Jacek Matulewski
http://www.fizyka.umk.pl/~jacek/

OpenGL (C++Builder)

Wersja pre-α

Toruń, 20 października 2007

Najnowsza wersja skryptu oraz kod źródłowy dostępne pod adresem
http://www.fizyka.umk.pl/~jacek/dydaktyka/3d/

Spis treści

Spis treści 2
Wstęp 4
1. Rysowanie figur w przestrzeni 3D 5
1.1 Inicjacja grafiki OpenGL w aplikacji projektowanej w środowisku C++Builder 5
1.2 Rysowanie figury płaskiej (trójkąta). Podwójne buforowanie 8
1.3 Poprawianie geometrii frustum 10
1.4 Kolor 11
1.5 Cieniowanie kolorów na powierzchniach 12
1.6 Kolor tła 13
1.7 Rysowanie figury przestrzennej (ostrosłupa) 14
1.8 Wyodrębnienie metody rysującej figurę 15
2. Projekt klasy TGLForm implementującej formę przystosowaną do wyświetlania grafiki OpenGL 16
2.1 Definiowanie klasy TGLForm 17
2.2 Obsługa komunikatu WM_PAINT 19
3. Obroty i przesunięcia 19
3.1 Obroty obiektów na scenie. Ruch kamery i ruch aktorów 19
3.2 Przesunięcia obiektu 22
3.3 Prosta animacja 23
3.4 Bardziej precyzyjne ustawianie kamery 24
4. Kontrola położenia kamery za pomocą myszy 25
4.1 Ruch kamery kontrolowany myszą (lewy przycisk myszy) 26
4.2 Odległość kamery (rolka) 28
4.3 Przesuwanie sceny myszą (prawy klawisz myszy) 28
4.4 Inicjowane myszą swobodne obroty kamery 30
4.5 Dodanie wygaszania obrotów 32
4.6 Funkcja pozwalająca na inicjację obrotów 33
5. Nowi aktorzy 33
5.1 Rysowanie osi układu współrzędnych 33
5.2 Dodawanie kolejnych figur 34
5.3 Stos macierzy model-widok 35
5.4 Rysowanie sześcianu 36
5.5 Maksymalizacja okna klawiszem Enter 37
5.6 Tryb (pseudo) pełnoekranowy 38
6. Kolor i światło 38
6.1 Włączenie systemu oświetlenia i ustawianie światła tła 39
6.2 Uzgadnianie koloru „fizycznego” przedmiotów z kolorem ustalanym funkcją glColor 41
6.3 Ilu programistów potrzeba, aby wkręcić mleczną żarówkę 42
6.4 Definiowanie wektorów normalnych (sześcian) 43
6.5 Definiowanie wektorów normalnych (ostrosłup) 44
6.6 Barwne światło rozproszone 47
6.7 Uśrednianie normalnych 48
6.8 Gładkie materiały (rozbłysk) 49
6.9 Ustawianie reflektora 50
6.10 Montowanie włączników światła 51
6.11 Nieruchomy pokój 52
7. Mieszanie kolorów (alpha blending) 54
7.1 Inicjacja mieszania kolorów 54
7.2 Przezroczystość 54
7.3 Antyaliasing 56
7.4 Mgła 57
8. Teksturowanie 59
8.1 Przygotowanie projektu do teksturowania: 59
8.2 Wczytywanie obrazu tekstury 60
8.3 Wiązanie tekstury 61
8.4 Lakierowanie tekstury 63
8.5 Korzystanie z wielu tekstur 64
9. Napisy 66
9.1 Czcionki bitmapowe 66
9.2 Czcionki 3D 68
10. Szablon – ostatnie szlify 70
A. Biblioteka GLU. Kwadryki 71

A.1 Definiujemy kwadrykę i rysujemy sferę 71
A.2 Styl rysowania kwadryki 72
A.3 Teksturowanie kwadryki 73
A.4 Przegląd kwadryk 74
B. Krzywe i płaszczyzny Béziera 75
B.1 Dwuwymiarowe krzywe Béziera 75
B.2 Trójwymiarowe krzywe Béziera 77
B.3 Powierzchnie Beziera 78
B.4 Oświetlanie powierzchni Béziera 81

Poprawki ze skryptu!!!!

Metoda, funkcja
Numeracja rysunków i listingów bez 11
Uwagi ze skryptu!!!

Wstęp

OpenGL to biblioteka funkcji realizujących operacje graficzne pozwalające na budowanie trójwymiarowych scen (zbiorów figur), oświetlanie ich, dodawanie różnego typu efektów zwiększających wrażenie realności oraz na ich prezentację na ekranie. O OpenGL można myśleć jak o interfejsie programistycznym (API) karty graficznej. Lub odwrotnie, jak o standardzie (zbiorze poleceń), który jest implementowany przez większość kart graficznych. Tak czy inaczej — renderowanie (przygotowywanie obrazu przestrzennej sceny na płaskim ekranie) wspomagane jest sprzętowo przez procesor karty graficznej (GPU), dzięki czemu jest bardzo wydajne i nie obciąża przy tym głównego procesora komputera (CPU). To pozwala na generowanie wielu scen w ciągu sekundy i ich płynną animację.

OpenGL, podobnie jak konkurencyjny Direct3D (składnik DirectX odpowiedzialny za grafikę trójwymiarową), może służyć do tworzenia gier, programów CAD czy programów do wizualizacji naukowych. W tych ostatnich zastosowaniach dobrym przykładem są programy do prezentacji niezwykle skomplikowanych struktur przestrzennych białek, co bez technik 3D byłoby trudne do wyobrażenia. Z rynku gier OpenGL został niestety wyparty przez Direct3D i to pomimo powszechnej opinii, że OpenGL jest równie wydajny, a jednocześnie bardziej elegancki i wygodniejszy w użyciu. To ostatnie twierdzenie przestało być tak oczywiste od momentu wprowadzenia wersji 8 DirectX, a szczególnie w stosunku do DirectX.NET, który jest zorientowany obiektowo. Warto podkreślić zasadniczą różnicę między DirectX i OpenGL. Ten pierwszy jest własnością Microsoftu i jest rozwijany wyłącznie przez tę korporację na potrzeby jednego systemu operacyjnego — Windows, podczas gdy OpenGL jest standardem otwartym, dostępnym na wielu platformach, implementowanym przez wielu producentów kart graficznych[1] i otwartym na rozszerzenia. Jego rozwój nadzorowany jest przez ARB — konsorcjum firm, wśród których w chwili obecnej znajdują się m.in. ATI, NVidia, 3DLabs, SGI (pierwotny twórca tej biblioteki), Sun, Apple, IBM, Dell i Intel. A więc sami giganci. Nie ma już między nimi Microsoftu, pomimo że był jedną z korporacji-założycieli. Więcej informacji na temat OpenGL, jego specyfikacji, historii, ARB i samym API można znaleźć na jego oficjalnej stronie *http://www.opengl.org/*. Warto również zajrzeć na stronę twórcy biblioteki SGI: *http://www.sgi.com/products/software/opengl/*. Poza tym w internecie jest bardzo dużo stron zawierających opisy, poradniki i zbiory przykładów, które łatwo odnaleźć. Są one łatwo rozpoznawane na pierwszy rzut oka, dzięki obecnemu na nich logo OpenGL (rysunek 1).

Rysunek 1
Oficjalne logo OpenGL

Zanim rozpoczniemy zabawę, chciałbym uprzedzić, że celem tego rozdziału nie jest systematyczne wprowadzenie do OpenGL. Na to potrzeba osobnych książek, a nie rozdziałów[2]. Moim zamiarem jest jedynie zainteresowanie Czytelnika możliwościami tej biblioteki, przełamanie obawy przed projektowaniem aplikacji wykorzystującej grafikę trójwymiarową i pokazanie, że OpenGL jest stosunkowo łatwy do nauczenia. Postaram się to udowodnić w serii dwudziestu sześciu ćwiczeń, które zaczniemy od rysowania trójkątów i figur przestrzennych, potem przejdziemy do obracania ich i przesuwania, następnie omówimy oświetlenie, a zakończymy na teksturowaniu kwadryk. Postaram się przy tym, aby atrakcyjność przykładów nie została stłumiona przez nadmiernie teoretyczny komentarz, ale tego nie da się zawsze uniknąć.

[1] Istnieje również implementacja OpenGL 1.1, przygotowana przez Microsoft, która jest dołączana do Windows.
[2] Jednym z ciekawszych kompendiów wiedzy o OpenGL jest *OpenGL. Księga eksperta (Wydanie III)* napisana przez Richarda S. Wrighta i Benjamina Lipchaka (Helion 2005).

1. Rysowanie figur w przestrzeni 3D

1.1 Inicjacja grafiki OpenGL w aplikacji projektowanej w środowisku C++Builder

Rozpoczniemy od przystosowania aplikacji projektowanej w środowisku C++Builder do współpracy z biblioteką OpenGL. Jest to krok dość żmudny, ale na szczęście wystarczy zrobić go tylko raz; w kolejnych projektach możemy już korzystać z gotowego szablonu.

1. Tworzymy nowy projekt *VCL Forms Application — C++Builder*. W widoku projektowania powiększamy formę.
2. Przechodzimy do kodu źródłowego nagłówka *Unit1.h* i importujemy pliki nagłówkowe *gl.h* i *glu.h* (listing 1). Drugi z tych plików będzie tak naprawdę potrzebny dopiero w dalszych projektach.
3. W klasie `TForm1` deklarujemy dwie prywatne metody pomocnicze i dwa prywatne pola:

Listing 1. Pełny kod nagłówka Unit1.h

```
//---------------------------------------------------------------------------

#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>

#include <gl.h>
#include <glu.h>

//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:    // IDE-managed Components
   void __fastcall FormCreate(TObject *Sender);
   void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
private:        // User declarations
   HDC uchwytDC; //uchwyt do "display device context (DC)"
   HGLRC uchwytRC; //uchwyt do "OpenGL rendering context"
   bool GL_UstalFormatPikseli(HDC uchwytDC);
   void GL_UstawienieSceny();
public:         // User declarations
   __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

Przechowanie uchwytów do kontekstu renderowania jest konieczne, aby możliwe było usunięcie go przy zamknięciu okna.

1. Definicję pierwszej metody pokazuje listing 2.

Listing 2. Implementacja metody inicjującej aplikację OpenGL

```
bool TForm1::GL_UstalFormatPikseli(HDC uchwytDC)
{
   PIXELFORMATDESCRIPTOR opisFormatuPikseli;
   ZeroMemory(&opisFormatuPikseli,sizeof(opisFormatuPikseli));
   opisFormatuPikseli.nVersion=1;
   opisFormatuPikseli.dwFlags=PFD_SUPPORT_OPENGL | PFD_DRAW_TO_WINDOW |
   PFD_DOUBLEBUFFER; //w oknie, podwojne buforowanie
   opisFormatuPikseli.iPixelType=PFD_TYPE_RGBA; //typ koloru RGB
   opisFormatuPikseli.cColorBits=32; //jakosc kolorów 4 bajty
```

```
    opisFormatuPikseli.cDepthBits=16; //glebokosc bufora Z (z-buffer)
    opisFormatuPikseli.iLayerType=PFD_MAIN_PLANE;

    int formatPikseli=ChoosePixelFormat(uchwytDC,&opisFormatuPikseli);
    if (formatPikseli==0) return false;
    if (SetPixelFormat(uchwytDC,formatPikseli,&opisFormatuPikseli)!=true) return
    false;
    return true;
}
```

2. A drugiej — listing 3:

Listing 3. Ustawienia dotyczące „filmowanej" sceny

```
void TForm1::GL_UstawienieSceny()
{
    glViewport(0,0,ClientWidth,ClientHeight); //okno OpenGL = wnetrze formy
    (domyslnie)

    //ustawienie punktu projekcji
    glMatrixMode(GL_PROJECTION); //przełączenie na macierz projekcji
    glLoadIdentity();
    //left,right,bottom,top,znear,zfar (clipping)
    glFrustum(-0.1, 0.1, -0.1, 0.1, 0.3, 100.0); //mnozenie macierzy rzutowania przez
    macierz perspektywy - ustalanie frustum
    glMatrixMode(GL_MODELVIEW); //powrót do macierzy widoku modelu
    glEnable(GL_DEPTH_TEST); //z-buffer aktywny = ukrywanie niewidocznych
    powierzchni
}
```

3. Tworzymy metodę zdarzeniową do `OnCreate` formy i budujemy w niej kontekst OpenGL związany z bieżącym oknem (listing 4):

Listing 4. Konieczne jest pobranie i przechowywanie uchwytu kontekstu graficznego okna — jest on potrzebny zarówno do renderowania, jak i przy zamykaniu aplikacji do zwolnienia zasobów używanych przez OpenGL

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    //biezace okno staje sie oknem OpenGL
    uchwytDC=GetDC(Handle);
    if (!GL_UstalFormatPikseli(uchwytDC)) ShowMessage("Nie udało się ustalić formatu
    pikseli");
    uchwytRC=wglCreateContext(uchwytDC);
    if (uchwytRC==NULL) ShowMessage("Nie udało się pobrać uchwytu kontekstu
    grafiki");
    if (!wglMakeCurrent(uchwytDC,uchwytRC)) ShowMessage("Inicjacja grafiki OpenGL
    nie powiodła się");
    GL_UstawienieSceny();
    Caption=(AnsiString)"OpenGL "+(char*)glGetString(GL_VERSION);
}
```

4. Musimy pamiętać o usunięciu kontekstu OpenGL (kontekstu renderowania) przy zamknięciu okna. W tym celu tworzymy metodę zdarzeniową do `OnClose` formy i umieszczamy w niej polecenia z listingu 5:

Listing 5. Zwalnianie zasobów zarezerwowanych podczas uruchamiania aplikacji

```
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
    wglMakeCurrent(NULL,NULL);
    wglDeleteContext(uchwytRC);
    ReleaseDC(Handle,uchwytDC);
    PostQuitMessage(0);
}
```

Po wykonaniu metody `FormCreate` główne okno aplikacji jest przygotowane do rysowania za pomocą OpenGL. W metodzie tej odczytywany jest uchwyt do kontekstu urządzenia odpowiedzialnego za przygotowywanie grafiki w oknie aplikacji (ang. *display device context*; funkcja `GetDC`) i tworzony jest kontekst związany z przygotowywaniem grafiki za pomocą biblioteki OpenGL (`wglCreateContext`), która do rysowania wykorzystywać będzie to okno (za to przypisanie odpowiada funkcja `wglMakeCurrent`). Jest to dość zawiłe, więc nie będę przezwyciężał swojej i Czytelnika zdrowej niechęci przed głębszym wnikaniem w to zagadnienie.

Ważniejsze jest moim zdaniem, abyśmy dobrze zrozumieli ramy, w jakich działa OpenGL, jak inicjowana jest jej maszyna stanów[3], która przechowuje ustawienia sceny, za co odpowiedzialna jest metoda `GL_UstawienieSceny`.

Grafika trójwymiarowa, podobnie jak mechanika klasyczna, opiera się na korzystaniu z macierzy 3×3, które opisują różnego rodzaju przekształcenia w przestrzeni. Za pomocą takiej macierzy możemy zapisać przesunięcie, obrót oraz skalowanie figur, a z ich złożeń (iloczynów macierzy) zbudować dowolne przekształcenia[4]. W rzeczywistości macierze przekształceń używane w OpenGL mają rozmiar 4×4, gdzie dodatkowa współrzędna wektora opisującego punkt odgrywa rolę współczynnika skalowania (zob. prezentacja nt. współrzędnych jedorodnych). Większości przekształceń odpowiadają wygodne w użyciu funkcje (podstawowy zbiór tych funkcji zebrany został w tabeli 11.1), które budują odpowiednią macierz i mnożą przez nią bieżącą **macierz model-widok**. Czym jest macierz model-widok? To macierz zbierająca wszystkie przekształcenia[5] oryginalnego układu punktów zdefiniowanych na scenie (modelu) i transformująca je do postaci widzianej przez kamerę. Domyślnie macierz model-widok jest jednostkowa, co oznacza brak przekształceń — taką ją też zostawia metoda `GL_UstawienieSceny`.

Tabela 11.1. *Funkcje OpenGL mają wiele odmian przyjmujących różne typy argumentów*

Rodzaj przekształcenia	Funkcja (nazwa bez przyrostków)	Najczęściej wykorzystywana wersja
Przesunięcie (translacja)	`glTranslate`	`glTranslatef`
Obrót	`glRotate`	`glRotatef`
Skalowanie	`glScale`	`glScalef`

Pomnożenie oryginalnych współrzędnych sceny przez macierz model-widok to jednak dopiero pierwszy etap przygotowywania obrazu, jaki widzimy na ekranie. Kolejnym etapem jest rzutowanie. Związana jest z nim druga macierz nazywana **macierzą rzutowania**. Polecenie `glMatrixMode(GL_PROJECTION);`, które znajduje się w metodzie `GL_UstawienieSceny`, zmienia bieżącą macierz (tj. macierz, której dotyczą funkcje z tabeli 11.1) z macierzy model-widok identyfikowanej przez stałą `GL_MODELVIEW` na macierz rzutowania (stała `GL_PROJECTION`). Jeżeli chcemy, możemy ją przesunąć, obrócić czy powiększyć. Jednak nie ma to większego sensu. Zamiast tego zmienimy ją tak, aby przekształcała scenę w taki sposób, że rzeczy odległe są mniejsze od tych, które znajdują się bliżej kamery. Do tego właśnie służy macierz rzutowania — może wprowadzać rzut perspektywiczny. Rzutowanie to nie zachowuje oczywiście pierwotnych odległości pomiędzy punktami sceny. Alternatywą jest rzut izometryczny (prostopadły, ortogonalny), w którym przedmioty znajdujące się dalej od kamery są tej samej wielkości, co bliższe niej. Ten sposób rzutowania może wydawać się nierealistyczny i wobec tego nieprzydatny, ale znają go np. wszyscy miłośnicy gier fabularnych, w których sprawdza się doskonale. Co robimy z macierzą rzutowania? Modyfikujemy ją za pomocą funkcji `glFrustum`, która ustala własności **frustum**. Jeszcze jedno nowe pojęcie? Frustum to po prostu obszar, na którym budowana jest scena — wycinek przestrzeni, który „filmuje" kamera, ograniczony od strony kamery przez ekran. Przedmioty znajdujące się poza frustum nie są renderowane. W zależności od stosowanego typu rzutowania frustum może mieć kształt prostopadłościanu, którego jedną ze ścian jest ekran (rzutowanie ortogonalne), lub, jak jest w powyższym przykładzie, widocznego na rysunku 2 ostrosłupa o podstawie prostokąta ze ściętym czubkiem (rzutowanie perspektywiczne). Widoczna na listingu 3 funkcja `glFrustum` mnoży jednostkową macierz rzutowania[6] przez macierz perspektywy wygenerowaną na podstawie argumentów funkcji, generując w efekcie rzut perspektywiczny:

```
glFrustum(lewo,prawo,dół,góra,blisko,daleko);
```

Argumenty wyznaczają położenie ekranu w układzie odniesienia, w którym kamera (niektórzy wolą mówić „oko") znajduje się w środku układu współrzędnych (punkt `(0,0,0)`). Oznacza to, że dwa ostatnie argumenty wyznaczające najbliższy i najdalszy plan powinny być większe od zera. W powyższym przykładzie mają one wartości odpowiednio `0.3` i `100.0`. Należy pamiętać, że ich różnica wyznacza stopień zniekształcenia obrazu — jak wspomniałem, w rzucie perspektywicznym kąty i odległości nie są zachowane. Jeżeli *`blisko`* będzie bliskie

[3] W rozdziale 9. wspomniałem, że biblioteki DLL mogą być nie tylko zbiorem funkcji, ale również posiadać stan przechowywanych w jej zmiennych globalnych. Maszyna stanu OpenGL to właśnie zbiór zmiennych biblioteki OpenGL przechowujących ustawienia dotyczące przygotowywania grafiki 3D, oświetlenia itp.

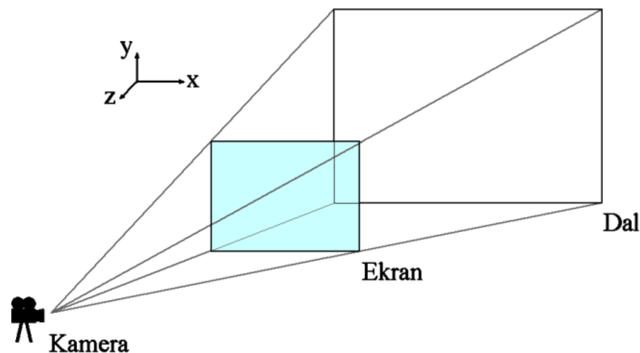
[4] Nie wymieniłem odbicia względem osi i punktu, bo OpenGL nie posiada osobnych funkcji, które realizowałyby te przekształcenia.

[5] Należy pamiętać, że iloczyn dwóch macierzy 4×4 jest nadal macierzą 4×4, a więc jedną macierzą można opisywać dowolne złożenie przekształceń.

[6] Macierz jednostkowa została przygotowana funkcją `glLoadIdentity`.

zera, to figury będą się szybko zmniejszać w miarę oddalania się od kamery i szybko przestaną być widoczne. Można się przekonać o tym, zmieniając na chwilę ten parametr z `0.3` na `0.03` i uruchamiając aplikację.

Rysunek 2.
Frustum w rzucie perspektywicznym



Przestrzeń ograniczana przez frustum opisana jest za pomocą prawoskrętnego układu współrzędnych, którego osie OX i OY skierowane są odpowiednio w prawo i do góry względem płaszczyzny ekranu, natomiast oś OZ skierowana jest od dali do kamery[7]. Domyślne położenie środka układu współrzędnych, względem którego ustalane jest położenie figur, zgodne jest z położeniem kamery. Zatem położenie płaszczyzny ekranu na osi OZ w naszym przykładzie to `-0.3`, a najdalszego planu `-100.0` (obie są ujemne). Środek układu współrzędnych znajduje się w ten sposób poza frustum. Rysowanie sceny zaczniemy od jej przesunięcia o `-10`, a więc w kierunku dali, aby środek współrzędnych był widoczny na ekranie (znajdował się wewnątrz frustum). Po zmodyfikowaniu macierzy rzutowania przywracamy macierz model-widok jako bieżącą macierz (polecenie `glMatrixMode(GL_MODELVIEW);`). To na niej wykonywać będziemy wszystkie pozostałe przekształcenia w programie. W istocie macierz rzutowania po jej zainicjowaniu rzadko jest zmieniana. Metoda `GL_UstawienieSceny` kończy się poleceniem `glEnable (GL_DEPTH_TEST);`, którym włączamy **bufor głębi**, odpowiedzialny za kontrolowanie odległości figur od kamery i ich odpowiednie wzajemne przesłanianie.

Podsumowując, przypomnijmy, z jakimi macierzami mamy do czynienia w OpenGL. Po pierwsze jest to macierz model-widok. Jeżeli na scenie narysowana jest postać człowieka stojącego na płaszczyźnie OXZ (głowa skierowana jest w kierunku dodatnim osi OY), a kamera wisi nad nim i jest skierowana w dół, to przekształcenie zadane przez macierz model-widok spowoduje, że pierwotny układ współrzędnych zostanie przesunięty tak, że jego początek będzie znajdował się w położeniu kamery oraz będzie obrócony wokół osi OX o 90 stopni przeciwnie do kierunku ruchu wskazówek zegara (patrząc w stronę dodatniej półosi OX). **Macierz model-widok mówi zatem, jak zmienić układ odniesienia sceny na układ odniesienia kamery.** Drugą macierzą jest macierz rzutowania, która może wprowadzić do obrazu perspektywę. Końcowym etapem jest przygotowanie obrazu dwuwymiarowego (tzw. **przekształcenie widoku**), który wyświetlany jest na ekranie. Przekształcenie widoku możemy wyobrazić sobie bardzo łatwo: gdybyśmy stanęli przed oknem i, trzymając głowę nieruchomo, obrysowali flamastrem na szybie kontury wszystkich budynków widzianych za oknem — powstałby dwuwymiarowy rzut trójwymiarowej przestrzeni. Mniej więcej na tym polega przekształcenie widoku (nie opisuje go już oczywiście macierz 3×3). Te trzy przekształcenia tworzą tak zwany **potok**, który obejmuje cały proces przygotowywania obrazu przez OpenGL.

1.2 Rysowanie figury płaskiej (trójkąta). Podwójne buforowanie

Trójkąty są szczególnie ważną figurą w grafice 3D, ponieważ można z nich zbudować dowolną bryłę przestrzenną.

1. Deklarujemy metodę `RysujScene` zgodnie ze wzorem z listingu 6.

Listing 6. Deklaracja metody renderującej

```
class TForm1 : public TForm
```

[7] Proszę zwrócić uwagę, że zwykle oś OY skierowana jest w dół ekranu. OpenGL jest w tym względzie zgodne z intuicją uzyskaną w szkole podstawowej. Podobnie jest z osią OZ, która jest skierowana „do nas”, a nie jak w Direct3D „od nas”. Dzięki temu układ współrzędnych jest prawoskrętny, co pozwala na stosowanie bez dodatkowych przekształceń wzorów z podręczników do geometrii.

```
{
__published:   // IDE-managed Components
   void __fastcall FormCreate(TObject *Sender);
   void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
private:      // User declarations
   HDC uchwytDC; //uchwyt do "display device context (DC)"
   HGLRC uchwytRC; //uchwyt do "OpenGL rendering context"
   bool GL_UstalFormatPikseli(HDC uchwytDC);
   void GL_UstawienieSceny();
   void __fastcall RysujScene();
public:       // User declarations
   __fastcall TForm1(TComponent* Owner);
};
```

2. Następnie definiujemy ją zgodnie z listingiem 7. Trójkąt rysowany jest w pobliżu punktu `(0,0,0)`, ale przesuniętego o `10` w dal[8].

Listing 7. Definicja metody renderującej

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); //czysci bufory
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //Rysowanie trojkata
   glBegin(GL_TRIANGLES);
   //ustalanie trzech wierzcholkow trojkata (werteksow (x,y,z))
   //(0,0,z) jest mniej wiecej w srodku ekranu
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, y0, z0); //gorny
   //koniec rysowania figury
   glEnd();

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```

Przyrostek nazwy wskazuje na ilość i typ przeciążonych funkcji; np. `glVertex3f` oznacza, że przyjmowane są trzy argumenty typu `float`. Funkcja `glVertex` ma znacznie więcej odmian (w sumie około dwudziestu pięciu). W dokumentacji dołączonej do BDS informacji o funkcjach OpenGL należy szukać pod hasłem bez przyrostka, a więc `glVertex`, a nie `glVertex3f`.

3. Za pomocą inspektora obiektów tworzymy metodę związaną ze zdarzeniem `OnPaint` formy i umieszczamy w niej wywołanie metody `RysujScene` (listing 8). Dzięki temu rysunek sceny będzie odświeżany.

Listing 8. Do zainicjowania renderowania sceny wykorzystujemy zdarzenie OnPaint formy

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
   RysujScene();
}
```

Metodę `RysujScene` otwiera polecenie przypisujące macierzy model-widok (będącej macierzą bieżącą) macierz jednostkową (funkcja `glLoadIdentity`), która oznacza, że układ odniesienia sceny i kamery są jednakowo ustawione. Nie trwa to jednak długo, bo znajdujące się w następnej linii polecenie `glTranslatef(0.0,0.0,-10.0);` rozsuwa scenę i kamerę o `10.0` jednostek. Dzięki temu rozsunięciu trójkąt, który rysujemy w pobliżu środka układu współrzędnych, znajduje się wewnątrz frustum i możemy zobaczyć go na ekranie.

---

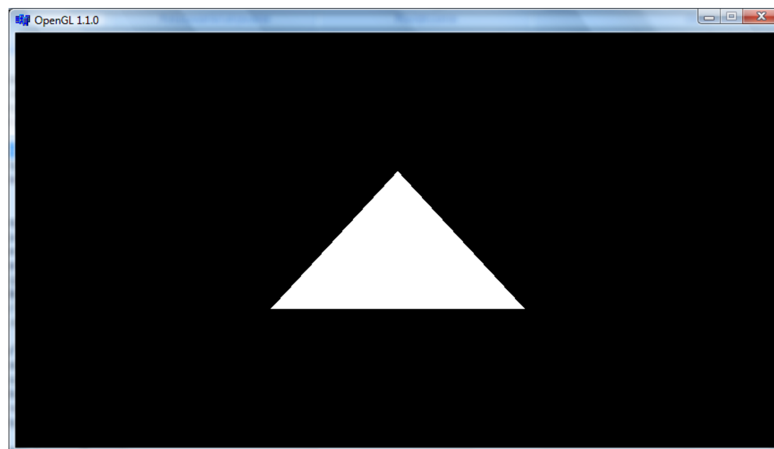
[8] Zauważmy, że nie ma znaczenia, czy przekształceniom poddana zostaje scena, czy kamera. Przesunięcie sceny w lewo jest całkowicie równoważne przesunięciu kamery w prawo — to tylko kwestia wyboru układu odniesienia.

Rysowanie trójkąta to sekwencja poleceń rozpoczynająca się od funkcji glBegin, a kończąca się wywołaniem funkcji glEnd. Argumentem funkcji glBegin jest typ figury (punkt, linia, trójkąt, czworokąt lub wielokąt oraz złożenia trójkątów). Między glBegin a glEnd powinny znaleźć się przede wszystkim wywołania funkcji glVertex, które definiują **werteksy**, czyli punkty w trójwymiarowej przestrzeni, a praktyce wierzchołki figur. Zadeklarowany w glBegin typ figury wyznacza sposób interpretacji werteksów przez bibliotekę OpenGL. Jeżeli argumentem glBegin jest GL_POINTS, to każdy werteks jest reprezentowany przez osobno rysowany punkt. Z kolei GL_LINES powoduje łączenie werteksów w pary, między którymi rysowany jest odcinek. Stała GL_LINE_STRIP powoduje rysowanie łamanej między wszystkimi zdefiniowanymi werteksami. Dla GL_TRIANGLES i GL_QUADS werteksy grupowane są odpowiednio w trójki i czwórki, a na ekranie pojawia się trójkąt lub czworokąt. Zatem definicja jednego trójkąta to sekwencja wywołań funkcji glBegin(GL_TRIANGLES), trzykrotnego glVertex i glEnd. Wywołań glVertex mogłoby być także sześć, dziewięć, dwanaście itd., a wówczas zdefiniowalibyśmy nie jeden, ale dwa, trzy, cztery i więcej trójkątów.

1.3 Poprawianie geometrii frustum

Zwróćmy uwagę, że w efekcie wykonania poleceń z poprzedniego ćwiczenia otrzymaliśmy trójkąt równoboczny (por. rysunek 3). Jednak definiowaliśmy trójkąt o podstawie 2*x0 i wysokości 2*y0, a więc o podstawie 2 i wysokości 2. Długości jego lewego i prawego boku powinny więc być równe w przybliżeniu 2.23. A ponieważ trójkąt ten leży na płaszczyźnie równoległej do ekranu, to wynika z tego, że nie powinniśmy uzyskać niczego, co wygląda na trójkąt równoboczny (rysunek 3); trójkąt powinien być lekko „wydłużony”. Co się wobec tego z nim stało? Okazuje się, że z trójkątem wszystko jest w porządku. Przyczyna deformacji obrazu jest inna: określając frustum określiliśmy wielkość najbliższego planu (ekranu) na rozciągającą się od -0.1 do 0.1 w kierunkach X i Y. Z tego wynika, że każdy plan sceny ma kształt kwadratu. A przecież okno widoczne na rysunku 3 nie ma takiego kształtu — jest rozciągnięty w poziomie. I to rozciągnięcie spowodowało zniekształcenie naszego trójkąta. Proporcje okna (a w trybie pełnoekranowym proporcje ekranu) należy uwzględnić przy definiowaniu frustum (listing 9).

Rysunek 3.
Pierwszy trójkąt

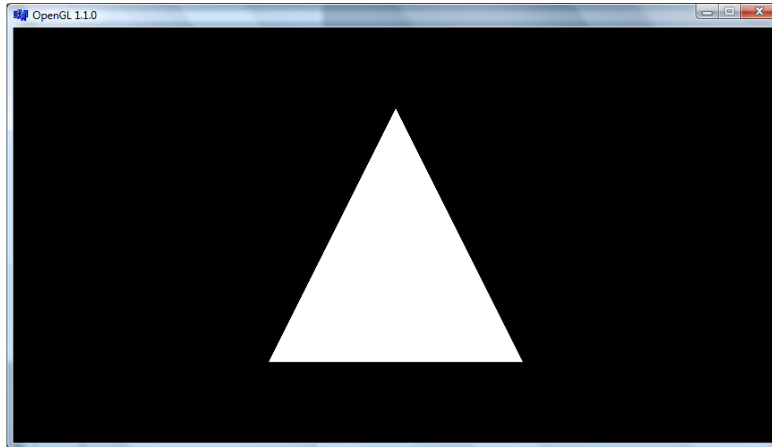


Listing 9. Definiując frustum, uwzględniamy proporcję okna

```
void TForm1::GL_UstawienieSceny()
{
   glViewport(0,0,ClientWidth,ClientHeight); //okno OpenGL = wnetrze formy
   (domyslnie)

   //ustawienie punktu projekcji
   glMatrixMode(GL_PROJECTION); //macierz projekcji
   glLoadIdentity();
   //left,right,bottom,top,znear,zfar (clipping)
   float wsp=ClientHeight/(float)ClientWidth;
   glFrustum(-0.1, 0.1, wsp*-0.1, wsp*0.1, 0.3, 100.0); //mnozenie macierzy przez
   macierz perspektywy - ustalanie piramidy frustum
   glMatrixMode(GL_MODELVIEW); //powrot do macierzy widoku modelu
   glEnable(GL_DEPTH_TEST); //z-buffer aktywny = ukrywanie niewidocznych trojkatow !!!
}
```

Na rysunku 4 widoczny jest trójkąt po skorygowaniu proporcji frustum.

Rysunek 4.
Trójkąt — tym razem wiernie odwzorowany na ekranie



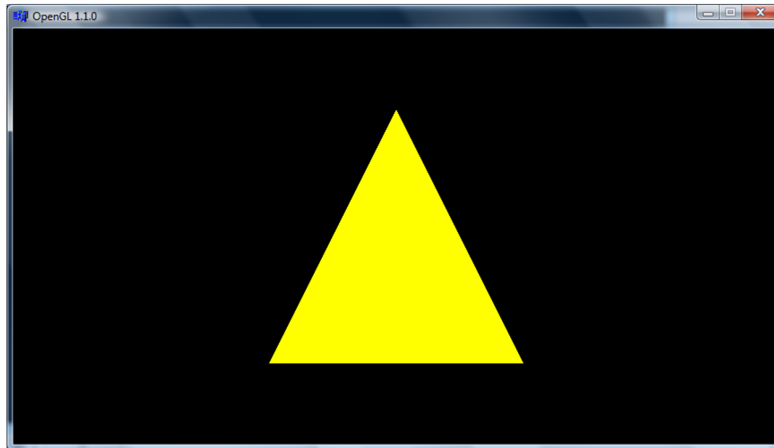
1.4 Kolor

Do powyższej funkcji dodajemy wywołanie funkcji `glColor`, a dokładnie jej wersji `glColor3ub` z argumentami `255`, `255` i `0`, które opisują odpowiednio czerwoną, zieloną i niebieską składową koloru (listing 10). W efekcie płaszczyzna trójkąta stanie się żółta, co dobrze byłoby widać na rysunku 5, gdyby książka miała kolorowe ilustracje.

Listing 10. Czarno-biały świat to żadna zabawa

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //Rysowanie trojkata
   glBegin(GL_TRIANGLES);
   //ustalanie trzech wierzcholkow trojkata (werteksow (x,y,z))
   //(0,0,z) jest mniej wiecej w srodku ekranu
   glColor3ub(255,255,0); //zolty
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, y0, z0); //gorny
   //koniec rysowania figury
   glEnd();

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```

Rysunek 5.
Zmiana koloru trójkąta



Zagadnienie kolorów to znacznie poważniejsza sprawa niż wynika z powyższego listingu. Jest ona w OpenGL tak ściśle związana z oświetleniem, że na zabawę kolorami musimy poczekać, aż włączymy mechanizm oświetlenia.

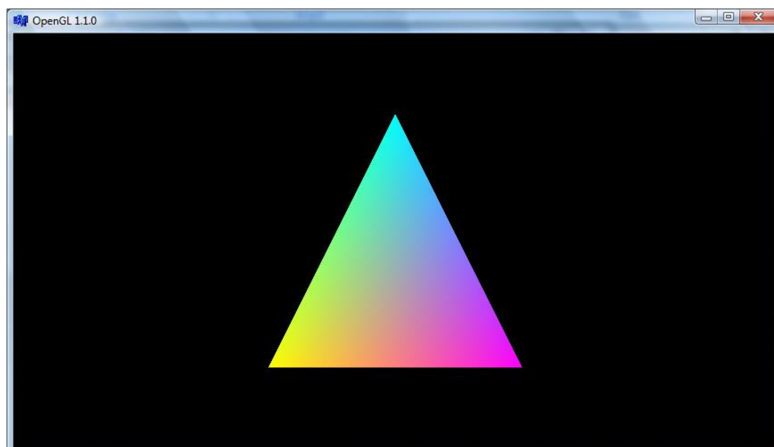
Na potrzeby zgodności z VCL wygodnie zdefiniować sobie funkcję pomocniczą:

```
GL_Kolor(TColor kolor)
{
   glColor3f(GetRValue(kolor)/255.0f,GetGValue(kolor)/255.0f,GetBValue(kolor)/255.0f);
}
```

1.5 Cieniowanie kolorów na powierzchniach

Z trójkątem nie musi być związany tylko jeden kolor. Osobny kolor można związać z każdym jego wierzchołkiem. Wówczas zastosowana zostanie płynna zmiana koloru pomiędzy wierzchołkami (cieniowanie). W niektórych przypadkach możemy cieniowania użyć do imitowania oświetlenia (por. rysunek 6).

Rysunek 6.
Cieniowanie kolorów może imitować oświetlenie



1. Modyfikujemy fragment metody `RysujScene` odpowiedzialny za rysowanie trójkąta zgodnie z wyróżnieniami na listingu 11.

Listing 11. Z każdym werteksem wiążemy inny kolor

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10
```

```
    //Rysowanie trojkata
    glBegin(GL_TRIANGLES);
    //ustalanie trzech wierzcholkow trojkata (werteksow (x,y,z))
    //(0,0,z) jest mniej wiecej w srodku ekranu
    glColor3ub(255,255,0); //zolty
    glVertex3f(-x0, -y0, z0); //dolny lewy
    glColor3ub(255,0,255); //fuksja
    glVertex3f(x0, -y0, z0); //dolny prawy
    glColor3ub(0,255,255); //blekit
    glVertex3f(0, y0, z0); //gorny
    //koniec rysowania figury
    glEnd();

    //Z bufora na ekran
    glFlush();
    SwapBuffers(uchwytDC);
}
```

4. Kompilujemy aplikację i uruchamiamy ją, a następnie podziwiamy uzyskany efekt (zob. rysunek 6).
5. Po tym usuwamy lub komentujemy dodatkowe linie, żeby cieniowanie nie interferowało z oświetleniem figury, które dodamy za chwilę.

