GL_FOG_COORD`.



Rysunek 10.3. Rozkład gęstości mgły skojarzony z obiektem.

Użycie mgły w wielu wypadkach umożliwia nam też w łatwy sposób usunięcie elementów, które są niewidoczne dla obserwatora, ponieważ znajdują się za granicą widzialności. o ile nie jest to tak drastycznie istotne w przypadku scen, na które składają się wnętrza pomieszczeń, bo tu możemy dokonać redukcji liczby prymitywów w oparciu o BSP (*Binary Space Partition*) czy systemy portalowe to w przypadku przestrzeni otwartych alternatywą są skomplikowane mechanizmy obsługujące obiekty o zmiennym stopniu szczegółowości.

Tabela 10.1. Wartości stałych pierwszego argumentu `glFog()`.

Stała	Wartości	Zmieniany parametr
GL_FOG_COLOR		Ustala wartość koloru mgły.
GL_FOG_COORD_SRC	GL_FOG_COORD_SRC	Oblicza wkład koloru mgły do koloru wynikowego w oparciu o wartość zdefiniowaną dla wierzchołka.
	GL_FRAGMENT_DEPTH	Oblicza wkład koloru mgły do koloru wynikowego w oparciu o wartość współrzędnej z fragmentu.
GL_FOG_DENSITY	> 0.0	Parametr gęstości dla wykładniczych trybów rozkładu gęstości mgły.
GL_FOG_END	≥ 0.0	Koniec zakresu obszaru mgły względem obserwatora dla liniowego rozkładu gęstości.
GL_FOG_INDEX		Pozycja koloru w palecie w trybach indeksowanych.
GL_FOG_MODE	GL_LINEAR	Liniowy tryb rozkładu gęstości mgły.
	GL_EXP	Wykładniczy tryb rozkładu gęstości mgły.
	GL_LINEAR	Kwadratowo wykładniczy tryb rozkładu gęstości mgły.
GL_FOG_START	≥ 0.0	Początek zakresu obszaru mgły względem obserwatora dla liniowego rozkładu gęstości.

Dodatek A

Parametry funkcji `pngBind()`

Tabela A.1. Kolejne argumenty funkcji pngBind() i ich możliwe wartości.

Argument	Wartość	Opis
filename		Nazwa pliku PNG wraz z rozszerzeniem.
mipmap	liczby całkowite	Głębokość generacji *mipmap*.
	PNG_BUILDMIPMAPS	Generacja *mipmap* analogiczna do zaimplementowanej w *gluBuild2DMipmaps()*.
	PNG_NOMIPMAP	Brak generacji *mipmap* dla bitmapy.
	PNG_SIMPLEMIPMAPS	Generacja każdego kolejnego poziomu *mipmap* z użyciem wartości pochodzących z górnego lewego piksela bloku 2x2.
trans	PNG_ALPHA	Użycie kanału przezroczystości pliku.
	PNG_BLEND1..8	Wygenerowanie kanału przezroczystości w oparciu o pozostałe kanały z wykorzystaniem ośmiu predefiniowanych funkcji.
	PNG_CALLBACK	Wygenerowanie kanału przezroczystości z wykorzystaniem funkcji użytkowanika.
	PNG_SOLID	Brak przezroczystości
	PNG_STENCIL	Zdefiniowanie przezroczystości przez użytkownika w oparciu o wartość składowych koloru.
info		Wskaźnik do struktury przechowującej informacje o szerokości, wysokości, głębi kolorów i kanale przezroczystości obrazu zawartego w pliku.
wrapst	GL_CLAMP	Zawijanie tekstury z duplikowaniem brzegowych pikseli w kierunkach S i T.
	GL_REPEAT	Powtarzanie tekstury w kierunkach S i T.
minfilter	Patrz tabela 9.1	Metoda interpolacji gdy rozmiar teksela jest mniejszy niż fragmentu.
magfilter	Patrz tabela 9.1	Metoda interpolacji gdy rozmiar teksela jest większy niż fragmentu.

Dodatek B

Miejsca w sieci

http://www.opengl.org/
Pierwsze miejscem do którego należy się udać jeżeli poszukuje się informacji o OpenGL. Zawiera dokumentację referencyjną, informacje o projektach zależnych, produktach korzystających z technologii. Ważnym fragmentem serwisu są też strony społecznościowe, ze szczególnym uwzględnieniem prawdopodobnie największego forum poświęconego tej bibliotece i jej użyciu.

http://www.khronos.org/
Organizacja odpowiedzialna za rozwój standardów OpenGL, OpenGL ES, OpenGL SC, WebGL i innych. Wśród zasobów jest aktualna i kompletna lista specyfikacji API. Serwis zawiera też katalog z szeroką bazą kursów, poradników i przykładowych programów.

http://www.mesa3d.org/
Strona projektu programowej implementacji standardu OpenGL. Pozwala na korzystanie z aplikacji tworzony dla OpenGL w systemach nie posiadających wsparcia sprzętowego lub posiadających dla starszych wersji.

http://nehe.gamedev.net/
Wieloczęściowy kurs programowania OpenGL. Dużo kodu, wiele interesujących technik. Minusem jest bardzo lakoniczny komentarz i brak szerszego odniesienia do zasad działania prezentowanych rozwiązań.

http://developer.nvidia.com/page/opengl.html
Dokumentacja związana ze wsparciem ze strony firmy Nvidia standardu OpenGL oraz własnych rozszerzeń.

http://jogamp.org/
Biblioteka umożliwiająca obsługę OpenGL z wykorzystaniem języka Java. Zgodna ze specyfikacją JSR-231.

http://developer.nvidia.com/object/cg_toolkit.html
Cg - alternatywa dla języka GLSL, która może być wykorzystana do tworzenia programów cieniowania OpenGL. Zawiera mechanizm obsługi profili dostosowujących używane metody do możliwości sprzętu.

http://www.swiftless.com/
Serwis zawierający kurs programowania OpenGL i podstaw GLSL oraz ich praktycznego użycia przy tworzeniu wizualizacji terenu.