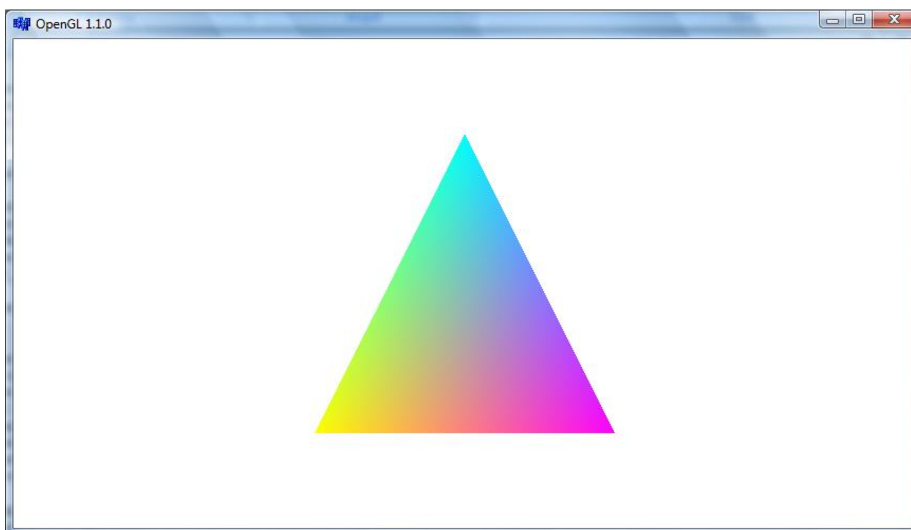
<< koniec ćwiczenia >>

Cieniowanie można też wyłączyć poleceniem `glShadeModel(GL_FLAT);`. Wówczas do kolorowania trójkąta używany jest kolor ostatniego wierzchołka. Domyślna wartość włączająca cieniowanie to `GL_SMOOTH`.

1.6 Kolor tła

Zwykle aplikacje OpenGL wyposażone są w czarne tło, na którym dobrze widać oświetlone przedmioty. Nie jest to jednak obowiązkowe. Tło może mieć dowolny kolor. Aby to sprawdzić modyfikujemy metodę `RysujScene` dodając polecenie wyróżnione na poniższym listingu 12.

Listing 12.
```
void __fastcall TGLForm::GL_RysujScene()
{
    //Przygotowanie bufora
    //glClearColor(1.0,1.0,1.0,0.0); //biale tlo
    glClearColor(0.0,0.0,0.1,0.0); //ciemnogranatowe tlo
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity(); //macierz model-widok = macierz jednostkowa

//dalsza czesc metody
```



Rysunek 7. Forma OpenGL z białym tłem

1.7 Rysowanie figury przestrzennej (ostrosłupa)

Jak wspominałem, między funkcjami `glBegin(GL_TRIANGLES)` i `glEnd` można umieścić nie tylko trzy definicję werteksów, ale dowolną wielokrotność tej liczby. Dodajmy wobec tego jeszcze trzy trójki, które określą trzy dodatkowe trójkąty (listing 13). Całość powinna utworzyć ostrosłup. Każda ściana ostrosłupa narysowana będzie w innym kolorze:

Listing 13. Rysowany wcześniej trójkąt też był figurą przestrzenną (umieszczoną w przestrzeni trójwymiarowej), tyle że… płaską

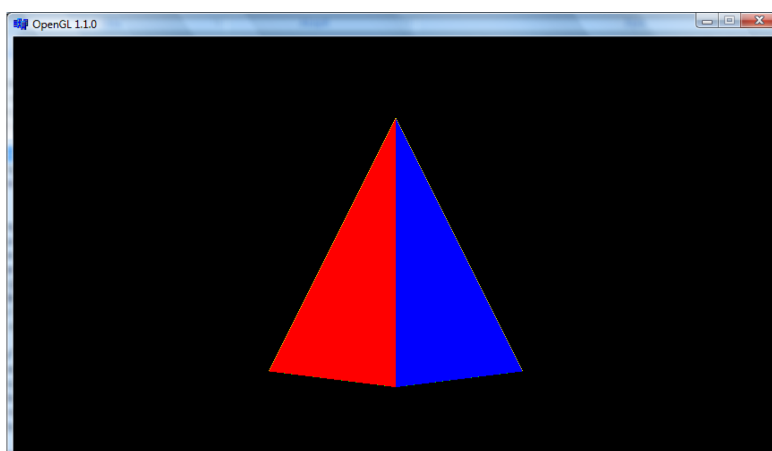
```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   //glClearColor(1.0,1.0,1.0,0.0); //biale tlo
   glClearColor(0.0,0.0,0.1,0.0); //ciemnogranatowe tlo
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //Rysowanie trojkata
   glBegin(GL_TRIANGLES);
   //ustalanie trzech wierzcholkow trojkata (werteksow (x,y,z))
   //(0,0,z) jest mniej wiecej w srodku ekranu

   //tylna
   glColor3ub(255,255,0); //zolty
   glVertex3f( -x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, y0, z0); //gorny

   //podstawa
   glColor3ub(0,255,0); //zielony
   glVertex3f( -x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, -y0, 2*z0); //dolny przedni

   //lewa
   glColor3ub(255,0,0); //czerwony
   glVertex3f( -x0, -y0, z0); //dolny lewy
   glVertex3f(0, -y0, 2*z0); //dolny przedni
   glVertex3f(0, y0, z0); //gorny

   //prawa
   glColor3ub(0,0,255); //niebieski
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, -y0, 2*z0); //dolny przedni
   glVertex3f(0, y0, z0); //gorny

   //koniec rysowania figury
   glEnd();

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```

Zamiast rysować cztery trójkąty (12 wierzchołków do przekształcania przez macierze model-widok i rzutowania), można było posłużyć się wstęgami lub wachlarzami trójkątów (jako argumentów funkcji `glBegin` należy wówczas użyć zamiast `GL_TRIANGLES` odpowiednio `GL_TRIANGLE_STRIP` lub `GL_TRIANGLE_FAN`). Wówczas kolejne trójkąty wyznaczane są nie przez nową trójkę werteksów, a przez jeden dodatkowy werteks i jeden z boków poprzednio zdefiniowanego trójkąta. W ten sposób zmniejsza się ilość wierzchołków, co w przypadku dużych figur zbudowanych z wielu werteksów może znacząco przyspieszyć ich przekształcanie w potoku.

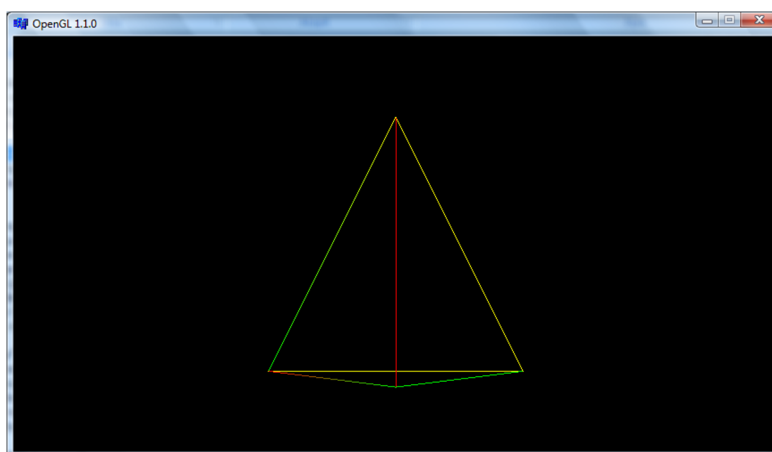
Natomiast aby zamiast „pełnego” ostrosłupa (rysunek 8) rysować jedynie jego szkielet, należy w `glBegin` użyć argumentu `GL_LINE_LOOP` (zob. rysunek 9).

Kolejną ważną sprawą związaną z rysowaniem figur płaskich jest tzw. **nawinięcie**. Kolejność werteksów wyznacza bowiem przód i tył figury. Jeżeli staniemy „za” figurą (tj. patrzymy w kierunku jej przodu), to wierzchołki powinny być zdefiniowane zgodnie z kierunkiem ruchu wskazówek zegara. Tym samym jeżeli patrzymy na trójkąt narysowany na ekranie i jego wierzchołki nawinięte są w kierunku przeciwnym do kierunku ruchu wskazówek zegara, to trójkąt skierowany jest przodem do nas. Dlaczego jest to ważne? Ponieważ można uniknąć rysowania figur, które skierowane są np. tyłem do kamery. Bardzo ogranicza to wysiłek włożony przez kartę graficzną w przygotowanie obrazu. Ale czy sprawdzanie, czy figura ułożona jest przodem, czy tyłem do kamery, nie jest kosztowniejsze niż rysowanie obu jej stron? Owszem, ale wyobraźmy sobie zamkniętą figurę przestrzenną, np. nasz ostrosłup. Jeżeli jego ściany będą tak nawinięte, że wszystkie będą skierowane przodem na zewnątrz, to możemy mieć pewność, że rysowanie „tyłów” nie będzie potrzebne. Są jednak od tej sytuacji wyjątki. Możemy na przykład dopuszczać sytuację, w której kamera wnika do wnętrza figury. Wówczas, jeżeli chcemy widzieć jej wnętrze, a nie ma po prostu znikać, musimy rysować figury z obu stron. Inna możliwość, i to jest właśnie nasz przypadek, jest taka, że zamierzamy uczynić jedną lub kilka ścian częściowo przezroczystych. A przez takie okno będzie można oczywiście również zajrzeć do wnętrza figury, a więc wówczas także konieczne jest rysowanie figur z obu stron.

Rysunek 8. *Każdą figurę przestrzenną można zbudować z trójkątów*



Rysunek 9. *Wszystkie możliwe argumenty funkcji glBegin znajdzie Czytelnik w dokumentacji MSDN dołączonej do C++Buildera*



1.8 Wyodrębnienie metody rysującej figurę

Wygodnie jest umieścić zbiór poleceń rysujących figurę w osobnej metodzie lub funkcji. Umożliwi to wielokrotne użycie tych funkcji i w ten sposób łatwe powielanie figury (por. projekt 204.).

1. Definiujemy metodę `RysujOstroslup` i umieszczamy w niej polecenia od `glBegin` do `glEnd z metody RysujScene` (listing 14).

Listing 14. Metoda definiująca wierzchołki ostrosłupa

```
void __fastcall TForm1::RysujOstroslup(float x0,float y0,float z0) const
```

```
{
   //Rysowanie trojkata
   glBegin(GL_TRIANGLES);
   //ustalanie trzech wierzcholkow trojkata (werteksow (x,y,z))
   //(0,0,z) jest mniej wiecej w srodku ekranu

   //tylna
   glColor3ub(255,255,0); //zolty
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, y0, z0); //gorny

   //podstawa
   glColor3ub(0,255,0); //zielony
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, -y0, 2*z0); //dolny przedni

   //lewa
   glColor3ub(255,0,0); //czerwony
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(0, -y0, 2*z0); //dolny przedni
   glVertex3f(0, y0, z0); //gorny

   //prawa
   glColor3ub(0,0,255); //niebieski
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, -y0, 2*z0); //dolny przedni
   glVertex3f(0, y0, z0); //gorny

   //koniec rysowania figury
   glEnd();
}
```

2. Do definicji klasy dodajemy odpowiednią deklarację nowej metody.
3. Następnie modyfikujemy metodę `RysujScene`, zastępując w niej sekwencję poleceń od `glBegin` do `glEnd` przez wywołanie `RysujOstroslup` (listing 15).

Listing 15. Taka zmiana zdecydowanie zwiększa czytelność metody renderującej

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   //glClearColor(1.0,1.0,1.0,0.0); //biale tlo
   glClearColor(0.0,0.0,0.1,0.0); //ciemnogranatowe tlo
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //rysowanie ostroslupa
   RysujOstroslup(x0,y0,z0);

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```

2. Projekt klasy TGLForm implementującej formę przystosowaną do wyświetlania grafiki OpenGL

Dla własnej wygody dobrze jest zgromadzić kod niezbędny do uruchomienia aplikacji korzystającej z OpenGL w osobnej klasie.

2.1 Definiowanie klasy TGLForm

1. W C++Builderze wybierany polecenie z menu *File*, *New*, *Unit* tworząc w ten sposób parę plików *.cpp* i *.h* (tzw. moduł).
2. Po utworzeniu nowego modułu wciskamy kombinację klawiszy *Ctrl+Shift+S* i zapisujemy nowe pliki pod nazwami *GLForm.cpp*/*GLForm.h*.
3. W pliku *GLForm.h* definiujemy klasę `TGLForm` dziedziczącą z `TForm` i umieszczamy w niej deklaracje pól i metod zdefiniowanych uprzednio w `TForm1` (listing 16).

Listing 16. Pełna zawartość pliku nagłówkowego GLForm.h

```
//---------------------------------------------------------------------------

#ifndef GLFormH
#define GLFormH
//---------------------------------------------------------------------------

#include <Forms.hpp>
#include <gl.h>
#include <glu.h>

class TGLForm : public TForm
{
__published:    //metody zdarzeniowe
private: //prywatne pola i metody zwiazane z OpenGL
   HDC uchwytDC; //uchwyt do "display device context (DC)"
   HGLRC uchwytRC; //uchwyt do "OpenGL rendering context"
   bool GL_UstalFormatPikseli(HDC uchwytDC);
   void GL_UstawienieSceny();
   TColor kolorTla;
protected:
   virtual void __fastcall GL_RysujScene();
   virtual void __fastcall RysujScene() = 0; //abstrakcyjna, musi byc zdefiniowana w
klasie potomnej
   virtual void __fastcall Pomoc(); //mozna zastapic te metoda inna
public:         // User declarations
   __fastcall TGLForm(TComponent* Owner);
   __fastcall ~TGLForm();
protected:
   void __fastcall FormPaint(TObject *Sender);
};

#endif
```

4. Do pliku *GLForm.cpp* przenosimy metody `GL_UstalFormatPikseli` i `GL_UstawienieSceny`. Należy pamiętać, aby w ich definicji zmienić w sygnaturze nazwę klasy na `TGLForm` np. `void TGLForm::GL_UstawienieSceny()`.
5. W nowej klasie zdefiniowana jest metoda `GL_RysujScene`, do której przenosimy z metody `TForm1::RysujScene` polecenia związane z inicjowaniem macierzy model-widok i zmianą bufora (listing 17). Wywołujemy z niej także metodę `RysujScene`.

Listing 17.

```
void __fastcall TGLForm::GL_RysujScene()
{
   //Przygotowanie bufora
   glClearColor(GetRValue(kolorTla)/255.0f, GetGValue(kolorTla)/255.0f,
GetBValue(kolorTla)/255.0f,0.0);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //przeksztalcenia macierzy model-widok i rysowanie figur
   RysujScene();

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```

6. Do konstruktora i destruktora nowej klasy przenosimy zawartość metod `TForm1::FormCreate` i `TForm1::FormClose`. Następnie za pomocą inspektora obiektów odłączamy metody `FormCreate` i `FormClose` zdarzeń formy `TForm1` i usuwamy je z klasy `TForm1`. Pamiętajmy o wywołaniu w konstruktorze wywołania klasy bazowej tj. `TForm` (listing 18)
   Listing 18.
```
#include <Dialogs.hpp>

__fastcall TGLForm::TGLForm(TComponent* Owner):TForm(Owner),kolorTla(clBlack)
{
   //biezace okno staje sie oknem OpenGL
   uchwytDC=GetDC(Handle);
   if (!GL_UstalFormatPikseli(uchwytDC)) ShowMessage("Nie udało się ustalić formatu pikseli");
   uchwytRC=wglCreateContext(uchwytDC);
   if (uchwytRC==NULL) ShowMessage("Nie udało się pobrać uchwytu kontekstu grafiki");
   if (!wglMakeCurrent(uchwytDC,uchwytRC)) ShowMessage("Inicjacja grafiki OpenGL nie powiodła się");
   GL_UstawienieSceny();
   Caption=(AnsiString)"OpenGL "+(char*)glGetString(GL_VERSION);
}

__fastcall TGLForm::~TGLForm()
{
   wglMakeCurrent(NULL,NULL);
   wglDeleteContext(uchwytRC);
   ReleaseDC(Handle,uchwytDC);
   PostQuitMessage(0);
}
```
7. W ramach tymczasowego rozwiązania przygotujemy metodę zdarzeniową związaną ze zdarzeniem `TGLForm::OnPaint`:
   a. W klasie `TGLForm` definiujemy metodę FormPaint o sygnaturze pasującej do typy zdarzenia `OnPaint` (tj. metoda o sygnaturze `void __fastcall TGLForm::FormPaint(TObject *Sender)`).
   b. Do konstruktora dodajemy polecenie przypisujące metodę `FormPaint` do własności `OnPaint` (zmienna o typie wskaźnika do metody) tj. `OnPaint=FormPaint;`.
8. Zdefiniujmy zadeklarowaną w klasie `TGLForm` metodę `Pomoc` (listing 19):
   Listing 19
```
void __fastcall TGLForm::Pomoc()
{
  AnsiString s=(AnsiString)"(c) Jacek Matulewski 2006-2007\n"+
                 "OpenGL, wersja "+(char*)glGetString(GL_VERSION)+"\n"+
                 "GLU, wersja "+(char*)gluGetString(GLU_VERSION);
  MessageBox(Handle,s.c_str(),"TGLForm",MB_OK | MB_ICONINFORMATION);
}
```
9. Końcowy etap to zmiana klasy bazowej klasy `TForm1` (czyli formy aplikacji) z `TForm` na `TGLForm`:
   a. Należy przede wszystkim do pliku *Unit1.h* dodać dyrektywę `#include "GLForm.h"`,
   b. Następnie pierwszą linię definicji klasy `TForm1` zmieniamy na: `class TForm1 : public TGLForm`,
   c. W konstruktorze klasy `TForm1` należy zmienić wywołanie klasy bazowej tj. `__fastcall TForm1::TForm1(TComponent* Owner):TGLForm(Owner)`
   d. sprawdzamy, czy z klasy `TForm1` zostały usunięte wszystkie metody rozpoczynające się od `GL_` i czy w metodzie `RysujScene` zostało tylko definicja zmiennych `x0`, `y0` i `z0` oraz wywołanie metody `RysujOstroslup`.

<< koniec ćwiczenia >>

Zdaję sobie sprawę, że cała ta operacja jest dość zawiła, dlatego pod adresem http://****/szablon_inicjacja.zip można znaleźć gotowy projekt po zakończeniu tej operacji. Dodatkowo w projekcie zdefiniowana jest własność formy pozwalająca na ustalanie koloru tła.

W klasie `TGLForm` zdefiniowana jest czysta wirtualna metoda `RysujScene`. To oznacza, że cała klasa jest klasą abstrakcyjną, a zatem klasa, która z niej dziedziczy powinna zawierać tę metodę jeżeli chcemy tworzyć jej instancję. Szczęśliwie klasa `TForm1` zawiera metodę `RysujScene` zawierającą polecenia rysowania ostrosłupa i to ona jest wywoływana w przypadku wystąpienia zdarzenia `OnPaint`.

2.2 Obsługa komunikatu WM_PAINT

Wywoływanie metody GL_RysujScene korzystające z mechanizmu zdarzeń, a więc polegające na zdefiniowaniu metody zdarzeniowej FormPaint i związaniu jej w konstruktorze klasy TGLForm ze zdarzeniem OnPaint najwyraźniej działa. Proponuję jednak zastąpić je innym – opartym na monitorowaniu kolejki komunikatów. Nadpiszemy metodę WndProc, w którą wyposażona jest każda kontrolka i okno Windows, a która wywoływana jest gdy do kontrolki lub okna przychodzi komunikat. Komunikaty, które tak naprawdę „stoją” za zdarzeniami pozwolą nam później obsłużyć inne zdarzenia związane z myszą i klawiaturą, a mają tę podstawową zaletę w stosunku do mechanizmu zdarzeń, że są niezależne od biblioteki VCL, a tym samym jest to rozwiązanie w pełni przenaszalne.

1. W pliku *GLForm.h* w sekcji chronionej klasy TGLForm definiujemy metodę o sygnaturze:
```
protected:
   void __fastcall WndProc(TMessage& Message);
```
2. Następnie w pliku *GLForm.cpp* definiujemy samą funkcję, która reaguje na razie na trzy zdarzenia: WM_PAINT (polecenie odświeżenia formy), WM_SIZE (zmiana rozmiaru formy) i WM_KEYDOWN (naciśnięcie klawisza):

   Listing 20. Nasza metoda reaguje na razie na dwa klawisze: *Esc* i *F1*
```
void __fastcall TGLForm::WndProc(TMessage& Message)
{
   TForm::WndProc(Message);

   switch(Message.Msg)
   {
      case WM_PAINT: //odswiezenie formy
         GL_RysujScene();
         break;

      case WM_SIZE: //zmiana rozmiaru formy
      case WM_SIZING:
         GL_UstawienieSceny();
         GL_RysujScene();
         break;

      case WM_KEYDOWN:
         if (Message.WParam==VK_ESCAPE) Close();
         if (Message.WParam==VK_F1) Pomoc();
         break;
   }
}
```
3. Teraz należy usunąć niepotrzebną już metodę FormPaint i polecenie wiążące ją ze zdarzeniem OnPaint klasy TGLForm.

<< koniec ćwiczenia >>

Rzeczą absolutnie kluczową jest, aby z nadpisanej metody WndProc wywołać tę metodę zdefiniowaną w klasie bazowej. To w niej i w metodach zdefiniowanych w kolejnych klasach bazowych aż do TControl obsługiwane są wszystkie komunikaty. Gdybyśmy tego nie zrobili – aplikacja natychmiast stałaby się „samotną wyspą” w oceanie Windows. Warto spróbować, ale to pewnie spowoduje, że aplikacja nawet nie uruchomi się bez zgłaszania wyjątku.

3. Obroty i przesunięcia

3.1 Obroty obiektów na scenie. Ruch kamery i ruch aktorów

Czas zająć się przekształceniami macierzy model-widok. Tu pojawia się jednak zagadnienie kolejności przekształceń i poleceń rysujących nowe figury. Musimy wiedzieć jedną ważną rzecz: **to, co zostało narysowane, nie jest już przekształcane**. Zatem jeżeli chcemy obracać ostrosłupem, to funkcje glRotate muszą znaleźć się przed RysujOstroslup, bo inaczej obracane będą figury zdefiniowane potem, ale ostrosłup zostanie nieruchomy.

Ta zasada pozwala na uniezależnienie ruchu poszczególnych przedmiotów znajdujących się na scenie (aktorów), a także, a może przede wszystkim, na oddzielenie ruchu kamery od ruchu aktorów. Jak dobrze wiemy, nie ma

zasadniczej różnicy między ruchem kamery, a ruchem sceny, szczególnie jeżeli na scenie nie ma niczego, co nasz umysł mógłby identyfikować jako nieruchome (np. jakiś rodzaj podłoża). Zresztą nawet wówczas stwierdzenie, który element się porusza, a który spoczywa, jest umowne[9]. Wyobraźmy sobie jednak metodę renderującą, w której wpierw rysowana jest płaszczyzna, następnie wykonywane jest przekształcenie translacji (przesunięcia) i wówczas rysowany jest ostrosłup. Wyglądałoby to tak, jakby ostrosłup przesuwał się po nieruchomej (tj. narysowanej przed przesunięciem) powierzchni — tak w naturalny sposób scenę tę interpretowałby nasz umysł. Jeżeli dodatkowe przekształcenie umieścimy przed narysowaniem owej powierzchni, to dotyczyć one będą zarówno niej, jak i poruszającego się po niej ostrosłupa. Wyglądać to będzie, jak ruch kamery, która filmuje ruch ostrosłupa po powierzchni.

Z tego wynika, że można umownie odróżnić ruch kamery od ruchu obiektów na scenie. Podział ten zależy od momentu, w którym rysowana jest część sceny, którą uznajemy za nieruchomą (owa powierzchnia). Wcześniejsze przekształcenia to ruch kamery, późniejsze — ruchy aktorów. Ogólny schemat metody renderującej powinien być zatem zgodny z przedstawionym w tabeli 11.2.

I tak dalej. Etapy 7. i 8. mogą się oczywiście powtarzać. Aktor nie tylko może nie być nieruchomy (tzn. może poruszać się po scenie), ale może również zmieniać kształt. Do tego względnego ruchu trójkątów, z których zbudowany jest aktor, stosuje się jednak ta sama zasada, co do ruchu aktorów po scenie.

Tabela 11.2. *Kolejność czynności wykonywanych w metodzie renderującej*

Etap	Czynności
1. Inicjacja	Macierz model-widok ustalana na jednostkową
2. Przedmioty trwale związane z kamerą (np. broń bohatera widoczna w grach FPP/FPS)	Rysowanie figur
3. Ruch kamery	Przekształcenia macierzy model-widok
4. Rysowanie nieruchomej części sceny	Rysowanie figur
5. Ruch pierwszego aktora	Przekształcenia macierzy model-widok
6. Rysowanie pierwszego aktora	Rysowanie figur
7. Ruch drugiego aktora	Przekształcenia macierzy model-widok
8. Rysowanie drugiego aktora	Rysowanie figur

Aby móc obracać nasz ostrosłup za pomocą klawiszy sterowania kursorem, musimy wykonać następujące czynności:

1. W sekcji prywatnej klasy `TForm1` definiujemy dwa pola `Theta` i `Phi` typu `float`, przechowujące kąty określające położenie ostrosłupa (listing 21).

Listing 21. Definiujemy dwa nowe pola

```
class TForm1 : public TForm
{
__published:    // IDE-managed Components
   void __fastcall FormCreate(TObject *Sender);
   void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
   void __fastcall FormPaint(TObject *Sender);
private:        // User declarations
   HDC uchwytDC; //uchwyt do "display device context (DC)"
   HGLRC uchwytRC; //uchwyt do "OpenGL rendering context"
   bool GL_UstalFormatPikseli(HDC uchwytDC);
   void GL_UstawienieSceny();
   void __fastcall RysujScene();
   void __fastcall RysujOstroslup(float x0,float y0,float z0) const;
   float Phi, Theta;
public:         // User declarations
   __fastcall TForm1(TComponent* Owner);
};
```

4. W konstruktorze formy inicjujemy nowe pola zerami (listing 22).

[9] Każdy chyba pamięta gag z filmu Monty Pythona, w którym to nie pociąg odjeżdżał, ale peron z osobą machającą białą chusteczką.

Listing 22. Inicjujemy oba kąty zerami

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TGLForm(Owner),
        Phi(0.0f), Theta(0.0f)
{
}
```

5. Do metody renderującej dodajemy polecenia obracające macierz model-widok o kąty zapamiętane w polach `Phi` i `Theta` (listing 23). Pierwsze wyznacza kąt obrotu wokół osi OY, a drugie — wokół osi OX[10].

Listing 23. W każdej renderowanej scenie położenie figury wyznaczone jest przez pola Phi i Theta

```
void __fastcall TForm1::RysujScene()
{
   //obroty
   glRotatef(Phi, 0.0, 1.0, 0.0); //wokol OY
   glRotatef(Theta, 1.0, 0.0, 0.0); //wokol OX

   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //rysowanie ostroslupa
   RysujOstroslup(x0,y0,z0);
}
```

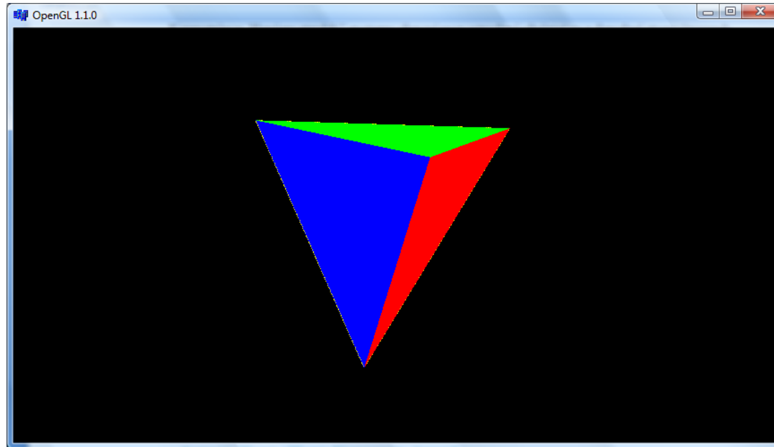
6. Zmieniamy własność `KeyPreview` formy na `true`.
7. Za pomocą inspektora tworzymy metodę związaną z `OnKeyDown` formy (listing 24). Będzie ona odgrywała rolę naszego pulpitu kontrolnego.

Listing 24. „Strzałki” pozwalają na kontrolowanie wartości kątów, o jakie obracany jest ostrosłup

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
     TShiftState Shift)
{
   switch (Key)
   {
      //obroty
      case VK_LEFT:    Phi-=3; break;
      case VK_RIGHT:   Phi+=3; break;
      case VK_UP :     Theta-=3; break;
      case VK_DOWN:    Theta+=3; break;
   }

   GL_RysujScene();
}
```

Korzystając z „klawiszy strzałek”, możemy obracać nasz ostrosłup i obejrzeć go z dowolnej strony (rysunek 10).

[10] Kolejność obrotów ma znaczenie! Tak samo jak kolejność mnożenia przez macierze. Wystarczy wziąć do ręki niniejszą książkę i obrócić ją wzdłuż grzbietu, a potem wzdłuż innego wybranego boku. Następnie wróćmy do pozycji wyjściowej i wykonajmy oba obroty w zmienionej kolejności. W obu przypadkach uzyskamy inne ułożenie książki.



Rysunek 10. *Teraz możemy przekonać się, że figura, którą zbudowaliśmy, jest rzeczywiście trójwymiarowa*

3.2 Przesunięcia obiektu

Analogicznie wygląda sprawa z przekształceniem translacji:

1. W klasie `TForm1` definiujemy jeszcze dwa pola prywatne typu `float` o nazwach `PozycjaX`, `PozycjaY` i `PozycjaZ`. W konstruktorze ustalamy ich wartości na równe 0 (listing 25).

Listing 25. Również te pola inicjujemy zerami

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TGLForm(Owner),
        Phi(0.0f), Theta(0.0f),
        PozycjaX(0.0f), PozycjaY(0.0f), PozycjaZ(0.0f)
{
}
```

2. Do metody `RysujScene` dodajemy polecenie przesuwające figurę o wektor `[PozycjaX, PozycjaY, PozycjaZ]` zgodnie ze wzorem na listingu 26:

Listing 26. Przesuwamy ostrosłup

```
void __fastcall TForm1::RysujScene()
{
   //obroty
   glRotatef(Phi, 0.0, 1.0, 0.0); //wokol OY
   glRotatef(Theta, 1.0, 0.0, 0.0); //wokol OX

   //przesuniecia
   glTranslatef(PozycjaX,PozycjaY,PozycjaZ);

   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //rysowanie ostroslupa
   RysujOstroslup(x0,y0,z0);
}
```

8. Rozbudowujemy metodę `FormKeyDown`, dodając do niej fragmenty wyróżnione na listingu 27. Dzięki temu będziemy mogli kontrolować przesunięcia ostrosłupa klawiszami sterowania kursorem przy jednoczesnym naciśnięciu klawisza *Shift* lub *Ctrl*.

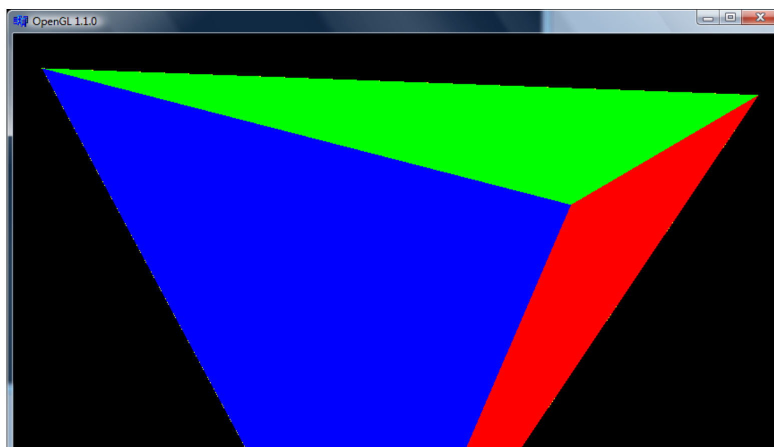
Listing 27. Zwiększamy ilość przycisków na pulpicie kontrolnym

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
   //obroty
   if (Shift.Empty())
   {
      switch (Key)
      {
```

```
            case VK_LEFT:   Phi-=3; break;
            case VK_RIGHT:  Phi+=3; break;
            case VK_UP :    Theta-=3; break;
            case VK_DOWN:   Theta+=3; break;
        }
    }

    //przesuniecia w pionie i poziomie
    if (Shift.Contains(ssCtrl))
    {
        switch (Key)
        {
                case VK_LEFT:  PozycjaX-=0.1; break;
                case VK_RIGHT: PozycjaX+=0.1; break;
                case VK_UP:    PozycjaY+=0.1; break;
                case VK_DOWN:  PozycjaY-=0.1; break;
        }
    }

    //przesuniecia w poziomie i przod-tyl
    if (Shift.Contains(ssShift))
    {
        switch (Key)
        {
            case VK_LEFT:  PozycjaX-=0.1; break;
            case VK_RIGHT: PozycjaX+=0.1; break;
            case VK_UP:    PozycjaZ-=0.1; break;
            case VK_DOWN:  PozycjaZ+=0.1; break;
        }
    }

    GL_RysujScene();
}
```

Po wykonaniu powyższych czynności możemy za pomocą „klawiszy strzałek” oraz klawiszy *Shift* i *Ctrl* w pełni kontrolować pozycję ostrosłupa (rysunek 11). Dopóki na scenie jest tylko jeden obiekt, nie ma znaczenia, czy nazwiemy to ruchem kamery, czy aktora. Nasz wzrok i tak nie ma żadnego „nieruchomego” układu odniesienia. To tak, jakbyśmy krążyli wokół innej osoby w pustej przestrzeni kosmicznej. Pytanie o to, kto wiruje wokół kogo, nie ma wówczas żadnego sensu. Lub mniej abstrakcyjny przykład — dwa pociągi stojące na sąsiednich torach. Gdy jeden z nich rusza (i nie widzimy nieruchomego podłoża) nie możemy stwierdzić który z nich jedzie, a który stoi. Sytuacja zmienia się całkowicie, gdy na scenie umieścimy coś, co nasz umysł zidentyfikuje jako nieruchome podłoże. Jednak, jak już podkreślałem wcześniej, z punktu widzenia OpenGL przekształcenia w punkcie 3. przedstawionego wyżej schematu niczym nie różnią się od przekształceń z punktów 5. i 7.

Rysunek 11. *Teraz naszemu ostrosłupowi możemy przyjrzeć się z bliska*



3.3 Prosta animacja

Animacja to po prostu przesunięcia i obroty wykonywane zgodnie z pewnym czasowym schematem. W C++Builderze ów schemat możemy najprościej wyznaczyć za pomocą komponentu `TTimer`. Wystarczy, że co określony czas będzie on uruchamiał metodę zdarzeniową, w której zmieniane będą pola przechowujące

wartości kątów i pozycji ostrosłupa, po czym wywoływał metodę GL_RysujScene. Identycznie jak przy naciskaniu klawiszy sterowania kursorem, z tym że teraz owe klawisze „naciska” sam komputer.

1. Umieszczamy na formie komponent TTimer z zakładki *System*.
2. Tworzymy metodę zdarzeniową do OnTimer (listing 28):

Listing 28. Aby ruch był widoczny, konieczne jest oczywiście wywołanie metody RysujScene

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
   Phi+=1;
   Theta+=0.1;
   GL_RysujScene();
}
```

3. Zmieniamy własność Interval komponentu Timer1 na 10.
4. Do metody FormKeyDown dodajemy możliwość włączenia i wyłączenia animacji za pomocą klawisza *Q* (listing 29):

Listing 29. Nowy przycisk na panelu kontrolnym

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
     TShiftState Shift)
{
   //obroty
   if (Shift.Empty())
   {
      switch (Key)
      {
         case VK_LEFT:   Phi-=3; break;
         case VK_RIGHT:  Phi+=3; break;
         case VK_UP :    Theta-=3; break;
         case VK_DOWN:   Theta+=3; break;

         case 'q':
         case 'Q':
            Timer1->Enabled=!Timer1->Enabled;
            break;
      }
   }

//ciag dalszy metody
```

Dzięki temu w trakcie animacji metoda GL_RysujScene nie tylko jest wywoływana w reakcji na komunikat WM_PAINT, co zachodzi stosunkowo rzadko, ale przede wszystkim cyklicznie co 10 milisekund (sto razy na sekundę). To z pewnością wystarczająco często, bo płynna animacja wymaga około 50 klatek na sekundę (a w najgorszym wypadku przynajmniej 25). To może nawet zbyt wiele, bo monitory CRT (kineskopowe) rzadko pracują z częstością większą niż 80 Hz (80 „ekranów” na sekundę). Częstość odświeżania monitorów LCD, z natury wolniejszych, wynosi zazwyczaj 60 Hz.

Jeżeli nawet nieco przeholujemy z częstością renderowania sceny, to nie musimy się martwić o obciążenie głównego procesora komputera — rysowaniem grafiki OpenGL zajmuje się przecież procesor karty graficznej[11].

3.4 Bardziej precyzyjne ustawianie kamery

Jak pamiętamy renderowanie rozpoczynamy od przesunięcia całej sceny o -10 w kierunku osi OZ (zob. metoda TGLForm::GL_RysujScene), a więc o 10 w głąb frustum. Zgodnie ze schematem przedstawionym w tabeli 11.2 przekształcenie to znajduje się w punkcie 3., a więc powinniśmy mówić raczej o odsunięciu kamery od sceny. Realizację tego schematu w tej chwili w metodach TGLForm::GL_RysujScene i TForm1::RysujScene przedstawia tabela 3.

[11] Oczywiście, jeżeli nie mamy jakiejś archaicznej karty graficznej nie obsługującej standardu OpenGL i korzystamy z programowej emulacji.

Tabela 3. *Schemat etapów w naszej metodach GL_RysujScene i RysujScene*

Etap	Czynności
1. Inicjacja	Macierz model-widok ustalana na jednostkową
2. Przedmioty trwale związane z kamerą	Brak
3. Ruch kamery	Odsunięcie kamery o 10 jednostek w kierunku osi OZ
4. Rysowanie nieruchomej części sceny	Rysowanie białych osi współrzędnych
5. Ruch pierwszego aktora	Obroty i przesunięcia kontrolowane z klawiatury
6. Rysowanie pierwszego aktora	Rysowanie ostrosłupa (oraz jego klonów)
7. Ruch drugiego aktora	Brak
8. Rysowanie drugiego aktora	Rysowanie zielonych osi współrzędnych

Czynność ustawienia kamery (można sobie wyobrazić, że etap 3. obejmuje znacznie więcej przekształceń niż tylko nasze przesunięcie) można znacznie uprościć, korzystając z funkcji `gluLookAt`. Jej argumentami są trzy trójki współrzędnych. Pierwsza ustala położenie kamery. W naszym przypadku będzie to punkt `(0,0,10)`. Argumenty od czwartego do szóstego wskazują punkt, na który kamera jest skierowana. My wybraliśmy środek układu współrzędnych, czyli miejsce, w którym znajduje się nasza figura (pamiętajmy, że kamera została odsunięta na pozycję `(0,0,10)`). Ostatnia trójka argumentów określa sposób, w jaki ustawiona jest kamera, wskazując jej kierunek „do góry”. My ustawiliśmy ją w kierunku osi OY, czyli tak, jak trzymalibyśmy prawdziwą kamerę, gdyby podłoże znajdowało się na płaszczyźnie OXZ (por. rysunek 2). Odpowiednie wywołanie funkcji `gluLookAt` pokazuje listing 30.

Funkcja `gluLookAt` jest zdefiniowana w bibliotece GLU (*OpenGL Utility Library*) — bibliotece zawierającej funkcje wyższego poziomu (krzywe, figury przestrzenne itp.). Więcej powiemy o niej w podrozdziale znajdującym się na końcu rozdziału. Do jej użycia potrzebny jest import nagłówka *glu.h*.

Listing 30. W metodzie TGLForm::GL_RysujScene zastąpiliśmy polecenie odsuwające scenę o –10 poleceniem precyzyjniej ustalającym położenie kamery

```
void __fastcall TGLForm::GL_RysujScene()
{
   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10
   gluLookAt(0,0,10,  //polozenie kamery
             0,0,0,   //punkt, na ktory skierowana jest kamera
             0,1,0);  //kierunek "do gory" kamery (polaryzacja)


   //przeksztalcenia macierzy model-widok i rysowanie figur
   RysujScene();

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```

4. Kontrola położenia kamery za pomocą myszy

Obracamy i przesuwamy naszego aktora (ostrosłup) klawiszami kierunkowymi. Teraz zajmiemy się kontrolą kamery. Pamiętajmy jednak, że różnica między translacjami i obrotami kamery i aktorów jest czysto umowna i zależy od tego, co na scenie uznamy za nieruchome.
Kamera kontrolowana będzie myszą. Lewy klawisz odpowiadał będzie za obroty, prawy za przesunięcia, a rolką będziemy zbliżać i oddalać kamerę od środka układu współrzędnych. To rozwiązanie umieścimy w klasie `TGLForm`.

4.1 Ruch kamery kontrolowany myszą (lewy przycisk myszy)

Kontrola pozycji kamery za pomocą myszki to standardowy element aplikacji korzystających z grafiki 3D. Możliwe jest kilka podejść do tego zagadnienia — ja wybiorę może nie do końca satysfakcjonujące, ale zdecydowanie najprostsze. Co więcej, będę niekonsekwentny. W poprzednim paragrafie przedstawiłem bowiem funkcję `gluLookAt`, która służy do łatwego ustawiania kamery. Natomiast do jej obracania zastosuję poznane wcześniej funkcje `glRotate`. Tak jest po prostu wygodniej[12].

Ruch kamery realizowany będzie w oparciu o obsługę komunikatów `WM_LBUTTONDOWN`, `WM_MOUSEMOVE` i `WM_LBUTTONUP`. ***************

1. Wpierw jednak wyłączamy animację, zmieniając własność `Timer1->Enabled` na `false`.

9. Do klasy `TGLForm` dodajemy trzy pola prywatne określające położenie kamery we współrzędnych sferycznych `KameraPhi`, `KameraTheta`, `KameraR`, dwa dodatkowe `tmpKameraPhi`, `tmpKameraTheta`, których zastosowanie zaraz się wyjaśni, oraz dwa pola `X0` i `Y0`, które umożliwiają mierzenie przesunięcia myszki z przyciśniętym przyciskiem (listing 31).

Listing 31. Definiujemy potrzebne pola

```
class TGLForm : public TForm
{
__published:
private:      //prywatne pola i metody zwiazane z OpenGL
   HDC uchwytDC; //uchwyt do "display device context (DC)"
   HGLRC uchwytRC; //uchwyt do "OpenGL rendering context"
   bool GL_UstalFormatPikseli(HDC uchwytDC);
   void GL_UstawienieSceny();
   void __fastcall WndProc(TMessage& Message);
   //kamera
   int X0,Y0;
   float KameraPhi, KameraTheta, KameraR;
   float tmpKameraPhi, tmpKameraTheta;
protected:
   virtual void __fastcall GL_RysujScene();
   virtual void __fastcall RysujScene() = 0; //abstrakcyjna, musi byc zdefiniowana w
klasie potomnej
   virtual void __fastcall Pomoc(); //mozna zastapic te metoda inna
public:
   __fastcall TGLForm(TComponent* Owner);
   __fastcall ~TGLForm();
};
```

10. Inicjujemy je, dodając do konstruktora instrukcje wyróżnione w listingu 32.

Listing 32. Wartość pola KameraR (odległość kamery) kontrolować będziemy rolką myszki

```
__fastcall TGLForm::TGLForm(TComponent* Owner)
   :TForm(Owner),
   X0(0),Y0(0),
   KameraR(10.0f),KameraPhi(0.0f),KameraTheta(0.0f)
{
   //biezace okno staje sie oknem OpenGL
   uchwytDC=GetDC(Handle);
   if (!GL_UstalFormatPikseli(uchwytDC)) ShowMessage("Nie udało się ustalić formatu
pikseli");
   uchwytRC=wglCreateContext(uchwytDC);
   if (uchwytRC==NULL) ShowMessage("Nie udało się pobrać uchwytu kontekstu grafiki");
   if (!wglMakeCurrent(uchwytDC,uchwytRC)) ShowMessage("Inicjacja grafiki OpenGL nie
powiodła się");
   GL_UstawienieSceny();
   Caption=(AnsiString)"OpenGL "+(char*)glGetString(GL_VERSION);
}
```

[12] Nie tylko wygodniej, ale również bezpieczniej. Korzystanie z `glLookAt` wymagałoby obliczania transformacji współrzędnych punktu, w którym znajduje się kamera (właśnie to robią funkcje `glRotate`). Naprawdę łatwo wówczas o błąd. A jedenaste przykazanie programisty mówi: nie pisz kodu, który ktoś napisał (i sprawdził!) za ciebie. A więc to nie tylko wygoda, ale i unikanie błędów.

11. W metodzie `GL_RysujScene` modyfikujemy argument funkcji `gluLookAt` odpowiadający za odległość kamery od środka układu współrzędnych oraz dodajemy polecenia obrotu (listing 33).

Listing 33. Dodane czynności należą oczywiście do etapu 3., przedstawionego w tabeli 3 schematu

```
void __fastcall TGLForm::GL_RysujScene()
{
   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   gluLookAt(0,0,KameraR,  //polozenie kamery
             0,0,0,        //punkt, na ktory skierowana jest kamera
             0,1,0);       //kierunek "do gory" kamery (polaryzacja)
   glRotatef(KameraPhi+tmpKameraPhi, 0.0, 1.0, 0.0); //wokol OY
   glRotatef(KameraTheta+tmpKameraTheta, 1.0, 0.0, 0.0); //wokol OX

   //przekształcenia macierzy model-widok i rysowanie figur
   RysujScene();

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```

12. Teraz do instrukcji `switch` w metodzie `WndProc` musimy dodać sekcje odpowiedzialne za obsługę komunikatów związanych z ruchem myszy i jej przyciskami:

Listing 34. Obroty kamerą ******

```
void __fastcall TGLForm::WndProc(TMessage& Message)
{
   TForm::WndProc(Message);

   switch(Message.Msg)
   {
      case WM_PAINT: //odswiezenie formy
         GL_RysujScene();
         break;

      case WM_SIZE: //zmiana rozmiaru formy
      case WM_SIZING:
         GL_UstawienieSceny();
         GL_RysujScene();
         break;

      case WM_KEYDOWN:
         if (Message.WParam==VK_ESCAPE) Close();
         if (Message.WParam==VK_F1) Pomoc();
         break;

      //KONTROLA POLOZENIA KAMERY MYSZA
      case WM_LBUTTONDOWN:
         X0=LOWORD(Message.LParam); //biezaca poz. x kursora
         Y0=HIWORD(Message.LParam); //biezaca poz. y kursora
         break;
      case WM_MOUSEMOVE:
      {
         int X=LOWORD(Message.LParam); //int X=Message.LParam & 0x0000FFFF;
         int Y=HIWORD(Message.LParam); //int Y=(Message.LParam & 0xFFFF0000) >> 16;
         if (X>Screen->Width) X-=0xFFFF;
         if (Y>Screen->Height) Y-=0xFFFF;
         int dX=X-X0;
         int dY=Y-Y0;
         if (Message.WParam==MK_LBUTTON) //wylacznie lewy przycisk myszy
         {
            const float czuloscMyszy=5.0f;
            tmpKameraPhi=dX/czuloscMyszy;
            tmpKameraTheta=dY/czuloscMyszy;//*cos((KameraPhi+tmpKameraPhi)*M_PI/180.0);
            GL_RysujScene();
         }
         //Tu nie nalezy wywolywac GL_RysujScene, bo moze nie byc przesuwania ani
obrotow
         break;
      }
      case WM_LBUTTONUP:
```

```
            KameraPhi+=tmpKameraPhi;
            KameraTheta+=tmpKameraTheta;
            tmpKameraPhi=0;
            tmpKameraTheta=0;
            //kosmetyka
            if (KameraPhi>=360) KameraPhi-=360;
            if (KameraPhi<0) KameraPhi+=360;
            if (KameraTheta>=360) KameraTheta-=360;
            if (KameraTheta<0) KameraTheta+=360;
            break;
    }
}
```

<< koniec ćwiczenia >>

4.2 Odległość kamery (rolka)

Ruch myszki steruje ruchem kamery na sferze wokół środka układu współrzędnych. Natomiast rolką myszki ustalać będziemy promień tej sfery, a więc odległość kamery od naszego ostrosłupa. To oznacza, że do powyższej metody musimy dodać jeszcze sekcję wykonywaną w razie odebrania komunikatu `WM_MOUSEWHEEL` (listing 35).

Listing 35. Zbliżanie i oddalanie kamery

```
case WM_MOUSEWHEEL:
{
    const float czuloscMyszy=10.0f;
    short WheelDelta=(short)HIWORD(Message.WParam);
    //zmiana odleglosci kamery od pocz. ukl. wsp.
    KameraR=KameraR*(1+WheelDelta/abs(WheelDelta)/czuloscMyszy);
    GL_RysujScene();
    break;
}
```

<< koniec ćwiczenia >>

W kodzie klasy `TGLForm` dostępnym w dołączonych do skryptu źródłach zdefiniowane jest pole `debug_mode`, które, jeżeli ustawiona jest na `true`, powoduje wyświetlanie w metodzie `WndProc` współrzędnych położenia kamery.

W momencie naciśnięcia przycisku myszy zapamiętywane jest położenie myszki (do tego wykorzystujemy pola `X0` i `Y0`). W sekcji wykonywanym w przypadku odebrania komunikatu WM_MOUSEMOVE, jeżeli przyciśnięty jest lewy klawisz myszy, obliczana jest wielkość przesunięcia myszy od momentu naciśnięcia przycisku myszy i na tej podstawie obliczane są kąty obrotu zapisywane w polach `tmpKameraPhi` i `tmpKameraTheta`. W metodzie `RysujScene` wartości te wykorzystywane są jako argumenty funkcji `glRotate`. W momencie zwolnienia przycisku myszy wartości pól `tmpKameraPhi` i `tmpKameraTheta` przepisywane są do pól `KameraPhi` i `KameraTheta`, co oznacza, że obrót jest zapamiętywany.

4.3 Przesuwanie sceny myszą (prawy klawisz myszy)

Lewy przycisk myszy zarezerwowaliśmy dla ruchu kamery wokół środka układu współrzędnych tj. do kontroli kątów we współrzędnych sferycznych. Rolka kontroluje promień w tych samych współrzędnych. W ten sposób możemy tylko obracać kamerę wokół środka układu współrzędnych, ale nie możemy jej po prostu przesunąć w górę lub w lewo.
Pozwolimy na to teraz wykorzystując prawy przycisk myszy. Nowy kod będzie oparty na tej samej idei, co w przypadku obrotów.

1. Deklarujemy prywatne pola klasy `TGLForm`:
   ```
   float KameraX, KameraY;
   float tmpKameraX, tmpKameraY;
   ```
2. Pola `KameraX` i `KameraY` inicjujemy w konstruktorze zerami.
   Listing 36.
   ```
   __fastcall TGLForm::TGLForm(TComponent* Owner)
      :TForm(Owner),
   ```

```
    X0(0),Y0(0),
    KameraR(10.0f),KameraPhi(0.0f),KameraTheta(0.0f),
    KameraX(0.0f),KameraY(0.0f)
{
```

3. Następnie uzupełniamy metodę WndProc poleceniami wykonywanymi w przypadk kliknięcia prawego przycisku myszy (listing 37)

Listing 37.

```
void __fastcall TGLForm::WndProc(TMessage& Message)
{
   TForm::WndProc(Message);

   switch(Message.Msg)
   {
      case WM_PAINT: //odswiezenie formy
         GL_RysujScene();
         break;

      case WM_SIZE: //zmiana rozmiaru formy
      case WM_SIZING:
         GL_UstawienieSceny();
         GL_RysujScene();
         break;

      case WM_KEYDOWN:
         if (Message.WParam==VK_ESCAPE) Close();
         if (Message.WParam==VK_F1) Pomoc();
         break;

      //KONTROLA POLOZENIA KAMERY MYSZA
      case WM_LBUTTONDOWN:
      case WM_RBUTTONDOWN:
         X0=LOWORD(Message.LParam); //biezaca poz. x kursora
         Y0=HIWORD(Message.LParam); //biezaca poz. y kursora
         break;
      case WM_MOUSEMOVE:
      {
         int X=LOWORD(Message.LParam); //int X=Message.LParam & 0x0000FFFF;
         int Y=HIWORD(Message.LParam); //int Y=(Message.LParam & 0xFFFF0000) >> 16;
         if (X>Screen->Width) X-=0xFFFF;
         if (Y>Screen->Height) Y-=0xFFFF;
         int dX=X-X0;
         int dY=Y-Y0;
         if (Message.WParam==MK_LBUTTON) //wylacznie lewy przycisk myszy
         {
            const float czuloscMyszy=5.0f;
            tmpKameraPhi=dX/czuloscMyszy;
            tmpKameraTheta=dY/czuloscMyszy;
            GL_RysujScene();
         }
         if (Message.WParam==MK_RBUTTON) //wylacznie prawy przycisk myszy
         {
            const float czuloscMyszy=75.0f; //powinno zalezec od rozmiaru frustum
            tmpKameraX=dX/czuloscMyszy;
            tmpKameraY=-dY/czuloscMyszy; //po uklad wsp. OpenGL ma y do gory, a wsp.
ekranu ma y do dolu
            GL_RysujScene();
         }
         //Tu nie nalezy wywolywac GL_RysujScene, bo moze nie byc przesuwania ani
obrotow
         break;
      }
      case WM_LBUTTONUP:
         KameraPhi+=tmpKameraPhi;
         KameraTheta+=tmpKameraTheta;
         tmpKameraPhi=0;
         tmpKameraTheta=0;
         //kosmetyka
         if (KameraPhi>=360) KameraPhi-=360;
         if (KameraPhi<0) KameraPhi+=360;
         if (KameraTheta>=360) KameraTheta-=360;
         if (KameraTheta<0) KameraTheta+=360;
         break;
      case WM_RBUTTONUP:
         KameraX+=tmpKameraX;
         KameraY+=tmpKameraY;
         tmpKameraX=0;
```

```
            tmpKameraY=0;
            break;
        case WM_MOUSEWHEEL:
        {
            const float czuloscMyszy=10.0f;
            short WheelDelta=(short)HIWORD(Message.WParam);
            //zmiana odleglosci kamery od pocz. ukl. wsp.
            KameraR=KameraR*(1+WheelDelta/abs(WheelDelta)/czuloscMyszy);
            GL_RysujScene();
            break;
        }
    }
}
```

4. Teraz pozostaje wprowadzić do metody `GL_RysujScene` polecenie przesuwające kamerę o wielkość wyznaczoną myszką (listing 38)

   Listing 38.

```
void __fastcall TGLForm::GL_RysujScene()
{
    //Przygotowanie bufora
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity(); //macierz model-widok = macierz jednostkowa
    gluLookAt(0,0,KameraR,  //polozenie kamery
              0,0,0,        //punkt, na ktory skierowana jest kamera
              0,1,0);       //kierunek "do gory" kamery (polaryzacja)
    glTranslatef(KameraX+tmpKameraX,KameraY+tmpKameraY,0);
    glRotatef(KameraPhi+tmpKameraPhi, 0.0, 1.0, 0.0); //wokol OY
    glRotatef(KameraTheta+tmpKameraTheta, 1.0, 0.0, 0.0); //wokol OX

    //przeksztalcenia macierzy model-widok i rysowanie figur
    RysujScene();

    //Z bufora na ekran
    glFlush();
    SwapBuffers(uchwytDC);
}
```

4.4 Inicjowane myszą swobodne obroty kamery

Spore wrażenie robi możliwość rozkręcenia kamery, która obraca się potem z nadaną myszką prędkością. Uzupełnijmy naszą klasę o taką możliwość. Do tego celu konieczne będzie użycie osobnego wątku lub komponentu `TTimer` (który zresztą też tworzy osobny wątek).

1. Do pliku *GLForm.cpp* dodajemy dyrektywę włączającą nagłówek modułu, w którym znajduje się definicja komponentu `TTimer`: `#include <ExtCtrls.hpp>`.

2. Dodajmy zespół nowych prywatnych pól do klasy `TGLForm`. Aby poprawić czytelność kodu nazwa wszystkich tych pól związanych ze swobodnymi obrotami zaczynać będzie się od słowa „Szybkosc”:

```
int SzybkoscX0,SzybkoscY0;
float SzybkoscPhi, SzybkoscTheta;
bool SzybkoscWlaczone;
TTimer* SzybkoscTimer;
```

3. Zainicjujmy je w konstruktorze klasy `TGLForm`:

Listing 39.

```
__fastcall TGLForm::TGLForm(TComponent* Owner)
    :TForm(Owner),
    X0(0),Y0(0),
    KameraR(10.0f),KameraPhi(0.0f),KameraTheta(0.0f),
    KameraX(0.0f),KameraY(0.0f),
    SzybkoscX0(0),SzybkoscY0(0),
    SzybkoscPhi(0.0f), SzybkoscTheta(0.0f),
    SzybkoscWlaczone(true)
{
...
```

4. Dwa pierwsze pola pomagać będą w obliczaniu szybkości poruszania myszką. Służące do tego polecenia wstawmy do `WndProc` do sekcji obsługującej zdarzenie `WM_MOUSEMOVE`:

   Listing 40

```
        case WM_MOUSEMOVE:
        {
            int X=LOWORD(Message.LParam); //int X=Message.LParam & 0x0000FFFF;
```

```
        int Y=HIWORD(Message.LParam); //int Y=(Message.LParam & 0xFFFF0000) >> 16;
        if (X>Screen->Width) X-=0xFFFF;
        if (Y>Screen->Height) Y-=0xFFFF;
        int dX=X-X0;
        int dY=Y-Y0;
        if (Message.WParam==MK_LBUTTON) //wylacznie lewy przycisk myszy
        {
           const float czuloscMyszy=5.0f;
           tmpKameraPhi=dX/czuloscMyszy;
           tmpKameraTheta=dY/czuloscMyszy;
           //szybkosc
           SzybkoscPhi=X-SzybkoscX0;
           SzybkoscTheta=Y-SzybkoscY0;
           SzybkoscX0=X;
           SzybkoscY0=Y;
           if(debug_mode) DebugInfo();
           GL_RysujScene();
        }
        if (Message.WParam==MK_RBUTTON) //wylacznie prawy przycisk myszy
        {
           const float czuloscMyszy=75.0f; //powinno zalezec od rozmiaru frustum
           tmpKameraX=dX/czuloscMyszy;
           tmpKameraY=-dY/czuloscMyszy; //po uklad wsp. OpenGL ma y do gory, a wsp.
ekranu ma y do dolu
           if(debug_mode) DebugInfo();
           GL_RysujScene();
        }
        //Tu nie nalezy wywolywac GL_RysujScene, bo moze nie byc przesuwania ani
obrotow
        break;
     }
```

5. Następnie zajmujemy się komponentem `TTimer`. Zacznijmy od zdefiniowania uruchamianej przez niego cyklicznie metody (listing 41). Trzeba ją oczywiście zadeklarować w *GLForm.h*.

Listing 41.

```
void __fastcall TGLForm::SzybkoscTimerTimer(TObject* Sender)
{
   //x=v*t
   KameraPhi+=SzybkoscPhi*SzybkoscTimer->Interval/100;
   KameraTheta+=SzybkoscTheta*SzybkoscTimer->Interval/100;
   GL_RysujScene();
}
```

6. Teraz możemy utworzyć i skonfigurować timer. W tym celu do konstruktora `TGLForm` dodajemy polecenia wyróżnione na poniższym listingu:

Listing 42.

```
 __fastcall TGLForm::TGLForm(TComponent* Owner)
    :TForm(Owner),
    X0(0),Y0(0),
         KameraR(10.0f),KameraPhi(0.0f),KameraTheta(0.0f),
    KameraX(0.0f),KameraY(0.0f),
    SzybkoscX0(0),SzybkoscY0(0),
         SzybkoscPhi(0.0f), SzybkoscTheta(0.0f),
    SzybkoscWlaczone(true)
 {
    //zmiana wlasnosci okna
    //BorderStyle=bsSingle;
    //TBorderIcons ikonyFormy=BorderIcons;
    //ikonyFormy >> biMaximize;
    //BorderIcons=ikonyFormy;

    //tworzenie timera odpowiedzialnego za swobodne obroty
    SzybkoscTimer=new TTimer(this);
    SzybkoscTimer->Enabled=false;
    SzybkoscTimer->Interval=10;
    SzybkoscTimer->OnTimer=SzybkoscTimerTimer;

    //biezace okno staje sie oknem OpenGL
    uchwytDC=GetDC(Handle);
    if (!GL_UstalFormatPikseli(uchwytDC)) ShowMessage("Nie udało się ustalić formatu
pikseli");
    uchwytRC=wglCreateContext(uchwytDC);
    if (uchwytRC==NULL) ShowMessage("Nie udało się pobrać uchwytu kontekstu grafiki");
    if (!wglMakeCurrent(uchwytDC,uchwytRC)) ShowMessage("Inicjacja grafiki OpenGL nie
powiodła się");
    GL_UstawienieSceny();
```

```
    Caption=(AnsiString)"GLForm, OpenGL "+(char*)glGetString(GL_VERSION);
  }
```

Szybkość odświeżania powinna być nie mniejsza niż kilkadziesiąt herców. To oznacza, że własność `Interval` (mierzony w milisekundach czas między wywołaniami metody zdarzeniowej) nie może być większa niż 10.

7. Pozostaje kwestia uruchamiania i zatrzymywania animacji. Uruchamiana będzie w momencie zwolnienia lewego klawisza myszy, a zatrzymywana w przypadku jego naciśnięcia. To prowadzi do kolejnych zmian w `WndProc`.

Listing 43. Uruchamianie animacji

```
case WM_LBUTTONUP:
   KameraPhi+=tmpKameraPhi;
   KameraTheta+=tmpKameraTheta;
   tmpKameraPhi=0;
   tmpKameraTheta=0;
   //kosmetyka
   if (KameraPhi>=360) KameraPhi-=360;
   if (KameraPhi<0) KameraPhi+=360;
   if (KameraTheta>=360) KameraTheta-=360;
   if (KameraTheta<0) KameraTheta+=360;
   //inicjacja swobodnyh obrotow
   if ((SzybkoscWlaczone && ((SzybkoscPhi!=0) || (SzybkoscTheta!=0))))
      SzybkoscTimer->Enabled=true;
   break;
```

Listing 44. Wstrzymywanie animacji

```
case WM_LBUTTONDOWN:
case WM_RBUTTONDOWN:
   X0=LOWORD(Message.LParam); //biezaca poz. x kursora
   Y0=HIWORD(Message.LParam); //biezaca poz. y kursora
   //wstrzymywanie swobodnych obrotow
   SzybkoscX0=X0;
   SzybkoscY0=Y0;
   SzybkoscPhi=0;
   SzybkoscTheta=0;
   SzybkoscTimer->Enabled=false;
   break;
```

4.5 Dodanie wygaszania obrotów

Aby obroty nie trwały w nieskończoność możemy stopniowo zmniejszać ich prędkość aż do całkowitego zatrzymania.

1. Dodajmy pole `SzybkoscWygaszania` typu `float`, którym będziemy mogli kontrolować szybkość wygaszania swobodnych obrotów.
2. W konstruktorze zainicjujmy ją wartością `0.3f`.
3. Teraz wystarczy do metody wywoływanej przez timer dodać polecenia zmniejszające prędkość obrotów (listing 45). Definiujemy także funkcję pomocniczą wyznaczającą znak liczby – wykorzystujemy ją do określenia kierunku zmian.

Listing 45.

```
#include <math.h>

char sign(double arg)
{
   if (arg==0) return 0;
   else return (char)(arg/fabs(arg));
}

void __fastcall TGLForm::SzybkoscTimerTimer(TObject* Sender)
{
   //x=v*t
   KameraPhi+=SzybkoscPhi*SzybkoscTimer->Interval/100;
   KameraTheta+=SzybkoscTheta*SzybkoscTimer->Interval/100;
   GL_RysujScene();

   if(SzybkoscWygaszania>0)
   {
```

```
      float     SzybkoscWygaszaniaPhi=sign(SzybkoscPhi)*SzybkoscWygaszania*SzybkoscTimer-
>Interval/100.0;
      float
SzybkoscWygaszaniaTheta=sign(SzybkoscTheta)*SzybkoscWygaszania*SzybkoscTimer-
>Interval/100.0;
      if (fabs(SzybkoscPhi)<fabs(SzybkoscWygaszaniaPhi))
         SzybkoscWygaszaniaPhi=SzybkoscPhi; //w nastepnym kroku zostanie zatrzymany
      if (fabs(SzybkoscTheta)<fabs(SzybkoscWygaszaniaTheta))
         SzybkoscWygaszaniaTheta=SzybkoscTheta;
      SzybkoscPhi-=SzybkoscWygaszaniaPhi;
      SzybkoscTheta-=SzybkoscWygaszaniaTheta;
      if ((fabs(SzybkoscPhi)<0.01) && (fabs(SzybkoscTheta)<0.01))
      {
         SzybkoscPhi=0;
         SzybkoscTheta=0;
         SzybkoscTimer->Enabled=false;
      }
   }
}
```

4.6 Funkcja pozwalająca na inicjację obrotów

Na koniec możemy dodać do klasy `TGLForm` metodę o zakresie chronionym (a więc dostępną w klasie potomnej), która pozwoli na inicjowanie obrotów z poziomu kodu (listing 46). Należy oczywiście zadeklarować ją w nagłówku klasy.