Bibliografia

[1] Tom McReynolds, David Blythe, "Advanced Graphics Programming Techniques Using OpenGL", SIGGRAPH 1998

[2] Markus Hadwiger, Patric Ljung, Christof Rezk Salama, Timo Ropinski, "Advanced Illumination Techniques for GPU-Based Volume Raycasting", SIGGRAPH ASIA, 2008

[3] Norman Chin, Chris Frazier, Paul Ho, Zicheng Liu, Kevin P. Smith, "The OpenGL Graphics System Utility Library (Version 1.3)", Silicon Graphics, Inc. 1998, `http://www.opengl.org/documentation/specs/glu/glu1_3.pdf`

[4] Paula Womack, Jon Leech, "OpenGL Graphics with the X Window System", Silicon Graphics, Inc. 1998, `http://www.opengl.org/documentation/specs/glx/glx1.3.pdf`

[5] Mark J. Kilgard, "The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3", Silicon Graphics, Inc. 1996, `http://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf`

[6] Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis, "OpenGL Programming Guide. The Official Guide to Learning OpenGL, Versions 3.0 and 3.1", Addison-Wesley Professional, 2010

[7] Kevin Hawkins, Dave Astle, "OpenGL. Programowanie gier", Helion 2003

[8] Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, Mike Weiblen, "OpenGL Shading Language", Addison-Wesley Professional 2009

[9] Richard S. Wright, Nicholas Haemel, Graham Sellers, Benjamin Lipchak, "OpenGL SuperBible. Comprehensive Tutorial and Reference", Addison-Wesley Professional, 2010

[10] Janusz Ganczarski, "OpenGL w praktyce", Wydawnictwo BTC, 2008.

Skorowidz

FreeGLUT, 16

glArrayElement(), 101
glBegin(), 39
glCallList(), 98
glClearColor(), 95
glClearDepth(), 95
glClearStencil(), 95
glColorMaterial(), 115
glCullFace(), 46
glDisable(), 9
glDisableClientState(), 100
glDrawElements(), 102
glEnable(), 9, 95
glEnableClientState(), 100
glEnd(), 39
glEndList(), 98
glFlush(), 21
glFog(), 133
glFrontFace(), 46
glFrustrum, 88
glGenerateMipmap(), 127
glGenLists(), 98
glGenTextures(), 121
glGetIntegerv(), 119
glHint(), 121, 134
glLight(), 109
glLightModel(), 107
glLineStipple(), 40
glLineWidth(), 40
glLoadIdentity(), 60
glLoadMatrix(), 83
glMaterial(), 113
glMatrixMode(), 60, 83
glMultiDrawElements(), 103
glNewList(), 98
glNormalPointer(), 100
glOrtho(), 19, 86
GLPNG, 118
glPointSize(), 40
glPolygonMode(), 47
glPolygonStipple(), 47
glPopAttrib(), 99
glPopMatrix(), 65
glPushAttrib(), 99
glPushMatrix(), 65
glRasterPos2f(), 29
glRasterPos3f(), 29
glRotatef(), 63
glScalef(), 64
glShadeModel(), 96
glTexCoord(), 122
glTexParameter(), 122
glTranslatef(), 61
GLU, 49
gluCylinder(), 50
gluDeleteQuadric(), 51
gluDisk(), 50
GLUI, 35
gluLookAt(), 89
gluNewQuadric(), 49
gluPartialDisk(), 50
gluPerspective(), 19
gluQuadricDrawStyle(), 51
gluQuadricOrientation(), 51
gluSphere(), 50
GLUT, 16
glutAddMenuEntry(), 31
glutAddSubMenu(), 34
glutAttachMenu(), 31
glutBitmapCharacter(), 27
glutBitmapLength(), 29
glutBitmapWidth(), 29
glutCreateMenu(), 31
glutCreateWindow(), 19
glutDisplayFunc(), 19

glutEntryFunc(), 26
glutGet(), 26
glutGetModifiers(), 22
glutIdleFunc(), 22
glutInitDisplayMode(), 18
glutInitWindowPosition(), 18
glutInitWindowSize(), 17
glutKeyboardFunc(), 23
glutMainLoop(), 19
glutMotionFunc(), 25
glutMouseFunc(), 25
glutPassiveMotionFunc(), 25
glutRemoveMenuItem(), 33
glutReshapeFunc(), 21
glutSetMenu(), 35
glutSolidCone(), 54
glutSolidCube(), 53
glutSolidDodecahedron(), 53
glutSolidIcosahedron(), 53
glutSolidOctahedron(), 53
glutSolidSphere(), 54
glutSolidTeapot(), 54
glutSolidTetrahedron(), 53
glutSpecialFunc(), 23
glutStrokeCharacter(), 28
glutStrokeLength(), 29
glutStrokeWidth(), 29
glutSwapBuffers(), 21
glutWireCone(), 54
glutWireCube(), 53
glutWireDodecahedron(), 53
glutWireIcosahedron(), 53
glutWireOctahedron(), 53
glutWireSphere(), 54
glutWireTeapot(), 54
glutWireTetrahedron(), 53
glVertex(), 39
glVertexPointer(), 100
glViewport(), 87

IRIS GL, 3

Mesa 3D, 9

NURBS, 49
NvTriStrip, 43

OpenGLUT, 16

pngLoad(), 118

RadiosGL, 107

Spacewars, 2

Tri Stripper, 43