Listing 46.

```
void __fastcall TGLForm::Obracaj(float SzybkoscPhi,float SzybkoscTheta,float
SzybkoscWygaszania=-1)
{
   this->SzybkoscPhi=SzybkoscPhi;
   this->SzybkoscTheta=SzybkoscTheta;
   if(SzybkoscWygaszania>=0) this->SzybkoscWygaszania=SzybkoscWygaszania;
   SzybkoscTimer->Enabled=true;
}
```

<< koniec ćwiczenia >>

Aby przetestować tę metodę umieśćmy w konstruktorze klasy `TForm1` instrukcję `Obracaj(10.0f,1.0f,0.0f);`.

5. Nowi aktorzy

5.1 Rysowanie osi układu współrzędnych

Narysujmy dwa układy współrzędnych: biały i niebieski. W naszym przykładzie białe osie będą odgrywały rolę nieruchomej części sceny. Ustalamy, że wszystkie przekształcenia macierzy model-widok wyprzedzające narysowanie owych białych osi nazywać będziemy ruchem kamery. Wszystkie późniejsze — ruchem aktorów.

1. W klasie `TGLForm` definiujemy metodę `RysujOsie` rysującą osie układu współrzędnych OXYZ (listing 47):

Listing 47. Jeżeli chcemy narysować dwa układy współrzędnych, wygodnie przygotować odpowiednią metodę, którą będziemy mogli wielokrotnie wykorzystywać

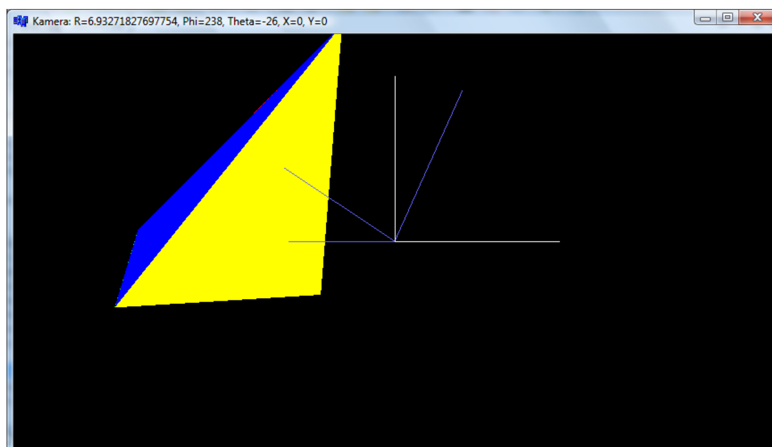
```
void __fastcall TGLForm::RysujOsie(float rozmiar)
{
   glBegin(GL_LINES);
   glVertex3f(0,0,0); glVertex3f(rozmiar,0,0); //OX, w prawo
   glVertex3f(0,0,0); glVertex3f(0,rozmiar,0); //OY, do gory
   glVertex3f(0,0,0); glVertex3f(0,0,rozmiar); //OZ, do kamery
   glEnd();
}
```

13. Metodę RysujOsie wywołujemy w metodzie GL_RysujScene po raz pierwszy przed poleceniami wykonującymi obroty i translacje ostrosłupa, a po raz drugi już po narysowaniu ostrosłupa (listing 48). Pierwsze wywołanie poprzedzamy zmianą koloru na biały, a drugie — na niebieski.

Listing 48. Rysowanie nieruchomego i ruchomego układu współrzędnych

```
void __fastcall TGLForm::GL_RysujScene()
{
   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa

   gluLookAt(0,0,KameraR,  //polozenie kamery
             0,0,0,        //punkt, na ktory skierowana jest kamera
             0,1,0);       //kierunek "do gory" kamery (polaryzacja)

   //nieruchome osie ukladu wspolrzednych zwiazane ze scena (frustum)
   float rozmiarUkladuWspolrzednych=1.0f;
   glColor3ub(255,255,255);
   RysujOsie(rozmiarUkladuWspolrzednych);

   //obroty i przesuniecia kamery
   glTranslatef(KameraX+tmpKameraX,KameraY+tmpKameraY,0);
   glRotatef(KameraPhi+tmpKameraPhi, 0.0, 1.0, 0.0); //wokol OY
   glRotatef(KameraTheta+tmpKameraTheta, 1.0, 0.0, 0.0); //wokol OX

   //przekształcenia macierzy model-widok i rysowanie figur
   RysujScene();

   //ruchome osie ukladu wspolrzednych
   glColor3ub(100,100,255);
   RysujOsie(rozmiarUkladuWspolrzednych);

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```

Metoda RysujOsie wywoływana jest dwukrotnie. Pierwsze wywołanie następuje tuż po odsunięciu kamery od początku układów współrzędnych, ale przed poleceniami realizującymi obroty i translacje związane z kontrolą kamery myszą (co możemy interpretować też jako ruchy aktorów na scenie). Dlatego ten rysowany na biało układ współrzędnych można traktować jak związany z nieruchomą sceną i nie mamy wpływu na jego ułożenie (por. rysunek 12). Drugi raz metoda RysujOsie wywoływana jest tuż przed wymianą buforów, dotyczą jej zatem wszystkie przekształcenia macierzy model-widok — innymi słowy, osie związane są z ostatnimi rysowanymi na scenie aktorami.



Rysunek 12. *Rysujemy dwie pary osi współrzędnych — pierwsza związana jest z nieruchomym układem odniesienia sceny, druga — z układem związanym z ostrosłupem*

5.2 Dodawanie kolejnych figur

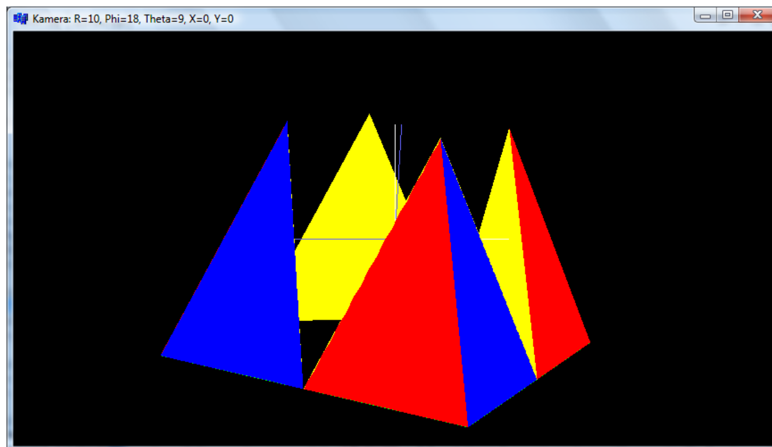
Do pierwszego ostrosłupa dorysujemy kolejne, uzupełniając metodę RysujScene w klasie TForm1 o pętlę wyróżnioną w listingu 49. Każdy z ostrosłupów obrócony jest o 90 stopni względem poprzedniego.

Wszystkie będą względem siebie nieruchome, ale dodanie względnego ruchu między nimi byłoby jedynie kwestią uzmiennienia wielkości obrotu, jaki jest wykonywany przed kolejnym wywołaniem metody RysujOstroslup. Uzyskany efekt widoczny jest na rysunku 13.

Listing 49. Klonowanie ostrosłupów

```
void __fastcall TForm1::RysujScene()
{
   //obroty kontrolowane klawiszami
   glRotatef(Phi, 0.0, 1.0, 0.0); //wokol OY
   glRotatef(Theta, 1.0, 0.0, 0.0); //wokol OX

   //przesuniecia kontrolowane klawiszami
   glTranslatef(PozycjaX,PozycjaY,PozycjaZ);

   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //rysowanie ostroslupa
   RysujOstroslup(x0,y0,z0);

   //kolejne ostroslupy
   for (int i=0;i<3;i++)
   {
      glRotatef(90.0, 0.0, 1.0, 0.0);
      RysujOstroslup(x0,y0,z0);
   }
   glRotatef(90.0, 0.0, 1.0, 0.0); //dopelnienie pelnego obrotu
}
```

Rysunek 13.
Korona zbudowana z czterech ostrosłupów



5.3 Stos macierzy model-widok

Maszyna stanu biblioteki OpenGL wyposażona jest w dwa stosy macierzy: jeden dla macierzy rzutowania, drugi dla macierzy model-widok. Każdy z nich ma rozmiar nie mniejszy niż 32 macierze.
Jeżeli na scenie umieszczamy wielu aktorów, a każdy z nich może mieć osobne ruchome części, to zazwyczaj niewygodne jest kontrolowanie, aby kolejne przekształcenia macierzy model-widok wprowadzane dla jednego aktora nie interferowały z przekształceniami następnego. Dlatego wygodnie jest przed ustalaniem pozycji aktora zapamiętać aktualną macierz model-widok, a po zakończeniu rysowania aktora – odtworzyć ją z pamięci.
Do pierwszej czynności służy funkcja glPushMatrix umieszczająca macierz na stosie, a do drugiej – glPopMatrix zdejmująca macierz ze stosu.
W przypadku naszego kodu, w którym jest na razie tylko jeden aktor (zbiór czterech ostrosłupów) korzystanie ze stosu macierzy model-widok jest pewnie przesadą, ale dzięki temu można uniknąć obrotu wykonywanego po pętli, który ma przywrócić pierwotną orientację sceny (zob. listing 49). Listing 50 pokazuje w jaki sposób można zapamiętać i odtworzyć stan macierzy model-widok.

Listing 50.

```
void __fastcall TForm1::RysujScene()
{
```

```
   //obroty kontrolowane klawiszami
   glRotatef(Phi, 0.0, 1.0, 0.0); //wokol OY
   glRotatef(Theta, 1.0, 0.0, 0.0); //wokol OX

   //przesuniecia kontrolowane klawiszami
   glTranslatef(PozycjaX,PozycjaY,PozycjaZ);

   glPushMatrix();

   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //rysowanie ostroslupa
   RysujOstroslup(x0,y0,z0);

   //kolejne ostroslupy
   for (int i=0;i<3;i++)
   {
      glRotatef(90.0, 0.0, 1.0, 0.0);
      RysujOstroslup(x0,y0,z0);
   }
   glRotatef(90.0, 0.0, 1.0, 0.0); //dopelnienie pelnego obrotu

   glPopMatrix();
}
```

5.4 Rysowanie sześcianu

Pewnie wszystkim znudził się już ostrosłup, nawet powielony. Narysujmy zatem nową bryłę – sześcian. Tym razem w argumencie funkcji glBegin ****

1. W tym celu w klasie `TForm1` definiujemy metodę `RysujSzescian` widoczną na listingu 51.

   Listing 51.

```
void __fastcall TForm1::RysujSzescian(float krawedz) const
{
   const float a=krawedz;

   glBegin(GL_QUADS);
   //tylnia
   glVertex3f(-a,-a,-a);
   glVertex3f(-a,a,-a);
   glVertex3f(a,a,-a);
   glVertex3f(a,-a,-a);
   //przednia
   glVertex3f(-a,-a,a);
   glVertex3f(a,-a,a);
   glVertex3f(a,a,a);
   glVertex3f(-a,a,a);

   //prawa
   glVertex3f(a,-a,a);
   glVertex3f(a,a,a);
   glVertex3f(a,a,-a);
   glVertex3f(a,-a,-a);
   //lewa
   glVertex3f(-a,-a,a);
   glVertex3f(-a,a,a);
   glVertex3f(-a,a,-a);
   glVertex3f(-a,-a,-a);

   //gorna
   glVertex3f(-a,a,a);
   glVertex3f(a,a,a);
   glVertex3f(a,a,-a);
   glVertex3f(-a,a,-a);
   //dolna
   glVertex3f(-a,-a,a);
   glVertex3f(a,-a,a);
   glVertex3f(a,-a,-a);
   glVertex3f(-a,-a,-a);
   //
```

```
        glEnd();
    }
```

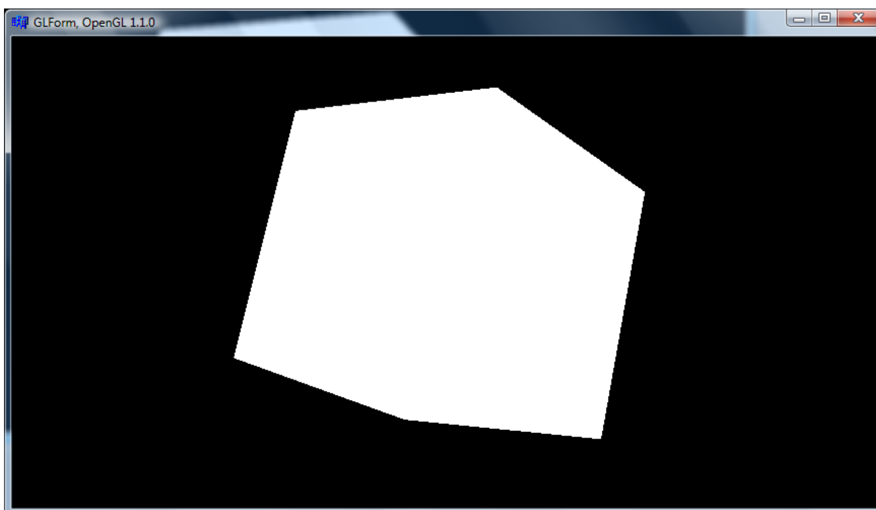
2. Należy ją oczywiście zadeklarować w pliku nagłówkowym.
3. Następnie w metodzie `RysujScene` usuwamy lub „zakomentowujemy” polecenia związane z rysowaniem ostrosłupów, a w zamian wstawiamy wywołanie metody `RysujSzescian` (listing 52).

Listing 52.

```
    void __fastcall TForm1::RysujScene()
    {
        //obroty kontrolowane klawiszami
        glRotatef(Phi, 0.0, 1.0, 0.0); //wokol OY
        glRotatef(Theta, 1.0, 0.0, 0.0); //wokol OX

        //przesuniecia kontrolowane klawiszami
        glTranslatef(PozycjaX,PozycjaY,PozycjaZ);

        glPushMatrix();

        const float x0=1.0;

        //rysowanie szescianu
        //glColor3ub(0,0,150);
        RysujSzescian(x0);

        glPopMatrix();
    }
```

<< koniec ćwiczenia >>

Ponieważ w metodzie `RysujSzescian` nie ma poleceń określających kolory poszczególnych ścian, cały sześcian jest jednokolorowy. To oczywiście w poważnym stopniu niweluje efekt trójwymiarowości (rysunek 14). Ale jest to stan zamierzony – uzasadni wprowadzenie systemu oświetlenia na scenie.



Rysunek 14. Bez oświetlenia lub kolorów trudno dostrzec w sześcianie figurę przestrzenną

5.5 Maksymalizacja okna klawiszem Enter

W każdej chwili możemy zwiększyć okno klikając przycisk maksymalizacji na pasku tytułu. Dzięki temu, że klasa TGLForm wykrywa odbiór komunikatu WM_SIZE, zawartość okna zostanie natychmiast przeskalowana. W aplikacjach korzytających z grafiki czasem możliwe jest przełączanie między oknem zmaksymalizowanym, a normalnym za pomocą klawisza *Enter*. Zaimplementujmy to wygodne rozwiązanie także w naszej przykładowej aplikacji.

1. Definiujemy prywatne pole klasy `TGLForm` typu logicznego:
   ```
   bool maksymalizacjaOkna;
   ```
2. Inicjujemy je w konstruktorze wartością `false`.

3. Następnie definiujemy prywatną metodę zmieniającą sposób wyświetlania okna:

```
void __fastcall TGLForm::MaksymalizujPrzywroc()
{
   maksymalizacjaOkna=!maksymalizacjaOkna;
   if(maksymalizacjaOkna)
   {
      ShowWindow(Handle,SW_MAXIMIZE);
      //this->BorderStyle=bsNone;
   }
   else
   {
      ShowWindow(Handle,SW_RESTORE);
      //this->BorderStyle=bsSingle;
   }
   GL_UstawienieSceny();
}
```

4. Do przełączania trybu wyświetlania okna służyć ma naciśnięcie klawisza *Enter*. Uzupełnijmy wobec tego metodę `WndProc`:

Listing 53.

```
case WM_KEYDOWN:
   if (Message.WParam==VK_ESCAPE) Close();
   if (Message.WParam==VK_F1) Pomoc();
   if (Message.WParam==13) MaksymalizujPrzywroc();
   break;
```

5. Należy jeszcze uwzględnić fakt, że użytkonik może zmaksymalizować rozmiar okna lub przywrócić pierwotny rozmiar za pomocą ikony na pasku tytułu. Na szczęście okno jest o tym informowane komunikatem `WM_SYSCOMMAND`. To pozwala na przełączenie stanu pola `maksymalizacjaOkna` (listing 54).

   Listing 54.

```
case WM_SYSCOMMAND: //wykrywanie maksymalizacji
   if (Message.WParam==SC_MAXIMIZE) maksymalizacjaOkna=true;
   if (Message.WParam==SC_RESTORE) maksymalizacjaOkna=false;
   break;
```

5.6 Tryb (pseudo) pełnoekranowy

Jeżeli maksymalizację okna przeprowadzimy przed pokazaniem formy, a jednocześnie zmienimy jej styl na pozbawiony brzegów i paska tytułu (własność `BorderStyle` ustawiona na `bsNone`)[13], obszar klienta zajmować będzie cały ekran włącznie z paskiem zadań. Uzyskamy w ten sposób imitację trybu pełnoekranowego. Nie jest to najlepszy sposób uruchamiania gier, ale np. do wygaszaczy ekranu wystarczy w zupełności

6. Kolor i światło

Co to jest kolor? I jaki jest jego związek ze światłem? Z fizycznego punktu widzenia kolor można identyfikować z długością fali światła docierającego do oka. Podobnie jak w przypadku dźwięku, nasze zmysły dość dobrze radzą sobie z jednoczesną detekcją fal o różnych długościach. Niestety, tak jak w znanym dowcipie o matematyku w balonie, odpowiedź ta jest prawdziwa, dla niektórych fascynująca, ale w przypadku definiowania oświetlenia w programowaniu grafiki trójwymiarowej mało przydatna. Ważniejsza od fizyki okazuje się tu biologia. Mówi ona, że oko rejestruje światło za pomocą trzech typów czopków reagujących na fale o różnych długościach oraz pręcików, które są czułe jedynie na natężenie światła. Skupmy się na czopkach. Wytworzone przez każdy z typów czopków impulsy nerwowe interpretowane są przez mózg jako kolory czerwony, zielony i niebieski. Pobudzenie wszystkich jednocześnie w równym stopniu, co odpowiada światłu, które jest „mieszaniną" fal o różnych długościach, daje kolor biały (względnie szary, jeżeli pobudzenie jest mniej intensywne). Różne kombinacje pobudzeń czopków prowadzą do całej palety kolorów, jakie szczęśliwie możemy oglądać. Wynika z tego, że kolory czerwony (R), zielony (G) i niebieski (B) tworzą układ współrzędnych (oznaczany symbolem RGB), w którym można zapisać dowolny widziany przez człowieka kolor. Ten fakt został wykorzystany w telewizorze, który generuje obraz zbudowany z czerwonych, zielonych i niebieskich pikseli. W ten sam sposób będziemy również określać kolor źródeł światła.

[13] W Windows Vista zmiana własności `BorderStyle` w trakcie działania aplikacji (tj. z poziomu kodu) prowadzi do poważnych błędów w wyświetlaniu obrazu OpenGL.

To jednak nie wyczerpuje tematu barwy i oświetlenia. Do oświetlania używamy zwykle światła o białej barwie. Rzeczy, które oświetlamy, są jednak różnobarwne. Z tego wynika, że czymś innym jest **kolor źródła światła** i **kolor oświetlanych przedmiotów**. To rozróżnienie odzwierciedlone jest również w OpenGL. Kolor oświetlanego przedmiotu, jaki widzimy, zależy wyłącznie od światła, jakie biegnie od tego przedmiotu w kierunku naszego oka. Załóżmy, że przedmiot ten nie emituje światła, a jedynie je odbija. Jeżeli przedmiot ten nie odbija wszystkich długości fali, to część padającego na ten przedmiot światła jest pochłaniana. W efekcie przedmioty oświetlane modyfikują światło, które na nie pada. I to nie znaczy tylko, że je osłabia, ale również zmienia jego kolor. Przedmiot może bowiem pochłaniać różne długości fali w różnym stopniu (innymi słowy niejednakowo odbijać składowe RGB). Zatem jeżeli źródło światła jest białe, a przedmiot pochłania składową niebieską, a dobrze odbija składowe czerwoną i zieloną, to przedmiot ten widzimy jako żółty[14]. I tak właśnie ustala się kolor przedmiotów na scenie w OpenGL — ustalając współczynniki odbicia poszczególnych składowych RGB światła na powierzchni tego przedmiotu.

Żeby bardziej sprawę skomplikować, powiem jeszcze tylko, że funkcja `glColor`, której użyliśmy w projektach 197. i 207., z całą tą teorią nie ma nic wspólnego. Nie zależy bowiem w żaden sposób od oświetlenia sceny — jest zwyczajnym oszustwem, które wprowadzone zostało do OpenGL, aby możliwe było rysowanie bez definiowania źródeł światła. To oszustwo możemy jednak zmniejszyć, jeżeli zażyczymy sobie, aby współczynniki odbicia ustalane zwykle funkcją `glMaterial` były uzgadniane z kolorem ustalonym za pomocą funkcji `glColor` (jednym słowem, aby użycie tych dwóch funkcji było równoważne). Służy do tego funkcja `glColorMaterial`, której użyjemy w kolejnych projektach.

6.1 Włączenie systemu oświetlenia i ustawianie światła tła

Nie można poprawnie oświetlić przedmiotu, jeżeli nie ustawi się wcześniej odpowiednich własności materiału, z którego jest on wykonany. Kolor, który ustawialiśmy, korzystając z funkcji `glColor`, nie jest kolorem w sensie fizycznym (nie wynika z własności materiału, a dokładnie z jego własności odbijania i pochłaniania światła). Zobaczymy to po włączeniu mechanizmu oświetlania. Nawet pomimo włączenia oświetlenia tła, cała scena pogrąży się w mroku — nasze ostrosłupy nie będą odbijać światła.

1. W pliku nagłówkowym *GLForm.h* deklarujemy metodę `GL_Oswietlenie` klasy `TGLForm` o zakresie chronionym oraz pole prywatne `natezenie_swiatla_otoczenia` typu `float`:
```
protected:
   virtual void __fastcall GL_Oswietlenie();
   virtual void __fastcall Oswietlenie(){};
private:
   float natezenie_swiatla_otoczenia;
```

2. Pole `natezenie_swiatla_otoczenia` inicjujemy wartością `0.3f` w konstruktorze klasy `TGLForm`.

3. Następnie definiujemy (w pliku *GLForm.cpp*) metodę `GL_Oswietlenie` (listing 55). W niej włączamy system oświetlenia i uruchamiamy światło otoczenia (ang. *ambient*):

Listing 55. Funkcja, w której umieszczać będziemy wszystkie polecenia ustalające oświetlenie

```
void __fastcall TGLForm::GL_Oswietlenie()
{
   const float kolor_otoczenie[]={natezenie_swiatla_otoczenia,
                                  natezenie_swiatla_otoczenia,
                                  natezenie_swiatla_otoczenia}; //biel-odcienie
                                  szarości
   glEnable(GL_LIGHTING); //wlaczenie systemu oswietlania
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT,kolor_otoczenie); //swiatlo tla

   Oswietlenie();
}
```

4. Metodę tę wywołujemy na samym końcu metody `GL_UstawienieSceny`.

---

[14] Czasem zamiast mówić, że przedmiot dobrze odbija składowe R i G, wygodniej jest powiedzieć, że odbija składową Y (żółć). Zamiast R i B można podstawić M (magentę), a zamiast G i B — C (cyjan). W ten sposób powstaje nowy układ współrzędnych, oznaczany jako CMY. O ile RGB wygodne jest w urządzeniach opartych na emisji światła (monitory), to CMY bardziej pasuje do określania kolorów na przedmiotach oświetlanych, tj. absorbujących światło (np. wydruk na papierze lub obraz na płótnie). To właśnie kolory z układu współrzędnych CMY uznawane są przez malarzy za kolory podstawowe. Takimi kolorami „plują” także drukarki atramentowe.

Włączenie systemu oświetlenia `glEnable(GL_LIGHTING);` bez ustalenia źródła światła i własności materiałów, z jakich zbudowane są obiekty znajdujące się na scenie, powoduje, że cała scena pogrąża się w mroku.

5. Wyposażmy klasę `TGLForm` we własność (dostępną w inspektorze obiektów), która pozwoli na wygodne kontrolowanie natężenia oświetlenia otoczenia:

   a) W sekcji `__published` definicji klasy `TGLForm` (w pliku nagłówkowym *GLForm.h*) dodajemy definicję własności oraz deklarację metody zmieniającej natężenia oświetlenia:

```
__published:
   __property float NatezenieSwiatlaOtoczenia =
      {read=natezenie_swiatla_otoczenia,write=UstawNatezenieSwiatlaOtoczenia};
private:
   void fastcall UstawNatezenieSwiatlaOtoczenia(float NatezenieSwiatlaOtoczenia);
```

   b) W pliku *GLForm.cpp* definiujemy funkcję `UstawNatezenieSwiatlaOtoczenia` (listing 60).

   Listing 60.

```
void __fastcall TGLForm::UstawNatezenieSwiatlaOtoczenia(float NatezenieSwiatlaOtoczenia)
{
   if (NatezenieSwiatlaOtoczenia<0.0f) NatezenieSwiatlaOtoczenia=0.0f;
   if (NatezenieSwiatlaOtoczenia>1.0f) NatezenieSwiatlaOtoczenia=1.0f;
   this->natezenie_swiatla_otoczenia=NatezenieSwiatlaOtoczenia;
   if(debug_mode) Caption=(AnsiString)"Natezenie oswietlenia otoczenia: "
      +(int)(100.0*this->natezenie_swiatla_otoczenia)+"%";
   GL_Oswietlenie();
   GL_RysujScene();
}
```

   c) W konstruktorze klasy TForm1 możemy umieścić polecenie zmieniające oświetlenie tła np. `NatezenieSwiatlaOtoczenia=0.5f;`.

6. Aby umożliwić w naszej aplikacji kontrolę natężenia światła otoczenia za pomocą klawiszy + i -, dodajemy do metody `TForm1::FormKeyDown` linie wyróżnione w listingu 61.

Listing 61. Funkcja GL_Oswietlenie wywoływana jest również w przypadku zmiany wartości pola kontrolującego natężenie oświetlenia tła

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
   //obroty
   if (Shift.Empty())
   {
      switch (Key)
      {
         case VK_LEFT:   Phi-=3; break;
         case VK_RIGHT:  Phi+=3; break;
         case VK_UP :    Theta-=3; break;
         case VK_DOWN:   Theta+=3; break;

         case 'q':
         case 'Q':
            Timer1->Enabled=!Timer1->Enabled;
            break;

         case VK_OEM_MINUS:
            NatezenieSwiatlaOtoczenia-=0.01;
            break;
         case VK_OEM_PLUS:
         case '=':
            NatezenieSwiatlaOtoczenia+=0.01;
            break;
      }
   }

//dalsza czesc metody
```

Metoda `Oswietlenie` ma ułatwić definiowanie źródeł oświetlenia w klasie potomnej klasy `TGLForm`. W odróżnieniu od metody `RysujScene`, której sposób użycia jest podobny, nie jest czysto wirtualna. Nie ma więc

konieczności jej definiowania w klasie potomnej. Warto to zrobić dopiero w momencie definiowania własnych źródeł oświetlenia.

Po uruchomieniu aplikacji zobaczymy jedynie ciemnoszary sześcian. Jeżeli wzmocnimy światło do maksimum (należy przytrzymać klawisz +, ale bez *Shift*), sześcian rozjaśni się. Jeżeli usunęlibyśmy warunek ograniczający wartość pola `natezenie_swiatla_otoczenia`, to przy wzmocnieniu znacznie przewyższającym wartość `1.0` zobaczymy, że nasz ostrosłup jest cały biały. Ściślej, że jest w kolorze światła tła zdefiniowanego funkcją `glLightModelfv`. Jeżeli tablica `kolor_otoczenia` definiowałaby kolor czerwony — ostrosłup byłby czerwony.

6.2 Uzgadnianie koloru „fizycznego” przedmiotów z kolorem ustalanym funkcją glColor

Aby przywrócić stan sprzed włączenia oświetlenia, musimy mieć dwie rzeczy — białe światło otoczenia i materiał, którego własności rozpraszające są zdefiniowane w sposób odpowiadający wybranym wcześniej kolorom. Domyślne ustawienia materiału są takie, że jednakowo odbijają wszystkie składowe światła otoczenia (tzn. R, G i B) ze współczynnikami równymi `0.2`. Można to jednak zmienić. I to na dwa sposoby. Po pierwsze, możemy zdefiniować własności poszczególnych ścian funkcją `glMaterial`. To wymagałoby zmiany w kodzie, polegającej na zastąpieniu wszystkich wywołań funkcji `glColor` funkcją `glMaterial`. Aby tego uniknąć, można wymusić, żeby własności materiału były zgodne z barwą nadaną funkcją `glColor`. I tym właśnie zajmiemy się teraz.

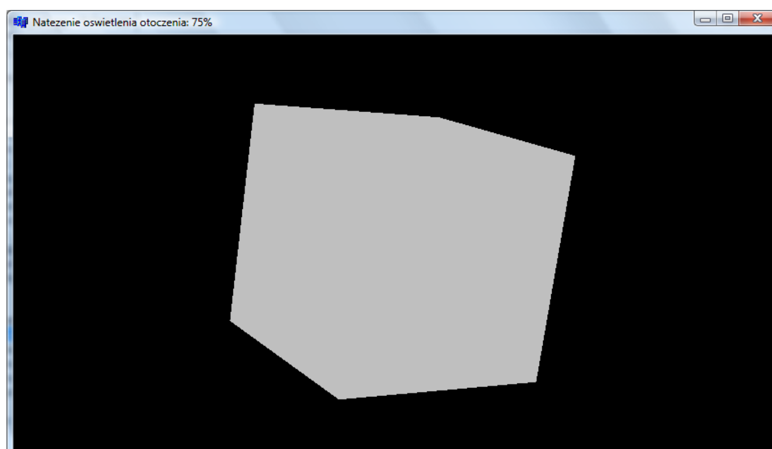
Do metody `GL_Oswietlenia` dodajemy dwa polecenia wyróżnione w listingu 62.

Listing 62. Włączanie mechanizmu uzgadniania dla światła tła i światła rozproszonego

```
void __fastcall TGLForm::GL_Oswietlenie()
{
   const float kolor_otoczenie[]={natezenie_swiatla_otoczenia,
                                  natezenie_swiatla_otoczenia,
                                  natezenie_swiatla_otoczenia}; //biel
   glEnable(GL_LIGHTING); //wlaczenie systemu oswietlania

   glEnable(GL_COLOR_MATERIAL);
   glColorMaterial(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);

   glLightModelfv(GL_LIGHT_MODEL_AMBIENT,kolor_otoczenie); //swiatlo tla

   Oswietlenie();
}
```

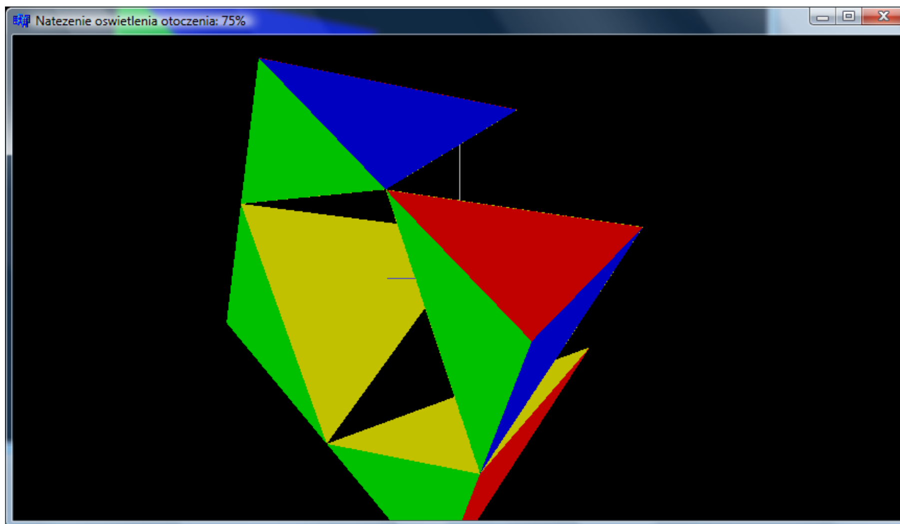
Przyciemnione światło tła (rysunek 15) pozwoli lepiej zobaczyć efekt dodatkowego źródła rozproszonego światła, które zaraz włączymy. Gdybyśmy światło otoczenia (kontrolowane klawiszami + i -) ustawili na maksimum (wszystkie składowe równe `1.0`) — trudniej byłoby to zobaczyć. Podobnie słabo widoczne jest światło żarówki włączonej w ciągu jasnego letniego dnia.

Rysunek 15. *Przytłumione oświetlenie tła pozwoli na lepsze dostrzeżenie kolejnych źródeł światła*



Światło tła ma jednak tę ważną wadę, że nie uwidaczniają trójwymiarowości bryły. Jest tak dlatego, że światło to nie pada z określonego punktu (kierunku), a jednorodnie rozświetla całą scenę.

Możemy także na chwilę przywrócić ostrosłupy (metoda TForm1::RysujScene), żeby zobaczyć w jaki sposób ostatnie polecenia wpływają na rendering kolorowych powierzchni (rysunek 16).



Rysunek 16. Poszarzałe kolory w przyciemnionym świetle

Polecenia dodane do metody GL_Oswietlenie włączają mechanizm uzgadniania koloru materiału (własności powierzchni, z jakich zbudowane są znajdujące się na scenie przedmioty) z kolorem ustalanym funkcją glColor. To pozwoliło nam znów zobaczyć nasze ostrosłupy. Zwróćmy uwagę na argumenty polecenia glColorMaterial, a szczególnie na drugi z nich. Określa on, jakie własności materiału (reakcja na jakie typy światła) mają być identyczne z bieżącym kolorem. Mamy do wyboru światło otoczenia (GL_AMBIENT), światło rozproszone (GL_DIFFUSE, np. mleczna żarówka) lub światło ukierunkowane (GL_SPECULAR, np. reflektor lub słońce na bezchmurnym niebie). My zażądaliśmy, aby OpenGL uzgadniał z kolorem ustalonym funkcją glColor sposób odbijania światła rozproszonego i tła (stała GL_AMBIENT_AND_DIFFUSE). Ustawienia te mają dotyczyć obu stron figur przedniej i tylnej, o czym informuje pierwszy argument. W przypadku światła otoczenia, które jest w tej chwili jedynym, jakie oświetla scenę, nie ma to wielkiego znaczenia. Nabierze go, gdy zdefiniujemy światło rozproszone i kierunkowe.

6.3 Ilu programistów potrzeba, aby wkręcić mleczną żarówkę

Zdefiniujmy źródło światła rozproszonego. Umieścimy je po lewej stronie sceny. Będzie więc świecić z tego samego miejsca, w którym znajduje się kamera, o ile polecenia ustalające to źródło światła umieścimy przed jakąkolwiek modyfikacją macierzy model-widok (np. w metodzie GL_Oswietlenie).

W ogólności źródłem światła można poruszać na scenie na podobnych zasadach, jak porusza się aktorami. Ich ruchem względem obiektów na scenie i względem kamery rządzą zasady analogiczne do przedstawionych na schemacie w tabeli 11.2. Jeżeli polecenia ustalające pozycję światła znajdą się przed poleceniem obrotu — światło pozostanie nieruchome, w przeciwnym wypadku dotyczyć go będą przekształcenia macierzy model-widok, tak samo jak przedmiotów na scenie.

1. Definiujemy metodę MlecznaZarowka (listing 63) aktywującą źródło światła rozproszonego identyfikowane przez stałą GL_LIGHT1 (jedno z ośmiu możliwych źródeł światła):

Listing 63. Metoda włączająca i konfigurująca rozproszone źródło światła ustawione na 50 procent mocy

```
void __fastcall TForm1::MlecznaZarowka()
{
   const float kolor1_rozproszone[]={0.5,0.5,0.5,1.0};
   glLightfv(GL_LIGHT1,GL_DIFFUSE,kolor1_rozproszone);
   glEnable(GL_LIGHT1);
}
```

2. Deklarujemy ją w pliku nagłówkowym, w definicji klasy.

3. Wywołanie metody MlecznaZarowka umieszczamy w nadpisanej metodzie Oswietlenie (listing 64).

Listing 64. Światło jest nieruchome — jest ostatecznie ustawiane tuż po inicjacji grafiki OpenGL i potem jego pozycja nie jest zmieniana

```
void __fastcall TForm1::Oswietlenie()
{
   MlecznaZarowka();
}
```

Po uruchomieniu aplikacji zobaczymy sześcian, którego jasność zmienia się cyklicznie wraz z obrotami względem kamery. Nie oznacza to jednak wcale, że jaśniejsza staje się ściana zwrócona w kierunku źródła światła zamocowanego na kamerze, a ciemniejsze pozostałe. Wszystkie ściany sześcianu zmieniają jasność równocześnie i w jednakowy sposób. Wydaje się, że ściany nie potrafią w poprawny sposób identyfikować kierunku „do źródła światła".

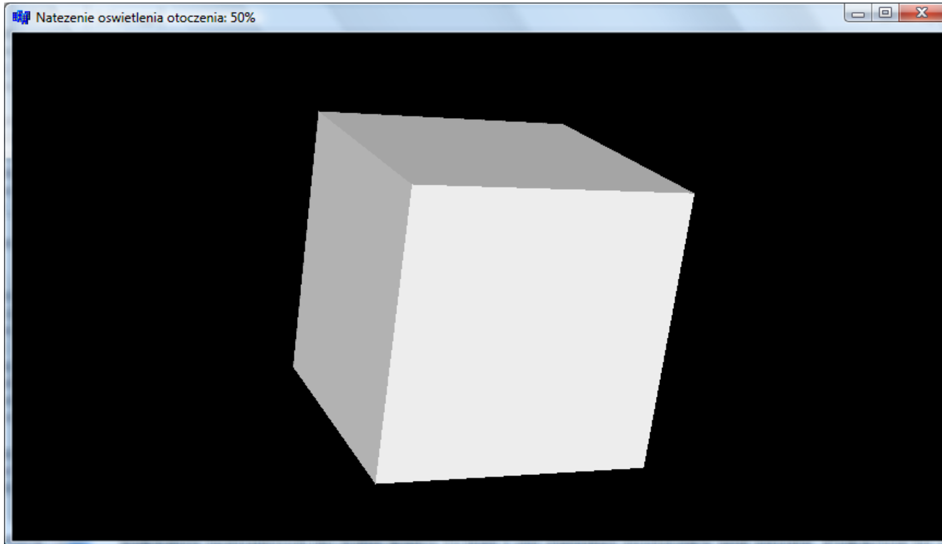
6.4 Definiowanie wektorów normalnych (sześcian)

Skoro ściany nie umieją same rozpoznać kierunków, musimy im w tym pomóc. Musimy każdej z powierzchni sześcianu wskazać kierunek prostopadły do niej, określając ich wektory normalne (najlepiej jednostkowe i skierowane „do przodu" kwadratu). Do ustalenia wektora normalnego służy funkcja glNormal. W przypadku sześcianu definiowanie wektorów normalnych jest bardzo proste – wszystkie ułożone są wzdłuż którejś z osi układu współrzędnych. Pokazuje to listing 65. Po dodaniu tych sześciu instrukcji do metody RysujSzescian renderowany obraz wreszcie nabiera „trójwymiarowości" (rysunek 17).

Listing 65.

```
void __fastcall TForm1::RysujSzescian(float krawedz) const
{
   const float a=krawedz;

   glBegin(GL_QUADS);
   //tylnia
   glNormal3f(0,0,-1); //wektory normalne skierowane na zewnatrz
   glVertex3f(-a,-a,-a);
   glVertex3f(-a,a,-a);
   glVertex3f(a,a,-a);
   glVertex3f(a,-a,-a);
   //przednia
   glNormal3f(0,0,1);
   glVertex3f(-a,-a,a);
   glVertex3f(a,-a,a);
   glVertex3f(a,a,a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(-a,a,a);

   //prawa
   glNormal3f(1,0,0);
   glVertex3f(a,-a,a);
   glVertex3f(a,a,a);
   glVertex3f(a,a,-a);
   glVertex3f(a,-a,-a);
   //lewa
   glNormal3f(-1,0,0);
   glVertex3f(-a,-a,a);
   glVertex3f(-a,a,a);
   glVertex3f(-a,a,-a);
   glVertex3f(-a,-a,-a);

   //gorna
   glNormal3f(0,1,0);
   glVertex3f(-a,a,a);
   glVertex3f(a,a,a);
   glVertex3f(a,a,-a);
   glVertex3f(-a,a,-a);
   //dolna
   glNormal3f(0,-1,0);
   glVertex3f(-a,-a,a);
   glVertex3f(a,-a,a);
```

```
      glVertex3f(a,-a,-a);
      glVertex3f(-a,-a,-a);
      //

      glEnd();
   }
```



Rysunek 17. Cienie na figurze nadają sześcianowi trójwymiarowości

6.5 Definiowanie wektorów normalnych (ostrosłup)

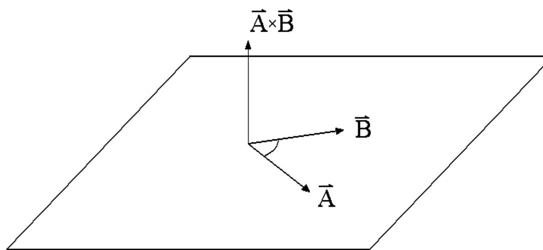
W przypadku ostrosłupa proste jest tylko zdefiniowanie wektorów normalnych do żółtej ściany tylnej i do zielonej podstawy. Pokazuje to listing 66. Pierwszy z wektorów ma kierunek –*z*, drugi –*y*:

Listing 66. Ustalanie normalnych dla ścian żółtej i zielonej

```
void __fastcall TForm1::RysujOstroslup(float x0,float y0,float z0) const
{
   //Rysowanie trojkata
   glBegin(GL_TRIANGLES);
   //ustalanie trzech wierzcholkow trojkata (werteksow (x,y,z))
   //(0,0,z) jest mniej wiecej w srodku ekranu

   //tylna
   glColor3ub(255,255,0); //zolty
   glNormal3f(0,0,-1.0); //w glab
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, y0, z0); //gorny

   //podstawa
   glColor3ub(0,255,0); //zielony
   glNormal3f(0,-1.0,0); //do dolu
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, -y0, 2*z0); //dolny przedni
//dalsza czesc metody
```

Z pozostałymi niestety nie będzie tak łatwo. Konieczne będzie przypomnienie sobie wiedzy o wektorach, a w szczególności o iloczynie wektorowym. Iloczyn wektorowy dwóch wektorów jest prostopadły do każdego z nich. Zatem jeżeli te dwa wektory opisują boki trójkąta, to uzyskujemy wektor prostopadły do jego powierzchni (rysunek 18), a więc dokładnie to, o co nam chodzi. To, w którą stronę wektor ten powinien być skierowany, zależy od nawinięcia trójkąta. Od tego powinna zależeć kolejność wektorów w iloczynie.

Rysunek 18.
Iloczyn wektorowy. W razie kłopotów z matematyką potrzebną do programowania grafiki trójwymiarowej polecam stronę MathWorld[15]. Tam są wszystkie potrzebne wzory i wyjaśnienia



Do wyznaczania wektorów normalnych przygotujemy funkcję, która wraz z funkcjami pomocniczymi umieszczona zostanie w osobnym module *GLWektory.cpp*/*GLWektory.h*. Polecenia umieszczone w tych funkcjach powinny być w miarę zrozumiałe dla osób, które wiedzą, czym jest iloczyn wektorowy. Pozostała część Czytelników skazana jest na korzystanie z nich „na wiarę" lub intensywny kurs geometrii analitycznej:

1. Z menu *File/New* wybieramy *Unit — C++Builder*.
2. Naciskamy klawisze *Ctrl+S*, aby zachować nowy moduł w pliku. Wybierzmy dla niego nazwę *GLWektory.cpp*.
3. W pliku *GLWektory.cpp* definiujemy funkcję obliczającą iloczyn wektorowy wraz z funkcjami pomocniczymi (listing 67).

Listing 67. Pełny kod modułu z funkcją obliczającą iloczyn wektorowy

```
//---------------------------------------------------------------------------


#pragma hdrstop

#include "GLWektory.h"

//---------------------------------------------------------------------------

#pragma package(smart_init)

float* Roznica3fv(float punkt1[3],float punkt2[3],float wynik[3])
{
   for(int i=0;i<3;i++) wynik[i]=punkt2[i]-punkt1[i];
   return wynik;
}

float* IloczynWektorowy3fv(float a[3],float b[3],float wynik[3])
{
   const int x=0;
   const int y=1;
   const int z=2;

   wynik[x]=  a[y]*b[z]-a[z]*b[y];
   wynik[y]=-(a[x]*b[z]-a[z]*b[x]);
   wynik[z]=  a[x]*b[y]-a[y]*b[x];
   return wynik;
}

#define SQR(x) ((x)*(x))
#include <math.h>  //sqrt

float* NormujWektor3fv(float wektor[3])
{
   float wsp=0;
   for(int i=0;i<3;i++) wsp+=SQR(wektor[i]);
   wsp=sqrt(wsp);
   for(int i=0;i<3;i++) wektor[i]/=wsp;
   return wektor;
}

float* JednostkowyWektorNormalny3fv(float punkt1[3],float punkt2[3],float
punkt3[3],float wynik[3])
{
   float wektor12[3],wektor13[3];
```

[15] Adres: *http://mathworld.wolfram.com/CrossProduct.html*

```
   return NormujWektor3fv(IloczynWektorowy3fv(Roznica3fv(punkt1,punkt2,wektor12),
      Roznica3fv(punkt1,punkt3,wektor13),wynik));
}
```

4. W pliku nagłówkowym *GLWektory.h* deklarujemy funkcję `JednostkowyWektorNormalny3fv` (listing 68), która samodzielnie będzie tworzyła interfejs modułu.

Listing 68. Tylko jedna funkcja modułu udostępniana jest „na zewnątrz”

```
//---------------------------------------------------------------------------

#ifndef GLWektoryH
#define GLWektoryH
//---------------------------------------------------------------------------

float* JednostkowyWektorNormalny3fv(float punkt1[3],float punkt2[3],
   float punkt3[3],float wynik[3]);

#endif
```

5. Wracamy do modułu formy *Unit1.cpp* i umieszczamy w nim dyrektywę importującą nagłówek *GLWektory.h*: `#include "GLWektory.h"`.

6. I wreszcie modyfikujemy metodę `RysujOstroslup` zgodnie ze wzorem widocznym na listingu 69:

Listing 69. Ustalamy normalne do pozostałych dwóch trójkątów: niebieskiego i czerwonego

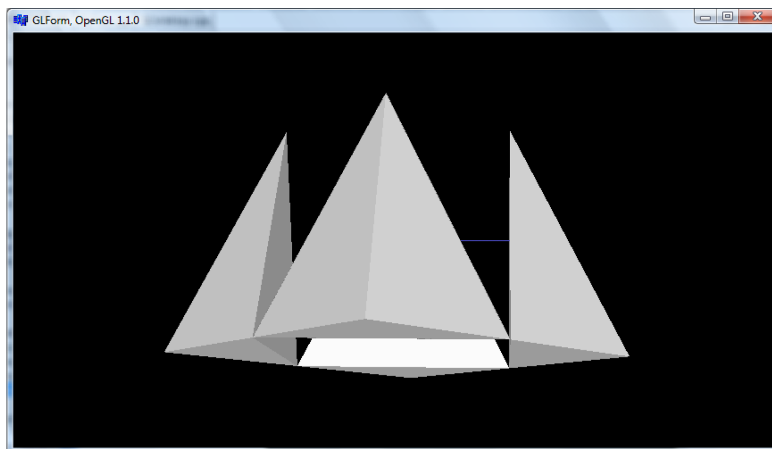
```
void __fastcall TForm1::RysujOstroslup(float x0,float y0,float z0) const
{
   const bool kolory=false;

   //Rysowanie trojkata
   glBegin(GL_TRIANGLES);
   //ustalanie trzech wierzcholkow trojkata (werteksow (x,y,z))
   //(0,0,z) jest mniej wiecej w srodku ekranu

   //tylna
   if(kolory) glColor3ub(255,255,0); //zolty
   glNormal3f(0,0,-1.0); //w glab
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, y0, z0); //gorny

   //podstawa
   if(kolory) glColor3ub(0,255,0); //zielony
   glNormal3f(0,-1.0,0); //do dolu
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, -y0, 2*z0); //dolny przedni

   //lewa
   if(kolory) glColor3ub(255,0,0); //czerwony
   float punktL1[3]={-x0,-y0,z0};
   float punktL2[3]={0,-y0,2*z0};
   float punktL3[3]={0,y0,z0};
   float normalnaL[3];
   JednostkowyWektorNormalny3fv(punktL1,punktL2,punktL3,normalnaL);
   glNormal3fv(normalnaL);
   glVertex3fv(punktL1); //dolny lewy
   glVertex3fv(punktL2); //dolny przedni
   glVertex3fv(punktL3); //gorny

   //prawa
   if(kolory) glColor3ub(0,0,255); //niebieski
   float punktR1[3]={x0, -y0, z0}; //dolny prawy
   float punktR2[3]={0, -y0, 2*z0}; //dolny przedni
   float punktR3[3]={0, y0, z0}; //gorny
   //odwrocone - aby nawijanie bylo prawidlowe (przy def. trojkata tez zmiana
kolejnosci)
   float normalnaR[3];
   JednostkowyWektorNormalny3fv(punktR1,punktR3,punktR2,normalnaR);
   glNormal3fv(normalnaR);
   glVertex3fv(punktR1); //dolny lewy
   glVertex3fv(punktR3); //dolny przedni
   glVertex3fv(punktR2); //gorny
```

```
        //koniec rysowania figury
        glEnd();
    }
```

Korzystając z tego, że i tak musieliśmy zdefiniować trójelementowe tablice przechowujące współrzędne wierzchołków trójkątów, zmieniliśmy także wersję użytej funkcji `glVertex` na `glVertex3fv`, która jako argument pobiera macierz trójelementową. Nie powtarzamy w ten sposób wpisywania współrzędnych jako argumentów tej funkcji.

Teraz gdy któraś płaszczyzna jest prostopadła do kamery, czyli gdy wektor normalny wskazuje źródło światła — płaszczyzna ta staje się najjaśniejsza. Gdy kąt między wektorem normalnym a kierunkiem, z którego pada wirtualne promieniowanie, rośnie — płaszczyzna ciemnieje. Dzięki temu, nawet przy wyłączonych kolorach, figury sprawiają wrażenie przestrzennych (zob. rysunek 19).

Rysunek 19. *Oświetlenie najlepiej badać przy wyłączonych kolorach (klawisz E); dla oka kolor niebieski jest bowiem ciemniejszy od czerwonego i zielonego*

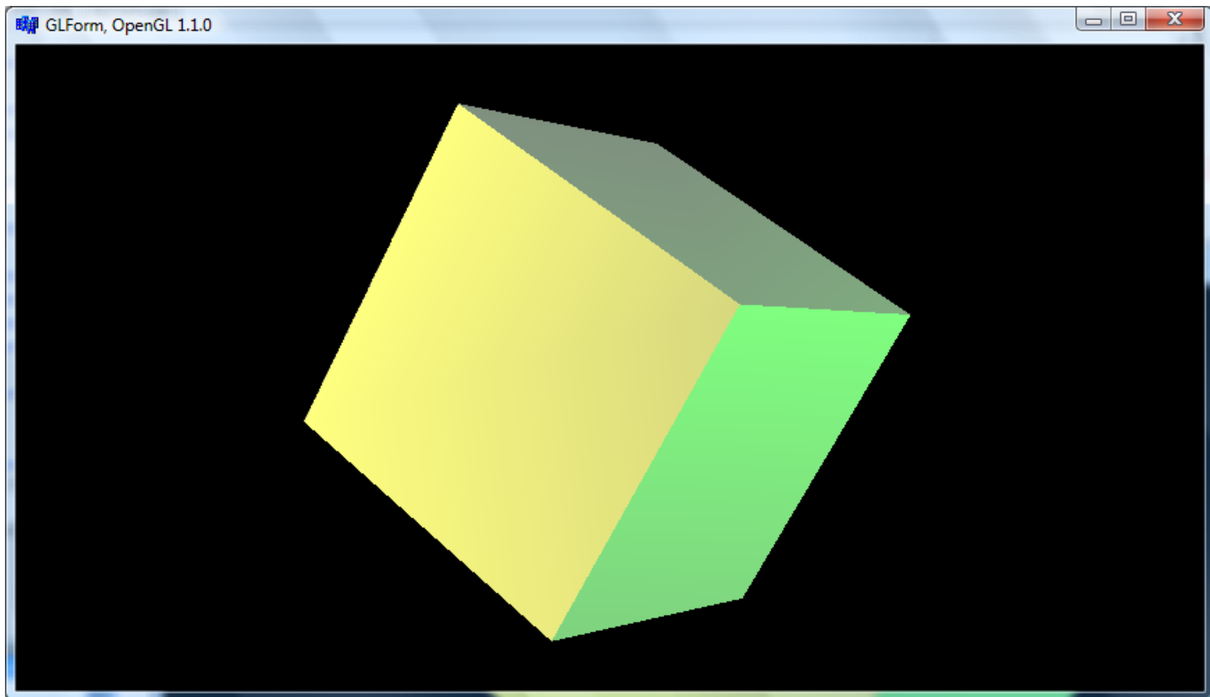


6.6 Barwne światło rozproszone

Światło nie musi być oczywiście białe. Dla przykładu zdefiniujemy metodę `ZoltaIZielonaMleczneZarowki` włączając w niej dwa źródła światła: żółte po lewej stronie sceny i zielone po prawej (listing 70). Efekt oświetlania nimi białego sześcianu widoczny jest na rysunku 20.

Listing 70.

```
void __fastcall TForm1::ZoltaIZielonaMleczneZarowki()
{
    //zolta mleczna zarowka
    const float kolor_rozproszone_zolta[4]={1.0f,1.0f,0.0f,1.0f};
    const float pozycja_zolta[4]={-2.0f,0.0f,1.0f,1.0f};

    glLightfv(GL_LIGHT2,GL_POSITION,pozycja_zolta);
    glLightfv(GL_LIGHT2,GL_DIFFUSE,kolor_rozproszone_zolta);
    glEnable(GL_LIGHT2);

    //zielona mleczna zarowka
    const float kolor_rozproszone_zielony[4]={0.0f,1.0f,0.0f,1.0f};
    const float pozycja_zielony[4]={2.0f,0.0f,1.0f,1.0f};

    glLightfv(GL_LIGHT3,GL_POSITION,pozycja_zielony);
    glLightfv(GL_LIGHT3,GL_DIFFUSE,kolor_rozproszone_zielony);
    glEnable(GL_LIGHT3);
}
```



Rysunek 20. Biały sześcian oświetlony kolorowymi światłami

6.7 Uśrednianie normalnych

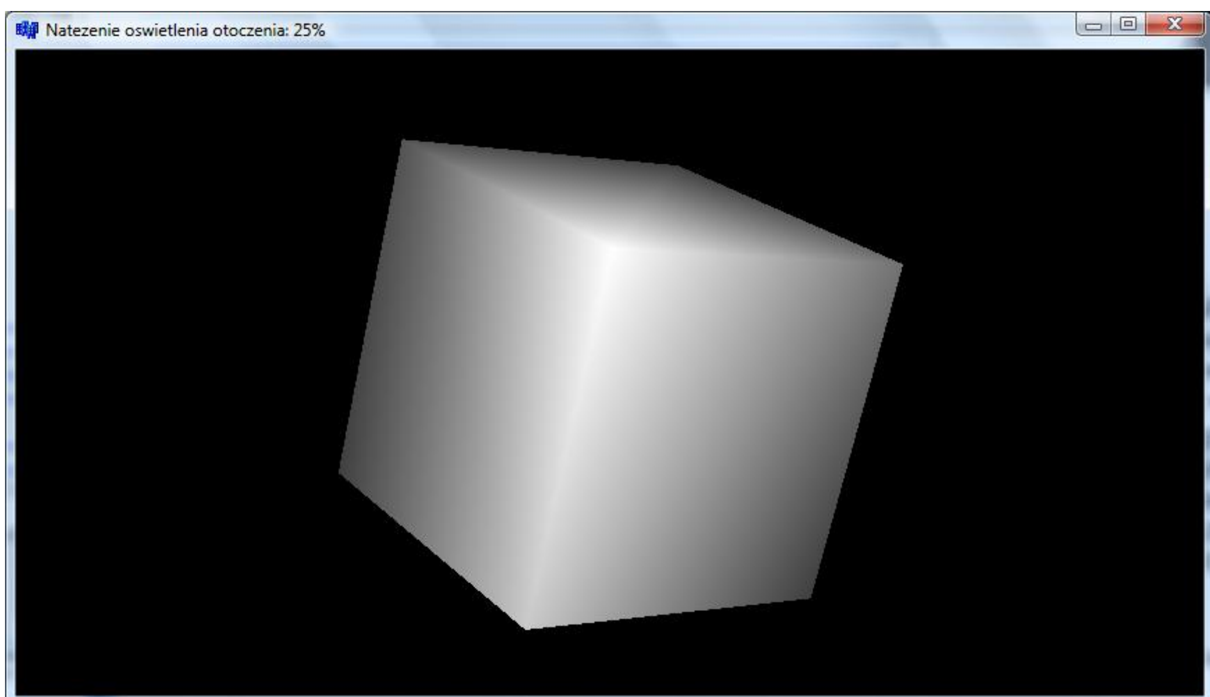
W dotychczasowych przykładach wywołanie funkcji glNormal umieszczaliśmy przed każdą trójką funkcji glVertex w przypadku trójkątów i przed każdą czwórką w przypadku prostokątów. To oczywiste, bo każdy trójkąt lub kwadrat może mieć tylko jedną normalną. A jednak nie jest to takie oczywiste w OpenGL. Normalną można bowiem ustalić osobno dla każdego wierzchołka figury. Efekt będzie podobny jak w przypadku cieniowania kolorów, a więc płynna zmiana wektora normalnego. Pozwala to uzyskać na płaskiej powierzchni efekt zaokrąglenia. Można również wygładzić w ten sposób kanciaste obiekty, kompensując małą liczbę trójkątów, z których są zbudowane. Ta ostatnia technika nazywa się uśrednianiem normalnych. Po obliczeniu wektorów prostopadłych do każdego trójkąta, każdemu wierzchołkowi przypisujemy wektor, który jest średnią wektorów normalnych wszystkich trójkątów spotykających się w tym punkcie. Efekt jest bardzo dobry — „kanty” znikają.

Tą techniką nie zmienimy sześcianu w kulę, ale warto zobaczyć jak zmieni ona sposób jego renderowania. Zdefiniujmy nową metodę RysujSzescian_UsrednianieNormalnych i skopiujmy do niej zawartość metody RysujSzescian. Następnie przed definicją każdego werteksu dodajmy wywołanie funkcji określającą normalną skierowaną w kierunku od środka układu współrzędnych (środek sześcianu) do wierzchołka wyznaczanego przez werteks.

Listing 71.

```
void __fastcall TForm1::RysujSzescian_UsrednianieNormalnych(float krawedz) const
{
   const float a=krawedz;

   glBegin(GL_QUADS);
   //tylnia
   glNormal3f(-1,-1,-1); glVertex3f(-a,-a,-a);
   glNormal3f(-1,1,-1); glVertex3f(-a,a,-a);
   glNormal3f(1,1,-1); glVertex3f(a,a,-a);
   glNormal3f(1,-1,-1); glVertex3f(a,-a,-a);
   //przednia
   glNormal3f(-1,-1,1); glVertex3f(-a,-a,a);
   glNormal3f(1,-1,1); glVertex3f(a,-a,a);
   glNormal3f(1,1,1); glVertex3f(a,a,a);
   glNormal3f(-1,1,1); glVertex3f(-a,a,a);

   //prawa
   glNormal3f(1,-1,1); glVertex3f(a,-a,a);
   glNormal3f(1,1,1); glVertex3f(a,a,a);
```

```
        glNormal3f(1,1,-1); glVertex3f(a,a,-a);
        glNormal3f(1,-1,-1); glVertex3f(a,-a,-a);
        //lewa
        glNormal3f(-1,-1,1); glVertex3f(-a,-a,a);
        glNormal3f(-1,1,1); glVertex3f(-a,a,a);
        glNormal3f(-1,1,-1); glVertex3f(-a,a,-a);
        glNormal3f(-1,-1,-1); glVertex3f(-a,-a,-a);

        //gorna
        glNormal3f(-1,1,1); glVertex3f(-a,a,a);
        glNormal3f(1,1,1); glVertex3f(a,a,a);
        glNormal3f(1,1,-1); glVertex3f(a,a,-a);
        glNormal3f(-1,1,-1); glVertex3f(-a,a,-a);
        //dolna
        glNormal3f(-1,-1,1); glVertex3f(-a,-a,a);
        glNormal3f(1,-1,1); glVertex3f(a,-a,a);
        glNormal3f(1,-1,-1); glVertex3f(a,-a,-a);
        glNormal3f(-1,-1,-1); glVertex3f(-a,-a,-a);
        //

        glEnd();
    }
```



Rysunek 21. Sześcian z uśrednionymi normalnymi w obecności jednego źródła światła rozproszonego związanego z kamerą

6.8 Gładkie materiały (rozbłysk)

Powróćmy do wyświetlania sześcianu bez uśrednionych normalnych.

Dodatkową zaletą zdefiniowania wektorów normalnych jest możliwość imitowania rozbłysków, jakie towarzyszą odbijaniu od gładkiej powierzchni silnego źródła światła (np. utworzony za pomocą lusterka „zajączek", który jakiś dowcipniś skieruje w nasze oczy). Własność ta nie musi dotyczyć wszystkich powierzchni. Na początek zastosujmy ją do powierzchni jednej ściany sześcianu (lewej). Dzięki temu będzie zachowywała się jak zbudowana ze szkła lub wypolerowanego metalu. Efekt działa tak, że ściana, która ustawiona jest w taki sposób, że promienie biegnące ze źródła światła typu reflektor odbite są od niej dokładnie w kierunku kamery, robi się coraz bielsza (w zależności od niewielkiego odchylenia). Jednym słowem, zobaczymy „prawdziwy" rozbłysk. Tę własność nadaje powierzchni polecenie:

```
glMaterialfv(GL_FRONT,GL_SPECULAR,wsp_odbicia_szklo);
```

w którym ostatni argument to tablica zawierająca współczynniki odbicia (my wszystkie ustawimy na 1.0). Chcąc zachować matowy wygląd kolejnych ścian, po zdefiniowaniu „szklanej” ściany wywołujemy tę funkcję jeszcze raz z tablicą o zerowych elementach.

Jeżeli zamiast jednego kwadratu będziemy mieli sferę zbudowaną z „wypolerowanych” trójkątów, to efekt rozbłysku będzie widoczny jako jasna plama na powierzchni sfery. Wielkość tej plamy ustalana jest parametrem, który może mieć wartości od 0 do 128. My użyjemy wartości 100:

```
glMateriali(GL_FRONT,GL_SHININESS,100);
```

W przypadku sześcianu po narysowaniu gładkiego kwadratu i ten parametr należy zmienić na 0.
Listing 72 zawiera metodę RysujScene, a dokładnie jej fragment w którym przed narysowaniem sześcianu włączana jest własność materiału nadająca powierzchniom charakter lśniącego materiały. Po narysowaniu sześcianu efekt jest wyłączany. Ponieważ rozbłysk jest zawsze biały, aby efekt był dobrze widoczny pomalowałem sześcian błękitem. Aby ten efekt zobaczyć, musimy jeszcze zdefiniować silne kierunkowe źródło światła (reflektor), czym zajmiemy się już w następnym ćwiczeniu.

Listing 72. Efekt będzie widoczny dopiero ustawieniu na scenie reflektora

```
void __fastcall TForm1::RysujScene()
{
   //obroty kontrolowane klawiszami
   glRotatef(Phi, 0.0, 1.0, 0.0); //wokol OY
   glRotatef(Theta, 1.0, 0.0, 0.0); //wokol OX

   //przesuniecia kontrolowane klawiszami
   glTranslatef(PozycjaX,PozycjaY,PozycjaZ);

   glPushMatrix();

   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   const float wsp_odbicia_szklo[4]={1.0,1.0,1.0,1.0};
   const float wsp_odbicia_matowy[4]={0.0,0.0,0.0,1.0};

   glMaterialfv(GL_FRONT,GL_SPECULAR,wsp_odbicia_szklo);
   glMateriali(GL_FRONT,GL_SHININESS,100);

   glColor3ub(0,0,150);
   RysujSzescian(x0);

   glMaterialfv(GL_FRONT,GL_SPECULAR,wsp_odbicia_matowy);
   glMateriali(GL_FRONT,GL_SHININESS,0);

   glPopMatrix();
}
```

6.9 Ustawianie reflektora

Umieszczenie na scenie reflektorów znacznie poprawia „przestrzenność” figur — cienie są wówczas silniej zarysowane niż w przypadku światła rozproszonego. Ponadto tego typu światło jest warunkiem zobaczenia zdefiniowanego w poprzednim ćwiczeniu rozbłysku. Dodamy nowe źródło światła, które będzie miało charakter reflektora, a które umieścimy pod sceną i skierujemy do góry tak, żeby oświetlało nasze ostrosłupy.

1. Zdefiniujmy metodę Reflektor (listing 73), w której zdefiniujemy drugie źródło światła (pamiętajmy o deklaracji tej metody w sekcji private klasy TForm1).

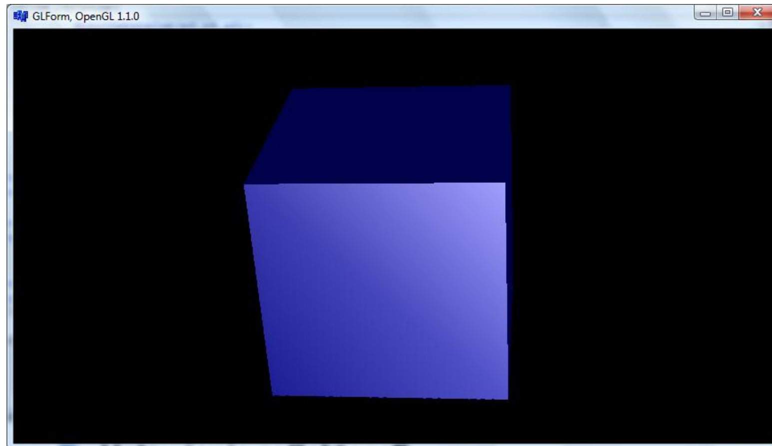
Listing 73. Metoda ustalająca parametry drugiego źródła światła

```
void __fastcall TForm1::Reflektor()
{
   const float kolor_rozproszone[4]={0.3,0.3,0.3,1.0};
   const float kolor_reflektora[4]={1.0,1.0,1.0,1.0};
   const float pozycja[4]={0.0,-10.0,10.0,1.0};
   const szerokosc_wiazki=60.0; //w stopniach
   glLightfv(GL_LIGHT4,GL_POSITION,pozycja);
   glLightfv(GL_LIGHT4,GL_DIFFUSE,kolor_rozproszone);
```

```
      glLightfv(GL_LIGHT4,GL_SPECULAR,kolor_reflektora);
      glLightf(GL_LIGHT4,GL_SPOT_CUTOFF,szerokosc_wiazki);
      glEnable(GL_LIGHT4);
   }
```

2. Dodajemy jej wywołanie do metody `TForm1::Oswietlenie` komentując w niej równocześnie pozostałe źródła światła.

Nowe źródło światła składa się z silnej części generującej światło odbite na obiektach i trzykrotnie słabszej części generującej światło rozproszone. Reflektor ustawiliśmy „pod kamerą", tj. w punkcie `(0,-10,10)`, i skierowaliśmy na środek układu współrzędnych (domyślne ustawienie). Szerokość smugi światła to aż 120 stopni (maksymalnie odchylony promień biegnie pod kątem 60 stopni od kierunku głównego). W efekcie na ekranie zobaczymy niebieską metalową kostkę (rysunek 22).



Rysunek 22. *Wyłączone światło przy kamerze pozwala lepiej zobaczyć, jaki efekt daje oświetlenie sceny reflektorem od dołu*

Ponieważ podstawowe źródło światła (`GL_LIGHT1`) jest dość silne, „zgasiliśmy je" (zakomentowane wywołanie metod `MlecznaZarowka` i `ZoltaIZielonaMleczneZarowki`), pozostawiając jedynie reflektor i oświetlenie tła. Eksperymentując z położeniem figur (klawisze sterowania kursorem lub myszka), sprawdźmy, czy i przy jakim ustawieniu uda nam się zrobić „zajączka" do kamery dowolną ze ścian sześcianu.

Realizm sceny znacznie wzmocniłyby cienie rzucane przez figurę. Oczywiście niezbędne byłoby zdefiniowanie jakiegoś podłoża, na którym cienie byłyby widoczne. OpenGL nie obejmuje jednak takich możliwości. Należałoby samodzielnie zdefiniować odpowiednie funkcje rzutujące figurę na podłoże (zobacz osobny skrypt dotyczący cieni).

6.10 Montowanie włączników światła

W metodzie TForm1::Oswietlenie wywołajmy wszystkie trzy metody definiujące źródła światła. Natomiast do metody `FormKeyDown` pełniącej funkcję pulpitu kontrolnego aplikacji (oczywiście częściowo rolę to pełni także metoda `TGLForm::WndProc`) dodajemy polecenia, które montują włączniki czterech zdefiniowanych źródeł światła (listing 74). Będziemy mogli je kontrolować klawiszami od *1* do *4*:

Listing 74. Montowanie na pulpicie kontrolnym włączników dla oświetleniowca

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
   //obroty
   if (Shift.Empty())
   {
      if (Key>='0' && Key<='7')
      {
         GLenum swiatlo=GL_LIGHT0;
         switch (Key)
         {
            case '1': swiatlo=GL_LIGHT1; break;
            case '2': swiatlo=GL_LIGHT2; break;
            case '3': swiatlo=GL_LIGHT3; break;
            case '4': swiatlo=GL_LIGHT4; break;
```

```
            case '5': swiatlo=GL_LIGHT5; break;
            case '6': swiatlo=GL_LIGHT6; break;
            case '7': swiatlo=GL_LIGHT7; break;
            default: swiatlo=GL_LIGHT0;
         }
         if (glIsEnabled(swiatlo)) glDisable(swiatlo);
         else glEnable(swiatlo);
      }

      switch (Key)
      {
         case VK_LEFT:   Phi-=3; break;
         case VK_RIGHT:  Phi+=3; break;
         case VK_UP :    Theta-=3; break;
         case VK_DOWN:   Theta+=3; break;

//dalsza czesc metody
```

6.11 Nieruchomy pokój

Ciekawy efekt daje umieszczenie całej sceny w „pudle”. Rozmiar pudła, a przynajmniej jego szerokość i wysokość, powinny być mniejsze lub równe rozmiarowi frustum. Wstawianie sceny do pudła ma oczywiście sens tylko, gdy w jego wnętrzu zdefiniowane są jakieś kierunkowe źródła światła. W innym przypadku nie będzie widoczne lub tworzyło jednolite tło.

1. Definiujemy funkcję `Pudlo`:

Listing 75.

```
void __fastcall TGLForm::Pudlo(float krawedz,float glebokosc)
{
   glBegin(GL_QUADS);
   //tylnia
   glNormal3f(0,0,1);
   glVertex3f(-krawedz,-krawedz,-glebokosc);
   glVertex3f(-krawedz,krawedz,-glebokosc);
   glVertex3f(krawedz,krawedz,-glebokosc);
   glVertex3f(krawedz,-krawedz,-glebokosc);
   //lewa
   glNormal3f(1,0,0);
   glVertex3f(-krawedz,-krawedz,-glebokosc);
   glVertex3f(-krawedz,krawedz,-glebokosc);
   glVertex3f(-krawedz,krawedz,0);
   glVertex3f(-krawedz,-krawedz,0);
   //prawa
   glNormal3f(0,0,-1);
   glVertex3f(krawedz,-krawedz,-glebokosc);
   glVertex3f(krawedz,krawedz,-glebokosc);
   glVertex3f(krawedz,krawedz,0);
   glVertex3f(krawedz,-krawedz,0);
   //gorna
   glNormal3f(0,-1,0);
   glVertex3f(-krawedz,krawedz,-glebokosc);
   glVertex3f(krawedz,krawedz,-glebokosc);
   glVertex3f(krawedz,krawedz,0);
   glVertex3f(-krawedz,krawedz,0);
   //dolna
   glNormal3f(0,1,0);
   glVertex3f(-krawedz,-krawedz,-glebokosc);
   glVertex3f(krawedz,-krawedz,-glebokosc);
   glVertex3f(krawedz,-krawedz,0);
   glVertex3f(-krawedz,-krawedz,0);
   glEnd();
}
```

2. Do klasy dodajemy prywatne pola kontrolujące wyświetlanie równoległościanu: `pudlo` typu `bool` i `pudlo_kolor` typu `TColor`. Pierwsze inicjujemy wartością `false`, a drugie `clBlue`.

3. Do metody `GL_RysujScene`, **przed** poleceniami ustawiającymi kamerę dodajemy:

Listing 76.

```
void __fastcall TGLForm::GL_RysujScene()
{
   //Przygotowanie bufora
```

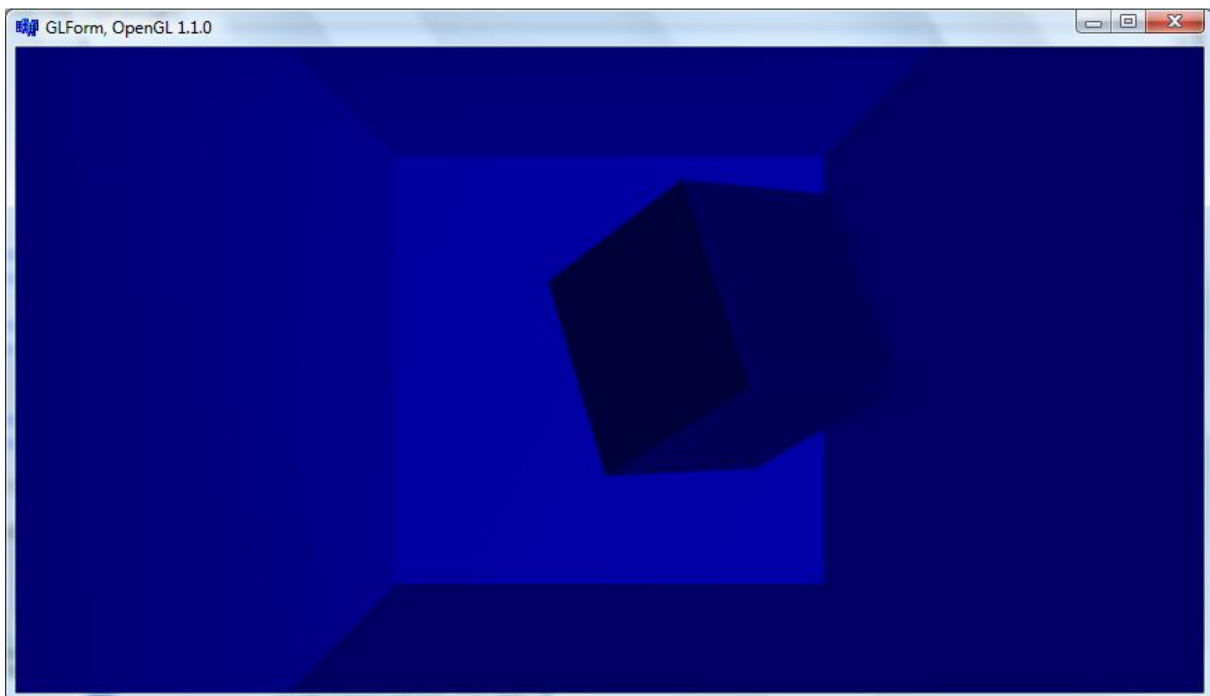
```
glClearColor(GetRValue(kolorTla)/255.0f,GetGValue(kolorTla)/255.0f,GetBValue(kolorTla)/2
55.0f,0.0);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa

   //nieruchome pudlo
   if (pudlo)
   {
      const float glebokosc=24.9f;
      const float krawedz=3.0f;
      glColor3f(GetRValue(pudlo_kolor)/255.0f,
                GetGValue(pudlo_kolor)/255.0f,
                GetBValue(pudlo_kolor)/255.0f);
      Pudlo(krawedz,glebokosc);
   }
```

1. Na koniec w klasie `TGLForm` definiujemy własność `NieruchomyPokoj`:

```
class TGLForm : public TForm
{
__published:
   __property TColor KolorTla = {read=kolorTla,write=UstawKolorTla};
   __property float NatezenieSwiatlaOtoczenia =
            {read=natezenie_swiatla_otoczenia,write=UstawNatezenieSwiatlaOtoczenia};
   __property bool NieruchomyPokoj = {read=pudlo,write=pudlo};
```

2. Do konstruktora klasy `TForm1` dodajemy polecenie `NieruchomyPokoj=true;`.
3. Warto również zamontować włącznik w metodzie `TForm1::FormKeyDown`:

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{

   ...

      switch (Key)
      {

   ...

         case 'n':
         case 'N':
            NieruchomyPokoj=!NieruchomyPokoj;
            break;
```



Rysunek 23. Scena zbudowana z niebieskich płyt

Dokładniejszą kontrolę wilkości pudła oraz jego koloru za pomocą własności zrealizujemy nieco później tworząc szablon projektu OpenGL.

7. Mieszanie kolorów (alpha blending)

Kolejne możliwości OpenGL, o których chciałbym teraz powiedzieć kilka słów, również wiążą się ze światłem. Tym razem chodzi o możliwość mieszania promieni światła biegnących od przedmiotów do kamery. Brzmi to enigmatycznie, ale jak się zaraz okaże, jest dość proste. A efekty uzyskane w ten sposób są bardzo ciekawe.

7.1 Inicjacja mieszania kolorów

Pierwszym zastosowaniem techniki mieszania kolorów jest efekt przezroczystości. Jeżeli dwa trójkąty znajdują się jeden za drugim, to ze względu na działanie bufora głębi ta część tylnego trójkąta, która znajduje się za bliższym, będzie na ekranie niewidoczna. Jednak jeżeli do koloru każdego punktu bliższego trójkąta dodamy ze współczynnikiem nazywanym zwykle **alfa** kolor odpowiednich punktów tylnego trójkąta, to będziemy mogli zobaczyć kontury dalszego trójkąta, mimo że będzie on znajdował się za bliższym trójkątem, ale jeżeli pierwszy trójkąt będzie na przykład czerwony, to tylny zostanie „zabarwiony" na czerwono. Jednym słowem, całość będzie sprawiała wrażenie, jakby bliższy trójkąt zbudowany był z czerwonego szkła.

Kolory mogą być mieszane za pomocą różnych równań. Najbardziej oczywiste jest mieszanie liniowe, w którym wszystkie trzy składowe koloru, jaki widzimy, mieszane są z wagami wyznaczanymi przez współczynniki alfa. Im mniej przejrzysty jest przedni trójkąt (jego współczynnik alfa ma większą wartość), tym bardziej w uzyskanym kolorze dominuje kolor przedniego trójkąta. Stosując tę samą zasadę, można także mieszać kolory wielu figur.

Przezroczystość powierzchni powoduje osłabienie niektórych efektów związanych z oświetleniem, np. rozbłysków odbitego światła.

1. W metodzie `GL_Oswietlenie` klasy `TGLForm` (listing 77) włączamy system mieszania kolorów i ustalamy równanie mieszania (które i bez tego byłoby równaniem domyślnym).

Listing 77. Inicjacja mechanizmu mieszania kolorów

```
void __fastcall TGLForm::GL_Oswietlenie()
{
   const float kolor_otoczenie[]={natezenie_swiatla_otoczenia,
                                  natezenie_swiatla_otoczenia,
                                  natezenie_swiatla_otoczenia}; //biel-odcienie szarości
   glEnable(GL_LIGHTING); //wlaczenie systemu oswietlania

   glEnable(GL_COLOR_MATERIAL);
   glColorMaterial(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);

   glLightModelfv(GL_LIGHT_MODEL_AMBIENT,kolor_otoczenie); //swiatlo tla

   Oswietlenie();

   //mieszanie kolorow
   glEnable(GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
}
```

Ponadto warto w metodzie `WndProc` umieścić ****

7.2 Przezroczystość

Teraz możemy uczynić nasz obiekt przezroczystym, a dokładnie jedną z jego ścian. Ściana ta musi być rysowana jako ostatnia (bez względu na działanie bufora głębokości), co pozwoli na ominięcie problemu kolejności rysowania. W związku z tym na okno wybieramy dolną ścianę naszego sześcianu.

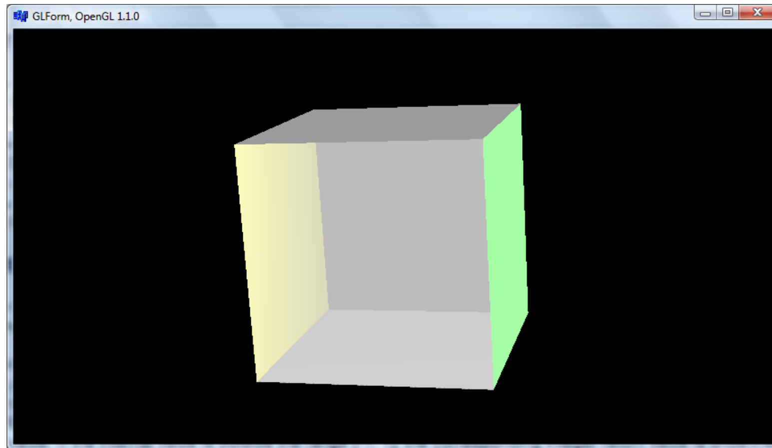
1. Modyfikujemy metodę `RysujSzescian` dodając do niej argument pozwalający na wybór koloru całego sześcianu (listing 78). Pamiętajmy, aby zmienić jej deklarację w pliku nagłówkowym.

Listing 78. Okna musimy rysować na końcu

```
void __fastcall TForm1::RysujSzescian(float krawedz,TColor kolor) const
{
   const float a=krawedz;
   int R=GetRValue(kolor);
   int G=GetGValue(kolor);
   int B=GetBValue(kolor);
   glColor4ub(R,G,B,255);

   glBegin(GL_QUADS);
   //tylnia
   glNormal3f(0,0,-1); //wektory normalne skierowane na zewnatrz
   glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,-a,-a);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(-a,a,-a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(a,a,-a);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(a,-a,-a);
   //przednia
   glNormal3f(0,0,1);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(-a,-a,a);
   glTexCoord2f(1.0f, 1.0f); glVertex3f(a,-a,a);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(a,a,a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(-a,a,a);

   //prawa
   glNormal3f(1,0,0);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(a,-a,a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(a,a,a);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(a,a,-a);
   glTexCoord2f(1.0f, 1.0f); glVertex3f(a,-a,-a);
   //lewa
   glNormal3f(-1,0,0);
   glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,-a,a);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(-a,a,a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(-a,a,-a);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(-a,-a,-a);

   //gorna
   glNormal3f(0,1,0);
   glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,a,a);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(a,a,a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(a,a,-a);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(-a,a,-a);
   //dolna
   glColor4ub(R,G,B,127);
   glNormal3f(0,-1,0);
   glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,-a,a);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(a,-a,a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(a,-a,-a);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(-a,-a,-a);
   glColor4ub(R,G,B,255);
   //

   glEnd();
}
```

2. Wywołujemy zmodyfikowaną funkcję poleceniem `RysujSzescian(x0,clWhite);` i ustawiamy myszą tak, żeby zobaczyć podstawę (rysunek 24), a przez nią wnętrze sześcianu.

Rysunek 24. *Ściana zwrócona do kamery jest częściowo przezroczysta*

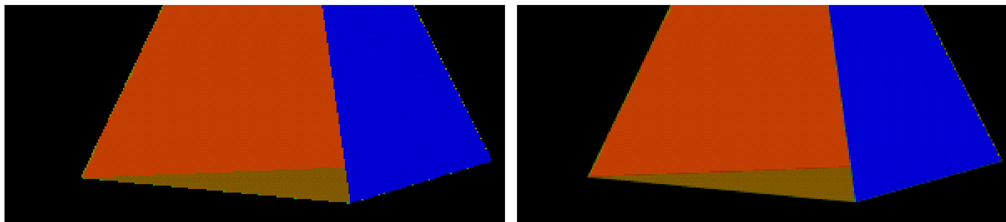


Przeniesienie rysowania elementów przezroczystych na koniec jest bardzo ważne. Rysując przezroczystą ścianę, biblioteka OpenGL chce dodać do niej kolory przedmiotów znajdujących się za nią — muszą się one wobec tego już znajdować na ekranie. W przypadku większej ilości obiektów półprzezroczystych konieczne jest sortowanie rysowanych przedmiotów ze względu na ich aktualne położenie względem kamery.

Samo mieszanie kolorów nie da nigdy efektu szklanego materiału. W rzeczywistości ten efekt powstaje w materiałach, które są niemal całkowicie przezroczyste, ale oko dostrzega niedoskonałości sceny widzianej za oknem (załamania) i odbicia widoczne na szybie. W ten sam sposób można uzyskać bardziej realistyczne wrażenie szkła w OpenGL.

7.3 Antyaliasing

Innym zastosowaniem mieszania kolorów jest antyaliasing. Wyobraźmy sobie krawędź dwóch ścian (np. niebieskiej i czerwonej w naszym ostrosłupie), która biegnie ukośnie. Ze względu na dyskretny charakter kart graficznych i monitorów na linii tej krawędzi widoczne będą schodki (zob. rysunek 25, lewy). Jednak jeżeli w pobliżu krawędzi wymieszamy oba kolory (dodając do koloru każdego piksela kolory jego kilku sąsiadów), to schodki znikną tak, jakbyśmy w tym miejscu zmniejszyli ostrość obrazu (por. rysunek 25, prawy). To jest właśnie antyaliasing.



Rysunek 25. *Przez mieszanie kolorów antyaliasing wygładza krawędzie. Problem widoczny jest najbardziej w starszych wersjach OpenGL (np. 1.0 i 1.1); w nowszych stosowane są dodatkowe mechanizmy optymalizujące wygląd ukośnych linii*

Do metody `TForm1::FormKeyDown` dodajmy polecenia inicjujące mechanizm antyaliasingu zgodnie ze wzorem na listingu 79. Jego włączanie i wyłączanie będzie możliwe za pomocą klawisza *A*. Przy okazji dodajemy polecenia montujące włącznik całego mechanizmu mieszania kolorów (klawisz *B*).

Listing 79. Montowanie włącznika antyaliasingu

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
   //obroty
   if (Shift.Empty())
   {
      if (Key>='0' && Key<='7')
      {
         GLenum swiatlo=GL_LIGHT0;
```

```
            switch (Key)
            {
               case '1': swiatlo=GL_LIGHT1; break;
               case '2': swiatlo=GL_LIGHT2; break;
               case '3': swiatlo=GL_LIGHT3; break;
               case '4': swiatlo=GL_LIGHT4; break;
               case '5': swiatlo=GL_LIGHT5; break;
               case '6': swiatlo=GL_LIGHT6; break;
               case '7': swiatlo=GL_LIGHT7; break;
               default: swiatlo=GL_LIGHT0;
            }
            if (glIsEnabled(swiatlo)) glDisable(swiatlo);
            else glEnable(swiatlo);
         }

         switch (Key)
         {
            case VK_LEFT:   Phi-=3; break;
            case VK_RIGHT:  Phi+=3; break;
            case VK_UP :    Theta-=3; break;
            case VK_DOWN:   Theta+=3; break;

            case 'q':
            case 'Q':
               Timer1->Enabled=!Timer1->Enabled;
               break;

            case 'b':
            case 'B':
               if (glIsEnabled(GL_BLEND)) glDisable(GL_BLEND);
               else glEnable(GL_BLEND);
               break;
            case 'a':
            case 'A':
               if (glIsEnabled(GL_POINT_SMOOTH))
               {
                  glDisable(GL_POINT_SMOOTH);
                  glDisable(GL_LINE_SMOOTH);
                  glDisable(GL_POLYGON_SMOOTH);
               }
               else
               {
                  glEnable(GL_POINT_SMOOTH);
                  glEnable(GL_LINE_SMOOTH);
                  glEnable(GL_POLYGON_SMOOTH);
               }
               break;

            case VK_OEM_MINUS:
               NatezenieSwiatlaOtoczenia-=0.01;
               break;
            case VK_OEM_PLUS:
            case '=':
               NatezenieSwiatlaOtoczenia+=0.01;
               break;
         }
      }
```

Antyaliasing realizowany jest różnie w zależności od wersji OpenGL i typu karty graficznej. Można zasugerować, czy użyty algorytm antyaliasingu ma być zoptymalizowany ze względu na końcowy efekt (wówczas powinniśmy użyć polecenia `glHint(GL_LINE_SMOOTH_HINT,GL_NICEST);`), czy ze względu na szybkość działania (wówczas drugim argumentem powinno być `GL_FASTEST`). Aby przywrócić wartość domyślną, należy użyć stałej `GL_DONT_CARE`.

7.4 Mgła

Ostatnim efektem uzyskanym dzięki mieszaniu kolorów jest mgła. Jej uzyskanie jest dość proste do wyobrażenia — im dalej przedmiot znajduje się od kamery, tym większa porcja bieli dodawana jest do jego kolorów. Oglądany z bliska ma naturalne kolory, a z daleka jest szarawy. Podobnie wpływa na kolory także prawdziwa mgła.

1. Do metody `TGLForm::GL_Oswietlenie` dodajemy polecenia konfigurujące mgłę, ale nie włączające jej (listing 80).

Listing 80. Konfiguracja efektu mgły

```
void __fastcall TGLForm::GL_Oswietlenie()
{
   const float kolor_otoczenie[]={natezenie_swiatla_otoczenia,
                                  natezenie_swiatla_otoczenia,
                                  natezenie_swiatla_otoczenia}; //biel-odcienie szarości
   glEnable(GL_LIGHTING); //wlaczenie systemu oswietlania

   glEnable(GL_COLOR_MATERIAL);
   glColorMaterial(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);

   glLightModelfv(GL_LIGHT_MODEL_AMBIENT,kolor_otoczenie); //swiatlo tla

   Oswietlenie();

   //mieszanie kolorow
   glEnable(GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);

   //mgla
   //glEnable(GL_FOG);
   const float biel[4]={1.0,1.0,1.0,1.0};
   glFogfv(GL_FOG_COLOR,biel);
   glFogf(GL_FOG_START,0.0);
   glFogf(GL_FOG_END,100.0);
   glFogf(GL_FOG_MODE,GL_LINEAR);
}
```

2. W klasie `TGLForm` definiujemy własność `Mgla` typu `bool` i dwie związane z nią metody:

   Listing 81.

```
__published:
   __property bool Mgla = {read=CzyMgla,write=UstawMgla};
private:
   bool __fastcall CzyMgla();
   void __fastcall UstawMgla(bool mgla);
```

3. Następnie definiujemy obie metody:

   Listing 82.

```
bool __fastcall TGLForm::CzyMgla()
{
   return glIsEnabled(GL_FOG);
}

void __fastcall TGLForm::UstawMgla(bool mgla)
{
   if(mgla) glEnable(GL_FOG);
   else glDisable(GL_FOG);
   GL_RysujScene();
}
```

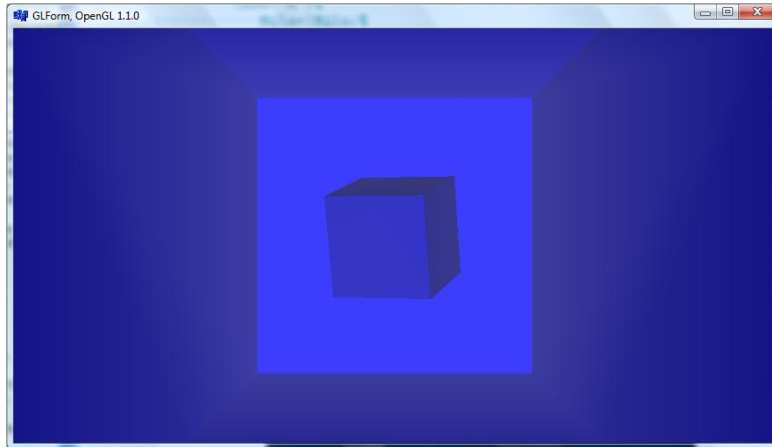
4. Teraz możemy w klasie `TForm1`, a dokładniej w metodzie `FormKeyDown` zamontować włącznik (listing 83).

Listing 83. I jeszcze jeden włącznik

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{

   ...

      switch (Key)
      {

   ...

         case 'm':
         case 'M':
            Mgla=!Mgla;
```

```
            break;
```

Działanie efektu mgły widoczne jest na rysunku 26 (nie jestem pewien, czy będzie widoczne na biało-czarnym wydruku) — po odsunięciu od kamery obiekt stał się bielszy. Im będzie dalej, tym bielsze będą kolory jego ścian.



Rysunek 26. *Im dalej od kamery znajduje się figura, tym bardziej jest „zamglona"*

8. Teksturowanie

8.1 Przygotowanie projektu do teksturowania:

W projekcie, który rozwijaliśmy do tej pory:

1. Włączmy światło rozproszone i reflektor wywołując z metody `TForm1::Oswietlenie` tylko metody `MlecznaZarowka` i `Reflektor`, a komentując wywołanie pozostałych.
2. Włączamy swobodne obroty umieszczając w konstruktorze klasy `TForm1` wywołanie metody `Obracaj`: `Obracaj(10.0f,1.0f,0.0f);` i przygaszamy światło otoczenia redukując je do 50% (listing 84). Wyłączamy też mgłę i rysowanie nieruchomego pokoju.

   Listing 84
   ```
   __fastcall TForm1::TForm1(TComponent* Owner)
           : TGLForm(Owner),
           Phi(0.0f), Theta(0.0f),
           PozycjaX(0.0f), PozycjaY(0.0f), PozycjaZ(0.0f)
   {
      KolorTla=clBlack;
      debug_mode=false;
      Mgla=false;
      NieruchomyPokoj=false;

      NatezenieSwiatlaOtoczenia=0.5f;
      Obracaj(10.0f,1.0f,0.0f);
   }
   ```

3. Wyświetlamy sześcian z metalicznym pobłyskiem tj. w metodzie `RysujScene` powinny znaleźć się polecenia z listingu 85.

   Listing 85.
   ```
   void __fastcall TForm1::RysujScene()
   {
      const float wsp_odbicia_szklo[4]={1.0,1.0,1.0,1.0};
      const float wsp_odbicia_matowy[4]={0.0,0.0,0.0,1.0};

      glMaterialfv(GL_FRONT,GL_SPECULAR,wsp_odbicia_szklo);
      glMateriali(GL_FRONT,GL_SHININESS,100);
   ```

```
    glColor4ub(255,255,255,255);
    RysujSzescian(1.0f);

    glMaterialfv(GL_FRONT,GL_SPECULAR,wsp_odbicia_matowy);
    glMateriali(GL_FRONT,GL_SHININESS,0);
}
```

Teraz możemy zabrać się za tapetowanie kostki.

Efekt „metalowej powierzchni” nie znalazł się tutaj bez powodu – teksturowanie zepsuje ten efekt i konieczne będą specjalne zabiegi, aby go przywrócić.

8.2 Wczytywanie obrazu tekstury

Stwórzmy osobny moduł o nazwie *Teksturowanie.h*/*Teksturowanie.cpp* i umieśćmy w pliku źródłowym poniższą funkcję (listing 86). Jej zadaniem jest wczytanie z podanego w argumencie pliku obrazu i zapisanie jej w tablicy całkowitych liczb 32-bitowych. Przez argumenty zwracane są natomiast rozmiary wczytanej tekstury. Funkcję należy oczywiście zadeklarować w pliku nagłówkowym.

Listing 86
```
unsigned long* __fastcall WczytajTeksture(AnsiString nazwaPliku,int& TeksturaSzer,int&
TeksturaWys)
{
   Graphics::TBitmap* obrazBMP=new Graphics::TBitmap();
   obrazBMP->PixelFormat=pf32bit;

   //Uwzgledniam czytanie z JPEG
   AnsiString rozszerzenie=ExtractFileExt(nazwaPliku);
   if (rozszerzenie==".jpg" || rozszerzenie==".jpe" || rozszerzenie==".jpeg")
   {
      //JPEG
      TJPEGImage* obrazJPEGtmp=new TJPEGImage();
      try
      {
         obrazJPEGtmp->LoadFromFile(nazwaPliku);
         obrazBMP->Width=obrazJPEGtmp->Width;
         obrazBMP->Height=obrazJPEGtmp->Height;
         obrazBMP->Canvas->Draw(0,0,obrazJPEGtmp);
      }
      catch(...)
      {
         ShowMessage("Brak pliku tekstury "+nazwaPliku);
         delete obrazJPEGtmp;
         delete obrazBMP;
         return NULL;
      }
      delete obrazJPEGtmp;
   }
   else
   {
      //BMP
      Graphics::TBitmap* obrazBMPtmp=new Graphics::TBitmap();
      try
      {
         //dodaje to kopiowanie obrazow, aby uniezaleznic sie od typu bitmapy
         obrazBMPtmp->LoadFromFile(nazwaPliku);
         obrazBMP->Width=obrazBMPtmp->Width;
         obrazBMP->Height=obrazBMPtmp->Height;
         obrazBMP->Canvas->Draw(0,0,obrazBMPtmp);
      }
      catch(...)
      {
         ShowMessage("Brak pliku tekstury "+nazwaPliku);
         delete obrazBMPtmp;
         delete obrazBMP;
         return NULL;
      }
      delete obrazBMPtmp;
   }

   //kopiowanie do dynamicznej tablicy RGBA
```

```
   TeksturaSzer=obrazBMP->Width;
   TeksturaWys=obrazBMP->Height;
   unsigned long* Tekstura=new unsigned long[TeksturaSzer*TeksturaWys];
   for(int ih=0;ih<TeksturaWys;ih++)
   {
      unsigned long* linia=(unsigned long*)obrazBMP->ScanLine[ih];
      for(int iw=0;iw<TeksturaSzer;iw++)
      {
         unsigned long c=*linia & 0x00FFFFFF;
         Tekstura[iw+(ih*TeksturaSzer)]=(((c & 0xFF) << 16)+(c >> 16)+ (c & 0xFF00)) |
0xFF000000; //podzial na 4 kanaly: RGBA
         linia++;
      }
   }
   delete obrazBMP;
   return Tekstura;
}
```

Obraz można odczytać z plików w formaci *.jpeg* i *.bmp*. W obu przypadkach zapisywane są do obiektów typu `TJPEGImage` i `Graphics::TBitmap` i kopiowane do `Graphics::TBitmap`. W drugim przypadku kopiowanie to może być zaskakujące, ale pozwala m.in. uniezależnić się od szczegółowego formatu bitmapy.

Oczywiście nie ma konieczności wczytywania bitmapy w trakcie działania programu. Można ją wczytać do komponentu `TImage` już w momencie projektowania aplikacji – będzie przechowana w pliku *.dfm*, a potem dołączona do pliku *.exe*. To zwiększy oczywiście plik *.exe*, ale nie będzie potrzeby osobnego przechowywania plików z obrazami.

8.3 Wiązanie tekstury

1. W pliku Unit1.cpp włączamy nagłówek nowego modułu dyrektywą: `#include "Teksturowanie.h"`.
2. Do konstruktora formy dodajemy polecenia ładujące teksturę:

Listing 87.

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TGLForm(Owner),
        Phi(0.0f), Theta(0.0f),
        PozycjaX(0.0f), PozycjaY(0.0f), PozycjaZ(0.0f)
{
   KolorTla=clBlack;
   debug_mode=false;
   Mgla=false;
   NieruchomyPokoj=false;

   NatezenieSwiatlaOtoczenia=0.5f;
   Obracaj(10.0f,1.0f,0.0f);

   //teksturowanie
   glEnable(GL_TEXTURE_2D);

   int TeksturaSzer,TeksturaWys;
   unsigned long* Tekstura=WczytajTeksture("tekstura.bmp",TeksturaSzer,TeksturaWys);
   if (Tekstura!=NULL)
   {
      glBindTexture(GL_TEXTURE_2D,Tekstura[0]);
      glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
      glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
      gluBuild2DMipmaps(GL_TEXTURE_2D,3,TeksturaSzer,TeksturaWys,GL_RGBA,
GL_UNSIGNED_BYTE, Tekstura);
      delete[] Tekstura; //oryginalne dane sa usuwane
   }
}
```

3. Aby tekstury były prawidłowo związane z sześcianem, musimy „przypiąć” je do każdej ze ścian:

Listing 88.

```
void __fastcall TForm1::Szescian(const float krawedz) //wartosc domyslna =1
{
   const float a=krawedz;

   glBegin(GL_QUADS);
   //tylnia
```

```
    glNormal3f(0,0,-1); //wektory normalne skierowane na zewnatrz
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,-a,-a);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-a,a,-a);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(a,a,-a);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(a,-a,-a);
    //przednia
    glNormal3f(0,0,1);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,-a,a);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(a,-a,a);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(a,a,a);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-a,a,a);

    //prawa
    glNormal3f(1,0,0);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(a,-a,a);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(a,a,a);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(a,a,-a);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(a,-a,-a);
    //lewa
    glNormal3f(-1,0,0);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,-a,a);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-a,a,a);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-a,a,-a);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-a,-a,-a);

    glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,a,a);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(a,a,a);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(a,a,-a);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-a,a,-a);
    //dolna
    glNormal3f(0,-1,0);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,-a,a);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(a,-a,a);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(a,-a,-a);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-a,-a,-a);
    //
    glEnd();
}
```

4. Aby uniknąć teksturowania nieruchomego pokoju i innych obiektów rysowanych poza metodą `RysujScene` należy zmodyfikować jej wywołanie w metodzie `TGLForm::GL_RysujScene` otaczając je wywołaniami funkcji włączających i wyłączających teksturowanie:

   Listing 89.

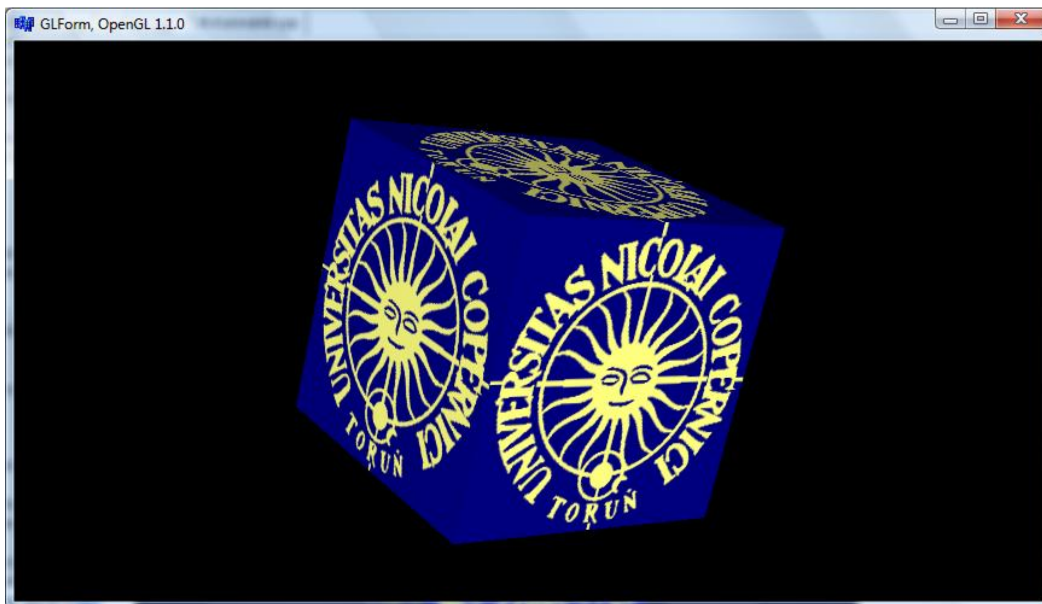
```
void __fastcall TGLForm::GL_RysujScene()
{
    //Przygotowanie bufora
    glClearColor(GetRValue(kolorTla)/255.0f,
                 GetGValue(kolorTla)/255.0f,
                 GetBValue(kolorTla)/255.0f,0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); //czysci bufory
    glLoadIdentity(); //macierz model-widok = macierz jednostkowa

    ...

    //przeksztalcenia macierzy model-widok i rysowanie figur
    glEnable(GL_TEXTURE_2D);
    RysujScene();
    glDisable(GL_TEXTURE_2D);

    ...
```

Przed uruchomieniem programu należy przygotować obraz tekstury. Może to być plik w formacie *.jpg* lub *.bmp*, najlepiej o równej wysokości i szerokości, zapisany w katalogu, w którym znalazł się skompilowany plik *.exe*. Jego nazwa powinna być wskazana w konstruktorze w jako pierwszy argument funkcji `WczytajTeksture`.

Zwróćmy uwagę na funkcję `gluBuild2DMipmaps`, która w konstruktorze klasy tworzy na podstawie oryginalnego obrazu tekstury serię tzw. mipmap tj. tekstur o różnych rozmiarach, które są wynikiem skalowania wyjściowej tekstury[16]. Dzięki temu do faktycznego teksturowania obiektu może być wybrany obraz, którego rozmiar najlepiej pasuje do aktualnego rozmiaru obiektu na scenie. W przypadku małych (dalszych) obiektów

[16] Zob. *http://pl.wikipedia.org/wiki/Mipmapping*

pozwala to na przyspieszenie renderowania, bo OpenGL korzysta z obrazów o mniejszej rozdzielczości. Ponadto technika mipmappingu wspomagana jest sprzętowo przez niektóre karty grafiki. Po utworzeniu mipmap oryginalny obraz może być usunięty z pamięci.
Rozmiar tekstury musi być dopasowany do teksturowanej powierzchni. Służy do tego funkcja `glTexCoord2f`, której argumentami są współrzędne wierzchołków naszych kwadratów we współrzędnych względnych. Każdy werteks powinien być poprzedzony wywołaniem tej funkcji, co pozwoli na określenie orientacji tekstury. Tekstura może być w dowolny sposób obrócona, przeskalowana lub pochylona. Jeżeli chcemy pokryć teksturą całą powierzchnię ścian sześcianu, to argumentami serii tych funkcji będą zera i jedynki.
Po uruchomieniu zobaczymy nowe tekstury – to dobra wiadomość. Mniej pomyślna jest ta, że tekstury zupełnie stłumiły efekt rozbłysków (choć są czułe na kierunek oświetlenia rozproszonego)[17].



Rysunek 27. Obiekt z teksturą.

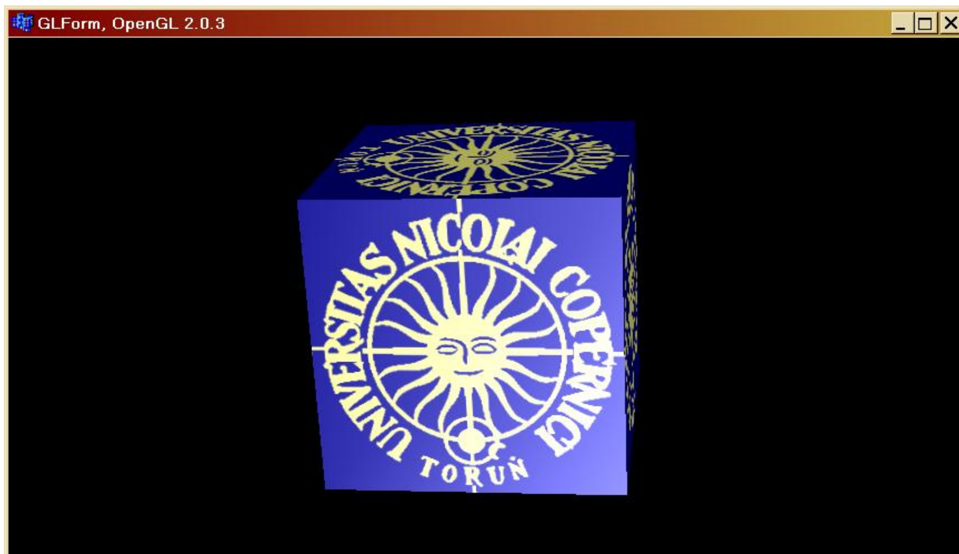
8.4 Lakierowanie tekstury

Aby przywrócić efekt gładkiej powierzchni musimy zastosować specjalny tryb oświetlenia, który dostępny jest w OpenGL 1.2 i nowszych. To oznacza, że może być niedostępny jeżeli korzystamy z bibliotek OpenGL wbudowanych w system Windows przez Microsoft (wersja 1.1). Z tego powodu w nagłówku *gl.h* używanym przez C++Builder niezdefiniowane mogą być potrzebne stałe. Możemy jednak zdefiniować je sami. Listing 90 przedstawia konieczne modyfikacje w metodzie `RysujScene`:

Listing 90.
```
void __fastcall TForm1::RysujScene()
{
   int GL_LIGHT_MODEL_COLOR_CONTROL=0x81F8,
   GL_SINGLE_COLOR=0x81F9,GL_COLOR_SIMPLE=0x81F9,
   GL_SEPARATE_SPECULAR_COLOR=0x81FA;

   const float wsp_odbicia_szklo[4]={1.0,1.0,1.0,1.0};
   const float wsp_odbicia_matowy[4]={0.0,0.0,0.0,1.0};

   glMaterialfv(GL_FRONT,GL_SPECULAR,wsp_odbicia_szklo);
   glMateriali(GL_FRONT,GL_SHININESS,100);
   glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL,GL_SEPARATE_SPECULAR_COLOR);   //konieczne,
aby tekstury nie psuly

   glColor4ub(255,255,255,255);
   RysujSzescian(1.0f);

   glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL,GL_SINGLE_COLOR);
   glMaterialfv(GL_FRONT,GL_SPECULAR,wsp_odbicia_matowy);
```

[17] Jeszcze mniej miła jest ta, że na niektórych implementacjach OpenGL, m.in. dołączanej do Windows, teksturowanie znacznie zwalnia renderowanie sceny.

```
    glMateriali(GL_FRONT,GL_SHININESS,0);
}
```



Rysunek 28. Lakierowana tekstura

8.5 Korzystanie z wielu tekstur

Oczywiście do renderowania obrazu możemy wykorzystać więcej niż jedną teksturę. Między innymi różnym ścianom sześcianu możemy przypisać różne tekstury. Konieczna jest oczywiście zmiana sposobu ładowania tekstur – teraz będziemy mieli ich tablicę.

1. W pliku nagłówkowym Unit1.h zdefiniujmy stałą TEKSTURY_ILOSC wskazującą ilość wykorzystywanych w programie tekstur:
   ```
   const int TEKSTURY_ILOSC=3;
   ```
2. w definicji klasy deklarujemy prywatną tablicę tablic tekstur
   ```
   GLuint Tekstury[TEKSTURY_ILOSC];
   ```
3. Zmieniamy sposób ładowania tekstur – tym razem ładujemy trzy tekstury, a po zmianie na tablice licz całkowitych ich adresy umieszczane są w tablicy `Tekstury`:

Listing 91.

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TGLForm(Owner)
{
   KolorTla=clBlack;
   debug_mode=false;
   Mgla=false;
   NieruchomyPokoj=false;

   NatezenieSwiatlaOtoczenia=0.5f;
   Obracaj(10.0f,1.0f,0.0f);

   //teksturowanie
   glEnable(GL_TEXTURE_2D);

   AnsiString
nazwyPlikow[TEKSTURY_ILOSC]={"tekstura1.bmp","tekstura2.bmp","tekstura3.bmp"};

   glGenTextures(TEKSTURY_ILOSC,Tekstury); //tworzenie tablicy tekstur
   for(int i=0;i<TEKSTURY_ILOSC;i++)
   {
      int TeksturaSzer,TeksturaWys;
      unsigned long* Tekstura=WczytajTeksture(nazwyPlikow[i],TeksturaSzer,TeksturaWys);
      if (Tekstura!=NULL)
      {
         glBindTexture(GL_TEXTURE_2D,Tekstury[i]);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
         gluBuild2DMipmaps(GL_TEXTURE_2D,3,TeksturaSzer,TeksturaWys,GL_RGBA,
GL_UNSIGNED_BYTE, Tekstura);
```

```
            delete[] Tekstura; //oryginalne dane sa usuwane
         }
      }
   }
```

4. Tym razem musimy przygotować trzy pliki, a ich nazwy wymienić w tablicy `nazwyPlikow` zdefiniowanej w konstruktorze.
5. Przed zamknięciem programu powinniśmy usunąć tablicę tekstur z pamięci:

Listing 92.

```
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
   glDeleteTextures(TEKSTURY_ILOSC,Tekstury);
}
```

6. Pozostaje tylko poprzypinać tekstury do odpowiednich ścian sześcianu. Każda tektura wymaga osobnego `glBegin..glEnd`.

Listing 93.

```
void __fastcall TForm1::RysujSzescian(float krawedz) const
{
   const float a=krawedz;

   glBindTexture(GL_TEXTURE_2D,Tekstury[0]);
   glBegin(GL_QUADS);
   //tylnia
   glNormal3f(0,0,-1); //wektory normalne skierowane na zewnatrz
   glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,-a,-a);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(-a,a,-a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(a,a,-a);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(a,-a,-a);
   //przednia
   glNormal3f(0,0,1);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(-a,-a,a);
   glTexCoord2f(1.0f, 1.0f); glVertex3f(a,-a,a);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(a,a,a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(-a,a,a);
   glEnd();

   glBindTexture(GL_TEXTURE_2D,Tekstury[1]);
   glBegin(GL_QUADS);
   //prawa
   glNormal3f(1,0,0);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(a,-a,a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(a,a,a);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(a,a,-a);
   glTexCoord2f(1.0f, 1.0f); glVertex3f(a,-a,-a);
   //lewa
   glNormal3f(-1,0,0);
   glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,-a,a);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(-a,a,a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(-a,a,-a);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(-a,-a,-a);
   glEnd();

   glBindTexture(GL_TEXTURE_2D,Tekstury[2]);
   glBegin(GL_QUADS);
   //gorna
   glNormal3f(0,1,0);
   glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,a,a);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(a,a,a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(a,a,-a);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(-a,a,-a);
   //dolna
   glNormal3f(0,-1,0);
   glTexCoord2f(1.0f, 1.0f); glVertex3f(-a,-a,a);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(a,-a,a);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(a,-a,-a);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(-a,-a,-a);
   glEnd();
}
```

Rysunek 29. Sześcian pokryty wieloma teksturami

W osobnym skrypcie http://www.fizyka.umk.pl/~jacek/dydaktyka/3d/OpenGL_scr.pdf dostępny jest opis wygaszacza ekranu zbudowanego na bazie powyższego projektu.

9. Napisy

9.1 Czcionki bitmapowe

Umieszczanie na scenie OpenGL napisów nie jest proste. Do tego powinniśmy dbać również o poprawne wyświetlanie polskich czcionek. Wiąże się to niestety z przygotowaniem sporej ilości kodu, co oczywiście warto zrobić raz, a potem korzystać z gotowych funkcji.

1. Tworzymy nowy moduł o nazwie *GLNapisy.h*/*GLNapisy.cpp*.
2. W pliku nagłówkowym importujemy nagłówek *System.hpp* zawierającego definicję typu `AnsiString` oraz nagłówek biblioteki OpenGL. Deklarujemy także funkcje `StworzCzcionkeBitmapowa` i `Pisz` (listing 94).

Listing 94.
```
//---------------------------------------------------------------------------

#ifndef GLNapisyH
#define GLNapisyH

#include <System.hpp> //AnsiString
#include <gl.h>

//---------------------------------------------------------------------------

GLuint StworzCzcionkeBitmapowa(HWND Handle,AnsiString NazwaCzcionki,bool Pogrubiona,bool
Kursywa);
void Pisz(AnsiString napis,GLuint czcionka);

#endif
```

3. W pliku *GLNapisy.cpp* definiujemy funkcję `StworzCzcionkeBitmapowa` zgodnie ze wzorem z poniższego listingu:

Listing 95.

```
GLuint StworzCzcionkeBitmapowa(HWND Handle,AnsiString NazwaCzcionki,bool Pogrubiona,bool
Kursywa)
{
   HFONT font;      //uchwyt do czcionki
   HFONT tmp_font;  //pomocniczy

   GLuint podstawa = glGenLists(256);   //Tworzy liste na pelen zestaw czcionek

   //funkcja WinAPI (GDI); tworzy obiekt czcionki
   tmp_font = CreateFont(30, //wysokosc czcionki
                      0,  //uzywam szerokosci czcionki proporcjonalnej do wysokosci
                      0,  //nachylenie czcionek
                      0,  //kat
                      Pogrubiona?FW_BOLD:FALSE, //pogrubionie
                      Kursywa?TRUE:FALSE,       //kursywa
                      FALSE,                   //podkreslenie
                      FALSE,                   //przekreslenie
                      ANSI_CHARSET,            //zbior liter ANSI
                      OUT_TT_PRECIS,           //czcionki true type
                      CLIP_DEFAULT_PRECIS,     //domyslna precyzja przycinania rozmiaru
                      ANTIALIASED_QUALITY,     //antyaliasing przy tworzeniu obrazow
                      FF_DONTCARE|DEFAULT_PITCH,  //styl i dekoracja czcionki
                      NazwaCzcionki.c_str());     //nazwa czcionki

   HDC uchwytDC=GetDC(Handle);   //ten uchwyt jest w klasie TGLForm, ale prywatny
   font=(HFONT)SelectObject(uchwytDC,tmp_font);   //zwiazanie z naszym kontekstem okna
   //Funkcja specyficzna dla Windows; 32-spacja; ilosc: 96 - alfabet lacinski, 256 - gdy
tez polskie znaki
   wglUseFontBitmaps(uchwytDC,0,256,podstawa);    //buduje pelen zestaw 256 czcionek
bitmapowych
   SelectObject(uchwytDC,font);   //Wybor czcionki, ktora stworzylismy
   DeleteObject(tmp_font);        //Usuwanie pomocniczego

   return podstawa;
}
```

5. Umieszczamy tam także krótszą funkcję `Pisz`:

Listing 96.

```
GLvoid Pisz(AnsiString napis,GLuint czcionka)
{
   if (napis.IsEmpty()) return;

   glPushAttrib(GL_LIST_BIT);   //Odklada na stos atrybuty wyswietlania
   glListBase(czcionka - 0);    //Ustawia podstawe znakow na 32
   glCallLists(napis.Length(), GL_UNSIGNED_BYTE, napis.c_str());
                                //Wyswietla liste liter (napis)
   glPopAttrib();               //Przywraca ze stosu atrybuty wyswietlania
}
```

6. Tworzymy zestaw czcionek w momencie uruchomienia aplikacji:
   a) W pliku *Unit1.cpp* importujemy nagłówek modułu *GLNapisy*: `#include "GLNapisy.h"`.
   b) Definiujemy trzy prywatne pola klasy `TForm1`:

```
GLuint czcionkaArial,czcionkaCourier,czcionkaTimes;
```

   c) Inicjujemy je za pomocą funkcji `StworzCzcionkeBitmapowa` w konstruktorze:

Listing 97.

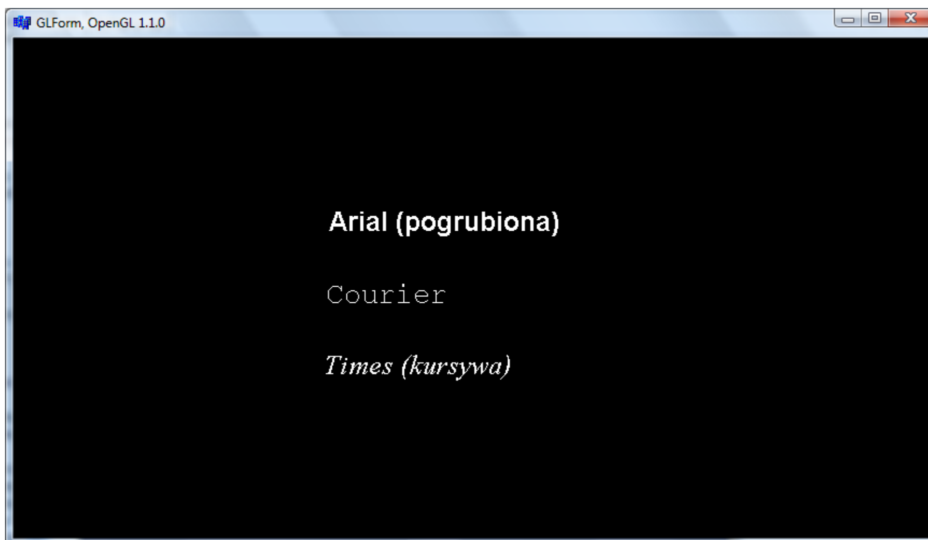
```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TGLForm(Owner)
{
   //CE = Central Europe
   czcionkaArial=StworzCzcionkeBitmapowa(Handle,"Arial CE",true,false);
   czcionkaCourier=StworzCzcionkeBitmapowa(Handle,"Courier New CE",false,false);
   czcionkaTimes=StworzCzcionkeBitmapowa(Handle,"Times New Roman CE",false,true);

   ...
```

   d) Musimy pamiętać o ich usunięciu przy zamykaniu aplikacji. Tworzymy do tego metodę zdarzeniową do `OnClose`:

Listing 98.
```
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
   glDeleteTextures(TEKSTURY_ILOSC,Tekstury);

   glDeleteLists(czcionkaArial,256);
   glDeleteLists(czcionkaCourier,256);
   glDeleteLists(czcionkaTimes,256);
}
```

7. Teraz możemy ich użyć w metodzie `RysujScene`:

Listing 99.
```
void __fastcall TForm1::RysujScene()
{
   glRasterPos3f(-1.0f,0.0f,0.0f);
   Pisz("Arial (pogrubiona)",czcionkaArial);

   glRasterPos3f(-1.0f,-0.5f,0.0f);
   Pisz("Courier",czcionkaCourier);

   glRasterPos3f(-1.0f,-1.0f,0.0f);
   Pisz("Times (kursywa)",czcionkaTimes);
}
```



Rysunek 30. To są napisy 2D, więc nie są obracane (ale poruszają się razem ze sceną)

9.2 Czcionki 3D

1. Do modułu *GLNapisy* dodajmy funkcję `StworzCzcionke3D` (listing 100) Różni się ona od poprzedniej tylko jednym poleceniem.

Listing 100. Wyróżnione różnice względem metody z listingu 95.
```
GLuint StworzCzcionke3D(HWND Handle,AnsiString NazwaCzcionki,bool Pogrubiona,bool Kursywa)
{
   HFONT font;      //uchwyt do czcionki
   HFONT tmp_font;  //pomocniczy

   GLuint podstawa = glGenLists(256);   //Tworzy liste na pelen zestaw czcionek

   //funkcja WinAPI (GDI); tworzy obiekt czcionki
   tmp_font = CreateFont(30, //wysokosc czcionki
                      0,  //uzywam szerokosci czcionki proporcjonalnej do wysokosci
                      0,  //nachylenie czcionek
                      0,  //kat
                      Pogrubiona?FW_BOLD:FALSE, //pogrubionie
                      Kursywa?TRUE:FALSE,       //kursywa
                      FALSE,                    //podkreslenie
                      FALSE,                    //przekreslenie
```

```
                                ANSI_CHARSET,          //zbior liter ANSI
                                OUT_TT_PRECIS,         //czcionki true type
                                CLIP_DEFAULT_PRECIS,   //domyslna precyzja przycinania rozmiaru
                                ANTIALIASED_QUALITY,   //antyaliasing przy tworzeniu obrazow
                                FF_DONTCARE|DEFAULT_PITCH,  //styl i dekoracja czcionki
                                NazwaCzcionki.c_str());     //nazwa czcionki

   HDC uchwytDC=GetDC(Handle); //ten uchwyt jest w klasie TGLForm, ale prywatny
   font=(HFONT)SelectObject(uchwytDC,tmp_font);          //zwiazanie z naszym kontekstem
okna
   GLYPHMETRICSFLOAT  gmf[256];  //informacje  o  rozmiarach  czcionek  (dalej  nie
wykorzystywane)
   wglUseFontOutlines(uchwytDC,0,256,podstawa,  //buduje pelen zestaw 256 czcionek 3D
                      0.0f, //precyzja budowania czcionki
                      0.2f, //glebokosc czcionki
                      WGL_FONT_POLYGONS,
                      gmf); //informacje o rozmiarach czcionek
   SelectObject(uchwytDC,font);   //Wybor czcionki, ktora stworzylismy
   DeleteObject(tmp_font);        //Usuwanie pomocniczego

   return podstawa;
}
```

2. W klasie `TForm1` deklarujemy pola, do których zapiszemy adresy czcionek 3D:
3. Stwórzmy zestaw takich czcionek. Ich tworzenie zajmuje zauważalnie więcej czasu. Przy okazji przyciemniamy światło otoczenia, aby zobaczyć trójwymiarowość nowych czcionek

```
GLuint czcionkaArial3D,czcionkaCourier3D,czcionkaTimes3D;
```

Listing 101.

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TGLForm(Owner)
{
   //CE = Central Europe
   czcionkaArial=StworzCzcionkeBitmapowa(Handle,"Arial CE",true,false);
   czcionkaCourier=StworzCzcionkeBitmapowa(Handle,"Courier New CE",false,false);
   czcionkaTimes=StworzCzcionkeBitmapowa(Handle,"Times New Roman CE",false,true);

   czcionkaArial3D=StworzCzcionke3D(Handle,"Arial CE",true,false);
   czcionkaCourier3D=StworzCzcionke3D(Handle,"Courier New CE",false,false);
   czcionkaTimes3D=StworzCzcionke3D(Handle,"Times New Roman CE",false,true);

   ...
```

4. Nie zapomnijmy o ich usunięciu w metodzie `FormClose`

   Listing 102.

```
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
   glDeleteTextures(TEKSTURY_ILOSC,Tekstury);

   glDeleteLists(czcionkaArial,256);
   glDeleteLists(czcionkaCourier,256);
   glDeleteLists(czcionkaTimes,256);

   glDeleteLists(czcionkaArial3D,256);
   glDeleteLists(czcionkaCourier3D,256);
   glDeleteLists(czcionkaTimes3D,256);
}
```

5. Pisanie tymi czcionkami możliwe jest za pomocą tej samej metody `Pisz`, którą zdefiniowaliśmy dla czcionek bitmapowych. W metodzie `RysujScene` z listingu 103 oprócz poleceń wyświetlających nowe czcionki dodane zostały polecenia zmieniające ich materiał na lśniący (por. listing 72).

Listing 103. Lśniące napisy 3D

```
void __fastcall TForm1::RysujScene()
{
   glColor3ub(255,255,255);
   glDisable(GL_TEXTURE_2D);

   glRasterPos3f(-1.0f,0.0f,0.0f); Pisz("Arial (pogrubiona)",czcionkaArial);
   glRasterPos3f(-1.0f,-0.5f,0.0f); Pisz("Courier",czcionkaCourier);
   glRasterPos3f(-1.0f,-1.0f,0.0f); Pisz("Times (kursywa)",czcionkaTimes);

   const float wsp_odbicia_szklo[4]={1.0,1.0,1.0,1.0};
   const float wsp_odbicia_matowy[4]={0.0,0.0,0.0,1.0};
```

```
    glMaterialfv(GL_FRONT,GL_SPECULAR,wsp_odbicia_szklo);
    glMateriali(GL_FRONT,GL_SHININESS,100);

    glPushMatrix(); //zapamietaj macierz model-widok (wloz na stos macierzy)
    glTranslatef(-2.0f,0.0f,1.0f);
    glColor3f(1.0f,1.0f,0.0f);
    Pisz("Arial 3D",czcionkaArial3D);

    glPopMatrix(); //zdejmij ze stosu macierzy = odtworz zapamietany stan
    glTranslatef(-2.0f,-1.0f,1.0f);
    glColor3f(0.0f,1.0f,0.0f);
    Pisz("Times 3D",czcionkaTimes3D);

    glMaterialfv(GL_FRONT,GL_SPECULAR,wsp_odbicia_matowy);
    glMateriali(GL_FRONT,GL_SHININESS,0);
}
```

Rysunek 31. Te napisy są prawdziwymi obiektami 3D

10. Szablon – ostatnie szlify

W tym momencie skończyliśmy w zasadzie przygotowywanie klasy `TGLForm`. Bez zmian będziemy wykorzystywali ją w kolejnych projektach w tym i w innych skryptach. Wyposażyliśmy ją w metody `RysujScene` i `Oswietlenie`, które po nadpisaniu w metodzie potomnej pozwalają programiście określić sposób wyświetlania aktorów na scenie oraz ich oświetlenie. Klasa `TGLForm` może być wyposażona w dodatkowe metody i własności ułatwiające korzystanie z niej do projektowania aplikacji OpenGL. W pliku http://www.fizyka.umk.pl/~jacek/dydaktyka/3d/szablon_bcb.zip dostępny jest szablon projektu oparty na tak rozszerzonej klasie, która została wzbogacona o własności: `ObslugaMyszy`, `KolorTla`, `Teksturowanie`, `SwobodneObroty`, `SwobodneObrotyWygaszanie`, `Mgla` itp. W skład szablonu, poza modułem *GLForm* wchodzą także moduły *GLWektory*, *GLNapisy* i Teksturowanie.

A. Biblioteka GLU. Kwadryki

Do biblioteki OpenGL — która zawiera zbiór funkcji pozwalających budować figury z werteksów i określać parametry poszczególnych źródeł światła oraz kontrolować efekty świetlne (dzięki powyższym projektom mamy chyba wyobrażenie, co umożliwia ta biblioteka) — dołączona jest zawsze biblioteka GLU. Zawiera ona zbiór funkcji wyższego poziomu, ułatwiających korzystanie z biblioteki OpenGL. Dobrym przykładem jest używana w projekcie 205. funkcja `gluLookAt`, która ustawia położenie, kierunek i polaryzację kamery, zwalniając nas z konieczności żmudnych obliczeń i przekształceń macierzy model-widok. Innym przykładem są funkcje pozwalające na definiowanie kwadryk, tj. figur przestrzennych zbudowanych z wielu wielokątów, jak na przykład sfera, cylinder, dysk i stożek. W matematyce kwadrykę definiuje się jako zbiór punktów w przestrzeni (powierzchnia), który można zadać równaniem kwadratowym[18]. Np. sfera będzie zadana równaniem $x^2 + y^2 + z^2 = r^2$. Biblioteka GLU zadba o przetłumaczenie równania na definicję zbioru werteksów.

A.1 Definiujemy kwadrykę i rysujemy sferę

Przygotowanie projektu: Kopiujemy schemat przygotowany w poprzedniej części skryptu i usuwamy wszystkie polecenia z metody `RysujScene` w klasie `TForm1`. Scenę oświetlamy „mleczną żarówką", kolorowymi źródłami światła i reflektorem (wywołania metod zdefiniowanych w podrozdziale 6, a konkretnie `MlecznaZarowka`, `ZoltaIZielonaMleczneZarowki` i `Reflektor` w metodzie `TForm1::Oswietlenie`). Zmniejszamy oświetlenie otoczenia do zera poleceniem `NatezenieSwiatlaOtoczenia=0.0f;` umieszczonym w konstruktorze `TForm1`.

Zbudowanie sfery wymaga utworzenia obiektu kwadryki. Robi to funkcja `gluNewQuadric`. Mając obiekt, wskazujemy mu równanie, które powinien „realizować". Użyjemy równania sfery zadawanego funkcją `gluSphere`. Jej jedynym argumentem geometrycznym jest promień `r` (drugi argument funkcji).

1. Tworzymy nową metodę klasy `TForm1` o nazwie `RysujGLU` (jej deklarację umieszczany w sekcji `private` klasy `TForm1`) zgodnie ze wzorem na listingu 104:

Listing 104. W osobnej metodzie umieszczamy polecenia związane z biblioteką GLU

```
void __fastcall TForm1::RysujGLU(float rozmiar)
{
   glColor4ub(255,255,255,255);
   GLUquadricObj* kwadryka=gluNewQuadric(); //tworzenie obiektu kwadryki
   gluSphere(kwadryka,rozmiar,30,30); //rysowanie
   gluDeleteQuadric(kwadryka); //usuwanie obiektu
}
```

2. Wywołanie metody `RysujGLU` dodajemy do metody `RysujScene` (listing 105).

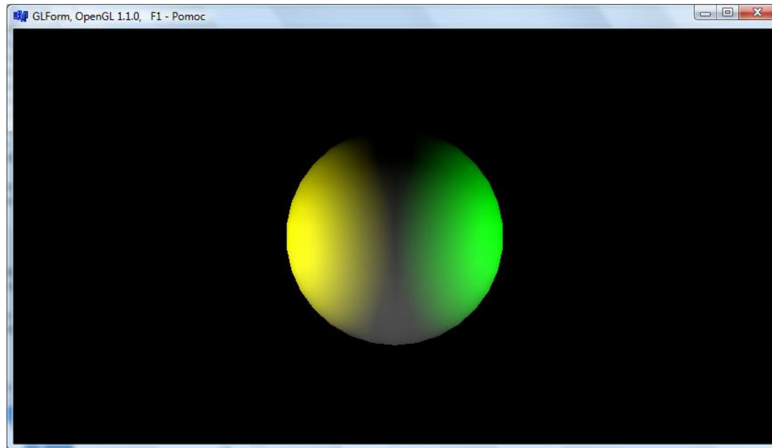
Listing 105. Funkcje biblioteki GLU to złożenia wielu funkcji niskopoziomowych biblioteki OpenGL

```
void __fastcall TForm1::RysujScene()
{
   float rozmiar=1.0f;
   RysujGLU(rozmiar);
}
```

W efekcie na scenie pojawi się sfera o promieniu równym `1.0`. Warto przetestować, w jaki sposób reaguje na różne typy oświetlenia (klawisze *1* – *4*; rysunek 32).

[18] Zob. *http://mathworld.wolfram.com/Quadric.html*.

Rysunek 32. *Sfera oświetlona czerwonym światłem z prawej strony*

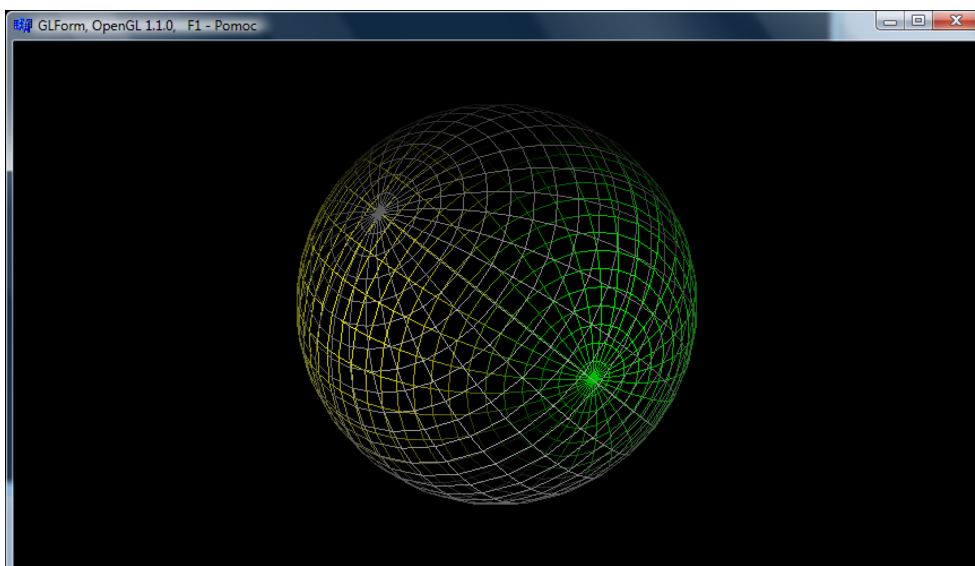


A.2 Styl rysowania kwadryki

Wróćmy jeszcze do metody `RysujGLU` z listingu 104. Uzupełnijmy ją o linię wyróżnioną na listingu 106. Znajdujące się w niej wywołanie funkcji `gluQuadricDrawStyle` ustala sposób rysowania kwadryki. Domyślne ustawienie identyfikowane jest przez stałą `GLU_FILL` — takiej wartości użyliśmy w poniższej linii, a więc jej dodanie nic nie zmieni. Kwadryka jest rysowana z pełną powierzchnią. Jednak jeżeli zmienimy tę stałą na `GLU_LINE` lub `GLU_SILHOUETTE`, to zamiast zamkniętej powierzchni sfery zobaczymy zbiór południków i równoleżników — krawędzi czworoboków, z których złożona jest sfera (rysunek 33). Z kolei użycie stałej `GLU_POINT` powoduje narysowanie jedynie wybranych punktów sfery. Proponuję jednak powrócić do domyślnej stałej `GLU_FILL`, żeby teksturowanie, które przygotujemy w następnym projekcie, wyglądało efektownie.

Listing 106. Gęstość siatki lub zbioru punktów zależy od trzeciego i czwartego argumentu funkcji gluSphere, a więc od ilości wielokątów wykorzystanych do narysowania sfery

```
void __fastcall TForm1::RysujGLU(float rozmiar)
{
   glColor4ub(255,255,255,255);
   GLUquadricObj* kwadryka=gluNewQuadric(); //tworzenie obiektu kwadryki
   gluQuadricDrawStyle(kwadryka,GLU_FILL); //ustalenie stylu
   gluSphere(kwadryka,rozmiar,30,30); //rysowanie sfery
   gluDeleteQuadric(kwadryka); //usuwanie obiektu
}
```



Rysunek 33. Dzięki stylowi, w którym sfera rysowana jest jako siatka, można ocenić ilość wielokątów użytych do narysowania kwadryki

A.3 Teksturowanie kwadryki

Na koniec zajmiemy się obłożeniem sfery teksturą. Wymaga to kilku przygotowań. Po pierwsze musimy dysponować mapą tekstury będącą tablicą liczb naturalnych typu `long`. Taką mapę trzeba oczywiście przygotować — najprościej będzie zbudować ją na podstawie obrazu wczytanego z pliku graficznego. Najczęściej dostępne są pliki JPEG. Dlatego taki typ pliku wykorzystamy w naszym przykładzie. Obraz tego typu należy skonwertować do formatu mapy bitowej, a z niej „wyciągnąć" tablicę odgrywającą rolę mapy tekstury. Tymi wszystkimi czynnościami zajmie się przygotowana wcześniej metoda `PrzygotujTeksture` z modułu Teksturowanie (zob. podrozdział 8). Następnie do metody `RysujGLU` dodamy polecenia dopasowujące i nakładające teksturę na sferę. Dysponując metodą wczytującą teksturę z pliku mamy bardzo ułatwione zadanie, a efekt jest atrakcyjny (rysunek 34).

1. Do klasy `TForm1` dodajemy prywatne pole `czyTeksturowacSfere`.

2. W konstruktorze klasy `TForm1` umieszczamy polecenia wczytujące teksturę (listing 107). Wykorzystujemy do tego funkcję `WczytajTeksture` z podrozdziału 8. Kod umieszczany w konstruktorze jest niemal identyczny z tym z listingu 87.

   Listing 107.

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TGLForm(Owner)
{
   ObslugaMyszy=true;
   KolorTla=clBlack;
   debug_mode=false;
   Mgla=false;
   NieruchomyPokoj=false;
   NieruchomyPokoj_Kolor=clBlue;
   NatezenieSwiatlaOtoczenia=0.5f;
   SwobodneObroty=true;
   SwobodneObrotyWygaszanie=0.0f;

   Obracaj(10.0f,0.0f,0.0f);

   Caption=(AnsiString)"GLForm, OpenGL "+(char*)glGetString(GL_VERSION);

   //teksturowanie sfery
   czyTeksturowacSfere=true;
   glEnable(GL_TEXTURE_2D);

   int TeksturaSzer,TeksturaWys;
   unsigned long* Tekstura=WczytajTeksture("tekstura.jpg",TeksturaSzer,TeksturaWys);
   if (Tekstura!=NULL)
   {
      glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
      glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
      gluBuild2DMipmaps(GL_TEXTURE_2D, 3,TeksturaSzer,TeksturaWys, GL_RGBA,
GL_UNSIGNED_BYTE, Tekstura);
      delete[] Tekstura; //oryginalne dane sa usuwane
   }
}
```

3. Modyfikujemy metodę `RysujGLU` zgodnie z wyróżnieniami na listingu 108.

Listing 108. Teksturowanie kwadryki

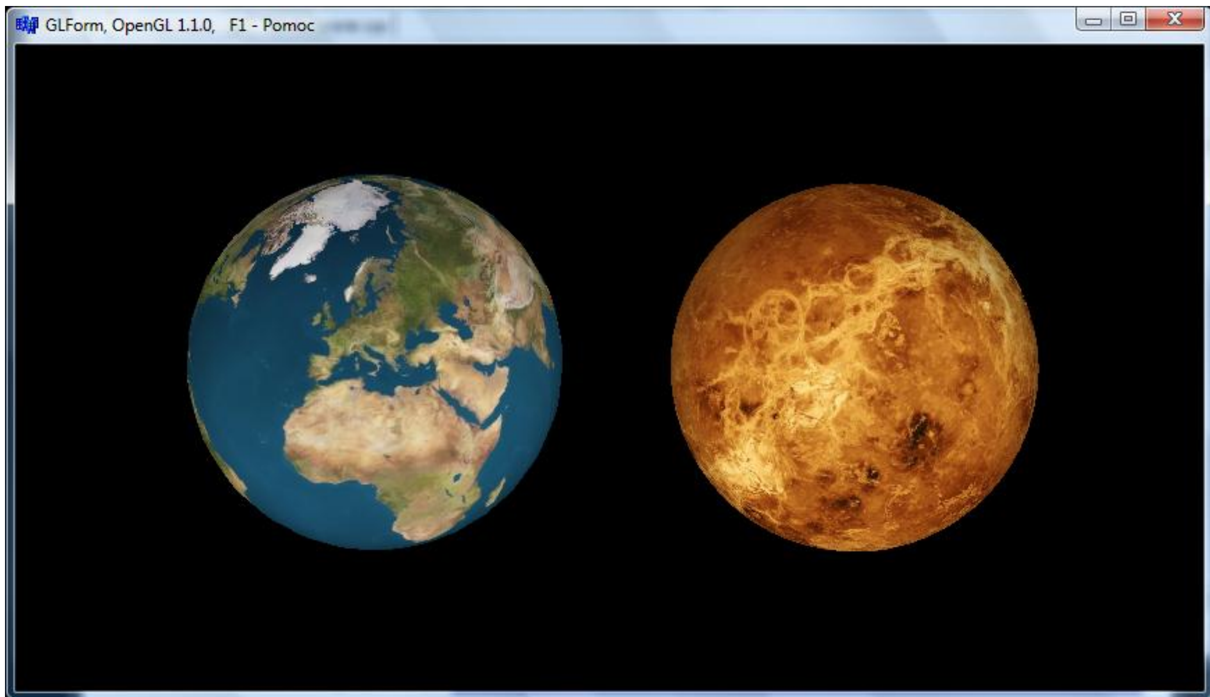
```
void __fastcall TForm1::RysujGLU(float rozmiar)
{
   glColor4ub(255,255,255,255);
   glRotatef(90.0f,1.0f,0.0f,0.0f);
   GLUquadricObj* kwadryka=gluNewQuadric(); //tworzenie obiektu kwadryki
   gluQuadricDrawStyle(kwadryka,GLU_FILL); //ustalenie stylu
   if (czyTeksturowacSfere) gluQuadricTexture(kwadryka,GL_TRUE);
   gluSphere(kwadryka,rozmiar,30,30); //rysowanie
   gluQuadricTexture(kwadryka,GL_FALSE);
   gluDeleteQuadric(kwadryka); //usuwanie obiektu
}
```

4. W metodzie `FormKeyDown` montujemy włącznik teksturowania:

Listing 109.

```
case 'i':
case 'I':
   czyTeksturowacSfere=!czyTeksturowacSfere;
   break;
```

5. Ze strony *http://www.oera.net/How2/TextureMaps2.htm* pobieramy obraz JPEG tekstury kuli ziemskiej i zapisujemy go w katalogu projektu, w podkatalogu *Debug_Build*, pod nazwą *tekstura.jpg*. Jeżeli mamy możliwość warto zmniejszyć obraz do rozmiaru około tysiąc na tysiąc punktów.
6. Kompilujemy projekt i uruchamiamy aplikację. Efekt, jaki zobaczymy, powinien wyglądać jak na rysunku 34.



Rysunek 34. Sfery pokryte teksturami planet (Ziemia i Wenus)

A.4 Przegląd kwadryk

Sfera to oczywiście nie jedyna kwadryka, jaką można utworzyć za pomocą biblioteki GLU. Dodajmy do metody `RysujGLU` polecenia, które narysują „kołnierz" oraz pierścień przypominający pierścienie Saturna (kod wyróżniony w listingu 110). Rysowanie dodatkowych kwadryk nie wymaga utworzenia osobnego obiektu kwadryki, jeżeli stosujemy do niego te same ustawienia. Jeżeli nie chcemy, aby tekstura użyta do sfery była nakładana też na nowe kwadryki, należy przed ich narysowaniem wywołać funkcję wyłączającą teksturowanie aktualnej kwadryki `gluQuadricTexture(kwadryka,GL_FALSE);`.

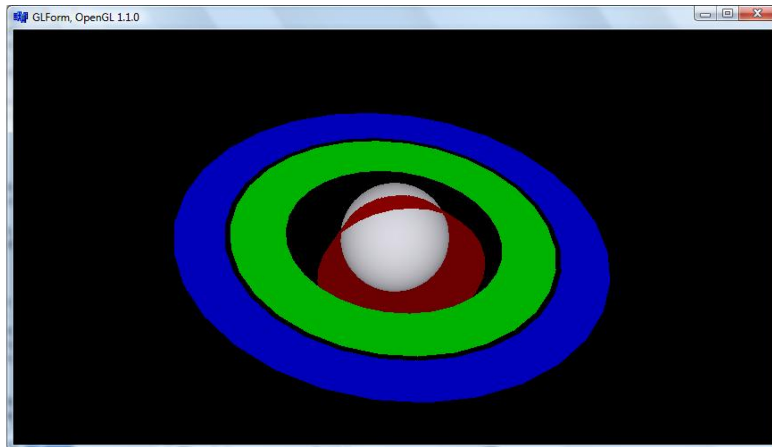
Listing 110. Dodatkowe kwadryki

```
void __fastcall TForm1::RysujGLU(float rozmiar)
{
   glColor4ub(255,255,255,255);
   glRotatef(90.0f,1.0f,0.0f,0.0f);
   GLUquadricObj* kwadryka=gluNewQuadric(); //tworzenie obiektu kwadryki
   gluQuadricDrawStyle(kwadryka,GLU_FILL); //ustalenie stylu
   if (czyTeksturowacSfere) gluQuadricTexture(kwadryka,GL_TRUE);
   gluSphere(kwadryka,rozmiar,30,30); //rysowanie
   gluQuadricTexture(kwadryka,GL_FALSE);

   glColor3ub(255,0,0);
   gluCylinder(kwadryka,rozmiar,1.5*rozmiar,rozmiar,30,30); //rysowanie stożka
   glRotatef(-90,1,0,0);
   glColor3ub(0,255,0);
   gluDisk(kwadryka,2*rozmiar,3*rozmiar,30,30); //rysowanie dysku
```

```
    glColor3ub(0,0,255);
    gluDisk(kwadryka,3.1*rozmiar,4*rozmiar,30,30); //rysowanie dysku
    glRotatef(90,1,0,0);
    gluDeleteQuadric(kwadryka); //usuwanie obiektu
}
```

Narysowany fragment stożka i pierścienie nie są figurami zamkniętymi (por. rysunek 35). Zatem jedna z ich stron reaguje na światło inaczej niż druga. Pierścień jest obrócony o 90 stopni wokół osi OX (po narysowaniu pierścienia wykonywany jest obrót przywracający pierwotną orientację).

Rysunek 35. *Dodatkowe kwadryki: dwa pierścienie i „kołnierz”*



B. Krzywe i płaszczyzny Béziera

Krzywa Béziera to parametryczna krzywa wielomianowa wyznaczona przez tzw. punkty kontrolne. Stopień wielomianu zależy od ilości tych punktów. Krzywe Béziera są szczególnie użyteczne w programach graficznych do rysowania krzywych; użytkownik wyznacza myszą punkty kontrolne zmieniając kształt krzywej (zob. chociażby dołączony do Windows edytor Paint).

Punkty krzywej Béziera dane są wzorem

$$\vec{r}(u) = [x(u), y(u), z(u)] = \sum_{i=0}^{n} \binom{n}{i} \vec{P}_i (1-u)^{n-i} u^i \,,$$

gdzie P_i to punkty kontrolne w przestrzeni 3D, u – zmienna parametryczna przebiegająca wartości z przedziału [0,1]. Dla pięciu punktów kontrolnych $n = 5$ mamy:

$$\vec{r}(u) = \vec{P}_0(1-u)^5 + 5\vec{P}_1 u(1-u)^4 + 10\vec{P}_2 u^2(1-u)^3 + 10\vec{P}_3 u^3(1-u)^2 + 5\vec{P}_4 u^4(1-u) + \vec{P}_3 u^5$$

Widać, że dla $u = 0$ $\vec{r}(0) = \vec{P}_0$, a dla $u = 1$ otrzymujemy $\vec{r}(1) = \vec{P}_n$. Tylko te dwa punkty kontrolne (pierwszy i ostatni) na pewno należą do krzywej. W OpenGL stosuje się najczęściej parametry u o wartościach od 0 do 100.

Więcej informacji:
http://mathworld.wolfram.com/BezierCurve.html,
http://pl.wikipedia.org/wiki/Krzywa_B%C3%A9ziera

B.1 Dwuwymiarowe krzywe Béziera

Przygotowanie projektu: Kopiujemy schemat przygotowany w poprzedniej części skryptu i usuwamy wszystkie polecenia z metody `RysujScene` w klasie `TForm1`. Scenę oświetlamy „mleczną żarówką”, kolorowymi źródłami światła i reflektorem. Światło otoczenia ustawiamy na 100%.

1. W nowym projekcie tworzymy nowy moduł *Bezier.h*/*Bezier.cpp*.

2. Definiujemy w nim funkcję `KrzywaBeziera` (listing 111). Deklarujemy ją w pliku nagłówkowym modułu.

Listing 111. Rysowanie krzywych Beziera

```
//---------------------------------------------------------------------------

#pragma hdrstop

#include "Bezier.h"

#include <System.hpp>
#include <gl.h>

//---------------------------------------------------------------------------

#pragma package(smart_init)

void    KrzywaBeziera(int    ilePunktowKontrolnych,float    punktyKontrolne[][3],bool
rysujPunktyKontrolne=false)
{
   //rysowanie punktow kontrolnych
   if (rysujPunktyKontrolne)
   {
      //glPointSize(2.0f);
      //glBegin(GL_POINTS);
      //linia przerywana
      glEnable(GL_LINE_STIPPLE);
      glLineStipple(5,0x5555);
      glBegin(GL_LINE_STRIP);
      for(int i=0;i<ilePunktowKontrolnych;i++)
         glVertex3fv(punktyKontrolne[i]);
      glEnd();
      glDisable(GL_LINE_STIPPLE);
   }

   float zakres_min=0.0f;
   float zakres_max=100.0f;
   glMap1f(
      GL_MAP1_VERTEX_3,        //typ generowanych danych = krzywa na plaszczyznie
      zakres_min,zakres_max,   //zakres parametru krzywej
      3,                       //rozmiar punktu (w 3D bez koloru = 3)
      ilePunktowKontrolnych,
      (GLfloat*)punktyKontrolne);

   glEnable(GL_MAP1_VERTEX_3); //wlaczenie ewaluatora

   glBegin(GL_LINE_STRIP);
   for(float u=zakres_min;u<=zakres_max;u++)
   {
      glEvalCoord1f(u);
   }
   glEnd();

   glDisable(GL_MAP1_VERTEX_3);   //wylaczenie ewaluatora
}
```

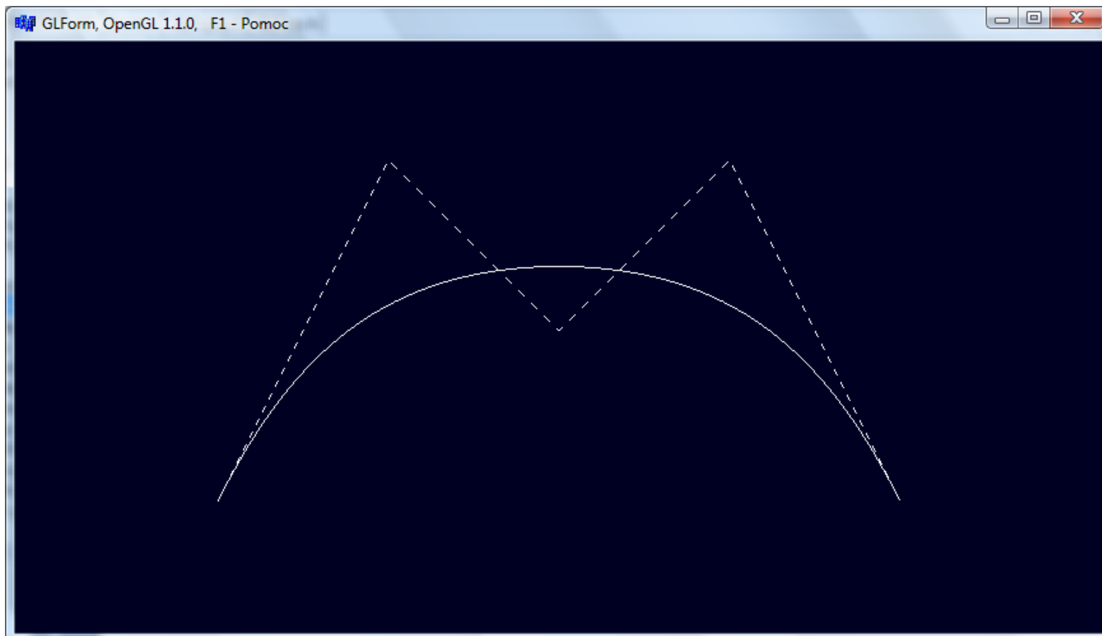
Funkcja `glEvalCoord1f` oblicza punkt z krzywej Beziera dla danej wartości parametru. Do nas należy już tykji zbudowanie linii (`GL_LINE_STRIP`).

3. Funkcję `KrzywaBeziera` wywołujemy z nadpisanej w `TForm1` metody `RysujScene` (listing 112, rysunek 36).

Listing 112.

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0f;
   const float y0=1.0f;
   const float z0=1.0f;
   const int ilePunktowKontrolnych=5;

   //2D
   GLfloat punktyKontrolne[ilePunktowKontrolnych][3]=
      {
         {-x0,0,0},
         {-x0/2,y0,0},
         {0,y0/2,0},
```

```
        {x0/2,y0,0},
        {x0,0,0}
    };
  glColor3f(1.0f,1.0f,1.0f);
  KrzywaBeziera(ilePunktowKontrolnych,punktyKontrolne,true);
}
```



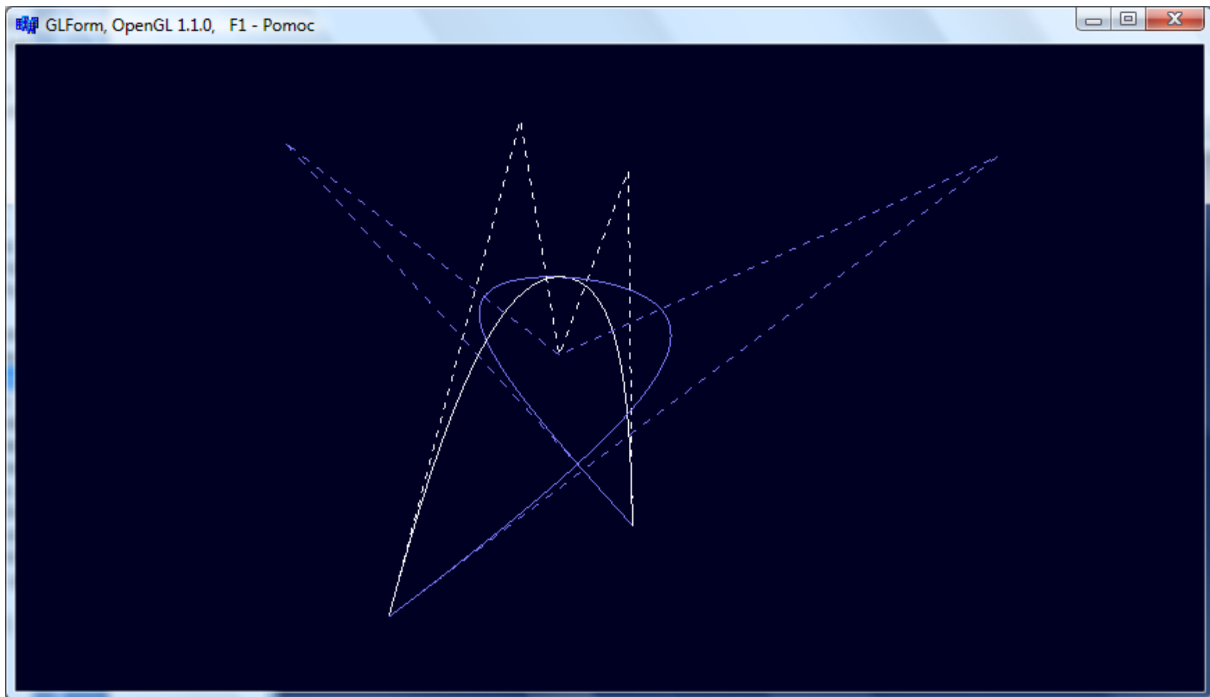
Rysunek 36. Dwuwymiarowa krzywa Béziera

B.2 Trójwymiarowe krzywe Béziera

Wszystkie punkty kontrolne zdefiniowane w listingu 112 miały składową *z* równą 0. Jeżeli pozwolimy na jej zmianę, tzn. nasze punkty kontrolne staną się niewspółpłaszczyznowe, uzyskamy trójwymiarowe krzywe Béziera. Listing 113 pokazuje niezbędne modyfikacje w metodzie `RysujScene`. Rysunek 37 pokazuje razem krzywą 2D i 3D.

Listing 113. Rysowanie trójwymiarowych krzywych Beziera

```
void __fastcall TForm1::RysujScene()
{
  const float x0=1.0f;
  const float y0=1.0f;
  const float z0=1.0f;
  const int ilePunktowKontrolnych=5;

  //2D
  GLfloat punktyKontrolne[ilePunktowKontrolnych][3]=
    {
      {-x0,0,0},
      {-x0/2,y0,z0},
      {0,y0/2,0},
      {x0/2,y0,-z0},
      {x0,0,0}
    };
  glColor3f(1.0f,1.0f,1.0f);
  KrzywaBeziera(ilePunktowKontrolnych,punktyKontrolne,true);
}
```



Rysunek 37. Dwu- i trójwymiarowe krzywe Beziera

B.3 Powierzchnie Beziera

Powierzchnia Beziera to po prostu zbiór krzywych Beziera. Narysujemy płaszczyznę Beziera dla 30 punktów kontrolnych ułożonych 5x6 (tj. tensor 5x6x3 liczb).

Etap 1: Przygotowanie funkcji

Listing 114.

```
void PowierzchniaBeziera(const int ilePunktowKontrolnych[2],float* punktyKontrolne,bool
rysujPunktyKontrolne=false)
{
   //rysowanie punktow kontrolnych
   if (rysujPunktyKontrolne)
   {
      glPointSize(2.0f);
      glBegin(GL_POINTS);
      for(int iu=0;iu<ilePunktowKontrolnych[0];iu++)
         for(int iv=0;iv<ilePunktowKontrolnych[1];iv++)
         {
            glVertex3fv(punktyKontrolne+(iu*ilePunktowKontrolnych[1]+iv)*3);
         }
      glEnd();
   }

   float zakres_min=0.0f;
   float zakres_max=100.0f;
   glMap2f(
      GL_MAP2_VERTEX_3,             //typ generowanych danych = krzywa na plaszczyznie
      //parametr u
      zakres_min,zakres_max,        //zakres parametru krzywej
      3,                            //rozmiar punktu (w 3D = 3)
      ilePunktowKontrolnych[1],
      //parametr v
      zakres_min,zakres_max,        //zakres parametru krzywej
      ilePunktowKontrolnych[1]*3,   //odleglosc miedzy punktami w danych dla v
      ilePunktowKontrolnych[0],
      punktyKontrolne);             //punktyKontrolne);

   glEnable(GL_MAP2_VERTEX_3);      //wlaczenie ewaluatora


   //linie w jednym kierunku
   for(float u=zakres_min;u<=zakres_max;u+=10.f)
   {
      glBegin(GL_LINE_STRIP);
```

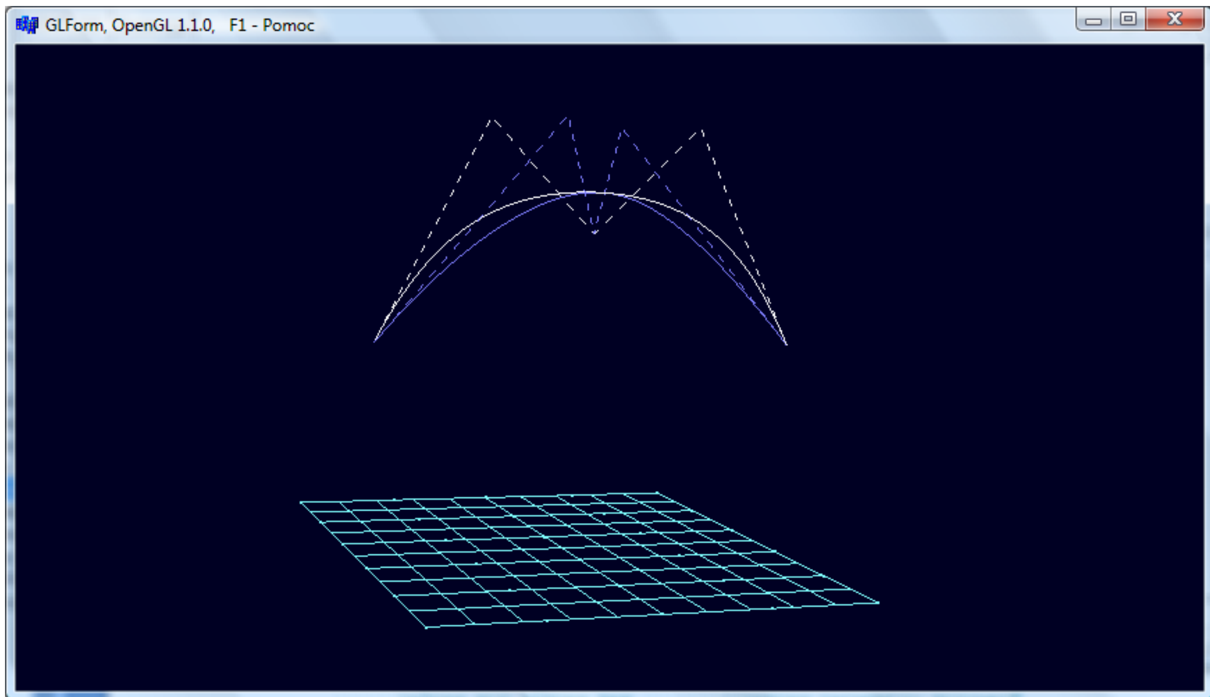
```
         for(float v=zakres_min;v<=zakres_max;v+=10.f)
         {
            glEvalCoord2f(u,v);
         }
         glEnd();
      }
      //linie w drugim kierunku
      for(float u=zakres_min;u<=zakres_max;u+=10.f)
      {
         glBegin(GL_LINE_STRIP);
         for(float v=zakres_min;v<=zakres_max;v+=10.f)
            glEvalCoord2f(v,u);
         glEnd();
      }
   }
```

Etap 2: Testy - powierzchnia płaska

Aby przetestować powyższą metodę narysujemy płaską powierzchnię `y=-1`.

Listing 115.

```
   void __fastcall TForm1::RysujScene()
   {
      const float x0=1.0f;
      const float y0=1.0f;
      const float z0=1.0f;

      ...

      //Powierzchnia
      const int nu=5;
      const int nv=6;
      const int ilePunktowKontrolnychPowierzchnia[2]={nu,nv};
      GLfloat punktyKontrolnePowierzchnia[nu][nv][3];
      for(int iu=0;iu<nu;iu++)
         for(int iv=0;iv<nv;iv++)
         {
            punktyKontrolnePowierzchnia[iu][iv][0]=x0*(iu-2)/2;
            punktyKontrolnePowierzchnia[iu][iv][1]=-y0;
            punktyKontrolnePowierzchnia[iu][iv][2]=z0*(iv-2)/2;
         }

      glColor3f(0.5f,1.0f,1.0f);
      PowierzchniaBeziera(ilePunktowKontrolnychPowierzchnia,
                          (float*)punktyKontrolnePowierzchnia,
                          true);
   }
```



Rysunek 38.

Etap 3: Automatyczne generowanie siatki i ewaluacja krzywych Béziera

Koniec funkcji `PowierzchniaBeziera` zajmują polecenia rysujące siatkę punktów (zestaw linii łamanych w kierunku rosnącego parametru *u*, a potem – *v*). Zamiast tego można użyć funkcji OpenGL, które narysują siatkę za nas. Lising ** pokazuje odpowiednie modyfikacje w funkcji `PowierzchniaBeziera`.

Listing 116. Zmiany w funkcji PowierzchniaBeziera

```
//linie w jednym kierunku
for(float u=zakres_min;u<=zakres_max;u+=10.f)
{
  glBegin(GL_LINE_STRIP);
  for(float v=zakres_min;v<=zakres_max;v+=10.f)
  {
          glEvalCoord2f(u,v);
  }
  glEnd();
}
//linie w drugim kierunku
for(float u=zakres_min;u<=zakres_max;u+=10.f)
{
  glBegin(GL_LINE_STRIP);
  for(float v=zakres_min;v<=zakres_max;v+=10.f)
          glEvalCoord2f(v,u);
  glEnd();
}

//automatyczne generowanie siatki i ewaluacja
const int gestoscSiatki=10;
glMapGrid2f(gestoscSiatki,zakres_min,zakres_max,gestoscSiatki,zakres_min,zakres_max);
glEvalMesh2(GL_LINE,0,gestoscSiatki,0,gestoscSiatki);
```

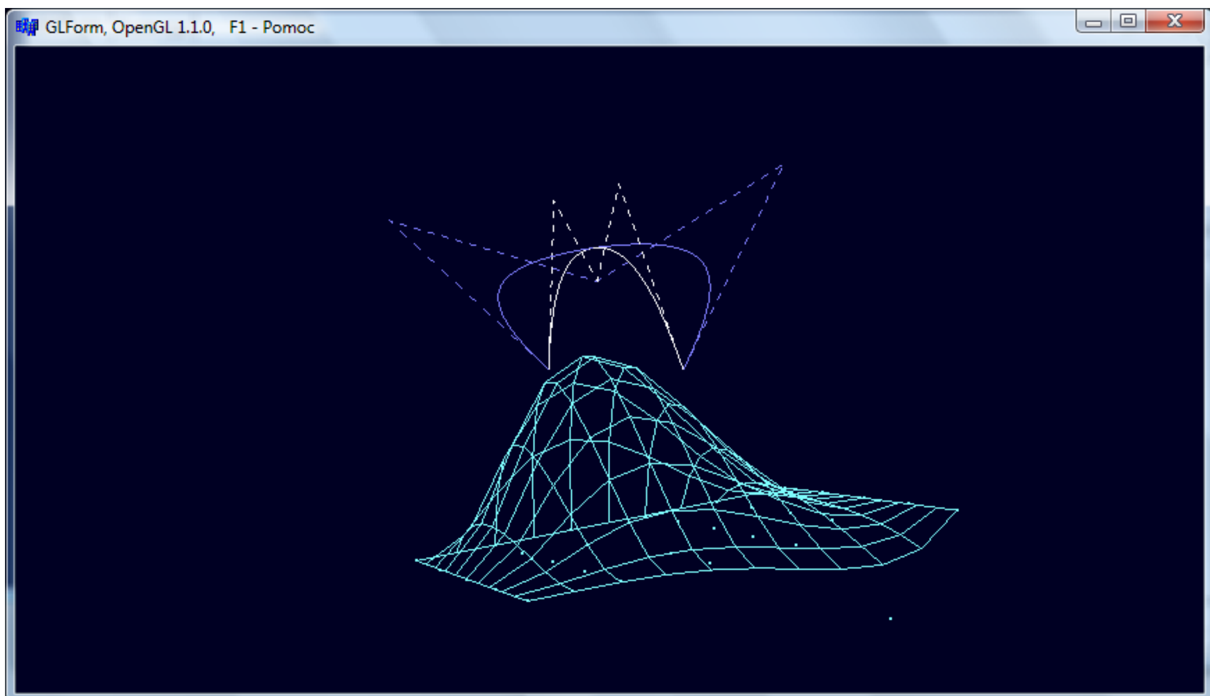
Etap 4: Zakrzywiamy przestrzeń

Teraz wystarczy jedynie pobawić się punktami z tablicy `punktyKontrolnePowierzchnia` np.:

Listing 117.

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0f;
   const float y0=1.0f;
   const float z0=1.0f;

   ...

   //Powierzchnia
```

```
   const int nu=5;
   const int nv=6;
   const int ilePunktowKontrolnychPowierzchnia[2]={nu,nv};
   GLfloat punktyKontrolnePowierzchnia[nu][nv][3];
   for(int iu=0;iu<nu;iu++)
      for(int iv=0;iv<nv;iv++)
      {
         punktyKontrolnePowierzchnia[iu][iv][0]=x0*(iu-2)/2;
         punktyKontrolnePowierzchnia[iu][iv][1]=-y0;
         punktyKontrolnePowierzchnia[iu][iv][2]=z0*(iv-2)/2;
      }

   punktyKontrolnePowierzchnia[2][2][1]=3*y0;
   punktyKontrolnePowierzchnia[3][2][1]=4*y0;
   punktyKontrolnePowierzchnia[0][4][1]=-1.5*y0;

   glColor3f(0.5f,1.0f,1.0f);
   PowierzchniaBeziera(ilePunktowKontrolnychPowierzchnia,
                       (float*)punktyKontrolnePowierzchnia,
                       true);
}
```



Rysunek 39.

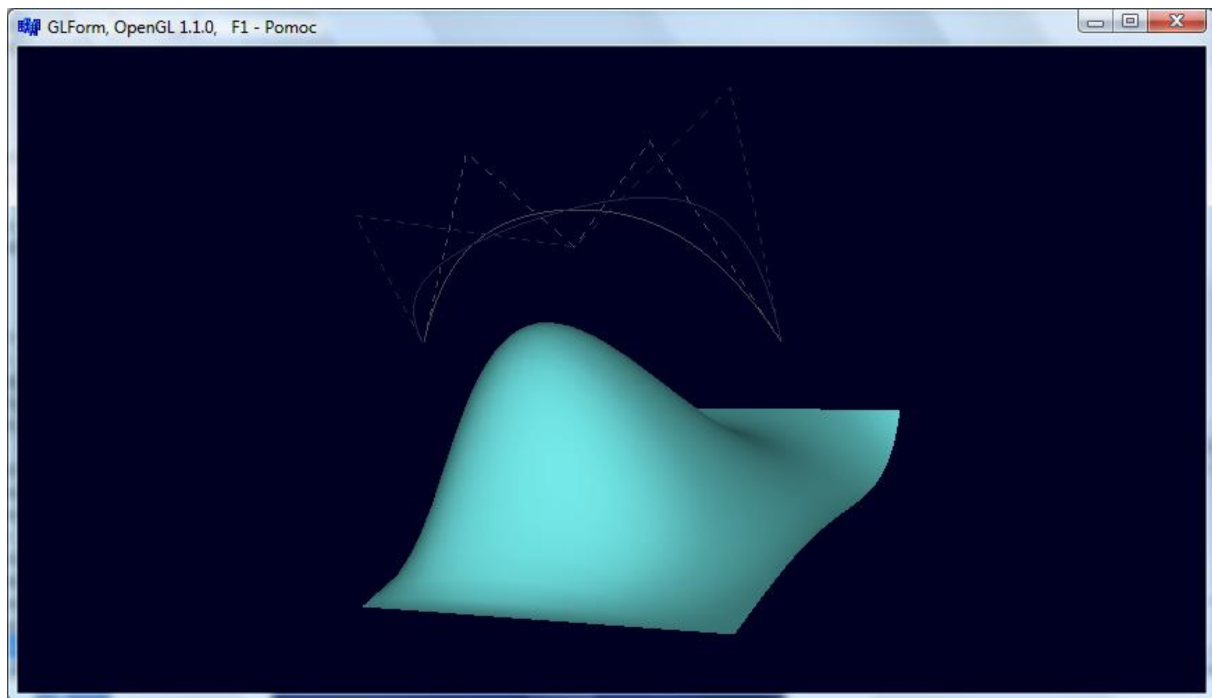
B.4 Oświetlanie powierzchni Béziera

Zaletą korzystania z automatycznych ewaluatorów jest generowanie wektorów normalnych. Wykorzystajmy tę możliwość przy oświetlaniu powierzchni Béziera.

1. Modyfikujemy ostatnie linie metody `PowierzchniaBeziera` zmieniając sposób rysowania powierzchni z `GL_LINE` na `GL_FILL`, zagęszczamy siatkę i włączamy generowanie normalnych:
   Listing 118.
   ```
   const int gestoscSiatki=30;
   glMapGrid2f(gestoscSiatki,zakres_min,zakres_max,gestoscSiatki,zakres_min,zakres_max);
   glEvalMesh2(GL_FILL,0,gestoscSiatki,0,gestoscSiatki);
   ```
2. Do poleceń inicjujących (np. w konstruktorze klasy `TForm1`) dodajemy polecenie włączające generowanie normalnych na automatycznie generowanych siatkach: `glEnable(GL_AUTO_NORMAL);`.
3. Redukujemy oświetlenie otoczenia umieszczając w konstruktorze klasy `TForm1` polecenie `NatezenieSwiatlaOtoczenia=0.0f;`.

Listing 119.
```
__fastcall TForm1::TForm1(TComponent* Owner)
  : TFormGL(Owner),
```

```
 pozycjaZarowkiX(2.0f),pozycjaZarowkiY(2.0f),pozycjaZarowkiZ(-2.0f)
{
 NieruchomyPokoj=true;
 NatezenieSwiatlaOtoczenia=0.3f;
 debug_mode=true;
}
```



Rysunek 40.

Zobacz także: krzywe B-sklejane (NURBS) w bibliotece GLU.