

# OpenGL.

Silicon Graphics (IRIS GL -stacje graficzne)

## **Biblioteka**

- przestrzeń 3D
- rzutowanie
- prymitywy graficzne
- operacje na barwach

## **HISTORIA**

**1992** - powstaje wersja 1.0 specyfikacji OpenGL przenośnej między platformami.

OpenGL Architecture Review Board (SG,HP,IBM)

**1995** - wersja 1.1 tej biblioteki z wieloma poprawkami przyspieszającymi wyświetlanie grafiki.

**OpenGL** nie należy do języków opisu sceny. Scena tworzona jest w OpenGL z wielokątów poprzez wywoływanie odpowiednich procedur

## **Języki programowania**

Pascal, C/C++, Visual Basic, Python, Java

# OpenGL (WINDOWS)

**OpenGL** nie ma funkcji obsługujących operacje:

- wejścia/wyjścia
- interakcje z użytkownikiem
- zarządzanie oknami.

## **WINDOWS**

Wprowadzono w wersjach

- Win95 OSR2
- WinNT 3.51/4.0

Obsługa okien w OpenGL - funkcje wigggle.

Wywołania funkcji w OpenGL są zgodne z konwencją wywoływania funkcji w języku C.

## Biblioteki w systemie WIN32

`glaux.lib` - Auxiliary (pomocnicza) przenośna  
`glaux.h`

przedrostek funkcji: `aux`

<code>opengl32.dll</code>	<code>gl.h</code>	<code>gl</code>
<code>glu32.dll</code>	<code>glu.h</code>	<code>glu</code>

# OpenGL (AUX)

Biblioteka aux raczej jest pewną podstawą dla wywoływania f. OpenGL. Zaletą jej jest to, że na każdej platformie wygląda ona tak samo.

OpenGL32.dll właściwe wywołania OpenGL.  
Glu32.dll biblioteka pomocnicza rysowanie skomplikowanych obiektów cylindry, walce, NURBS.

# OpenGL (Typy danych)

OpenGL w celu przenośności między platformami wprowadza swoje typy danych.

`Glbyte, Glshort, Glint, Glsize, Glfloat, Glclampf, Gldouble, Glclampd, Glubyte, Glboolean, Glushort, Gluint, Glenum, Glbitfield.`

**glClamp** - color amplitude

# OpenGL (Konwencje nazw)

<Przedrostek biblioteki><rdzeń  
polecenia><Opcjonalnie liczba  
argumentów><Opcjonalnie typ argumentów>

glColor3f(...)

gl - gl.h

Color - nazwa

3f - 3 argumenty float.

# OpenGL (Biblioteka aux)

Przewidziana jako wspomaganie dla OpenGL w postaci implementacji przenośnej między systemami.

Podstawowe operacje inicjalizujące wej/wyj.

# OpenGL (kod aux)

```
#include <windows.h>
#include <gl\gl.h>
#include <gl\glaux.h>
void CALLBACK RenderScene(void)
{
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0f, 0.0f, 0.0f);
    glRectf(100.0f, 150.0f, 150.0f, 100.0f);
    glFlush();
}
void main(void)
{
    auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
    auxInitPosition(100,100,450,450);
    auxInitWindow("Test2");
    auxMainLoop(RenderScene);
}
```

# OpenGL (kod aux)

```
auxInitDisplayMode (AUX_SINGLE | AUX_RGBA) ;
```

Tryb wyświetlania:

AUX\_SINGLE - pojedynczy bufor

AUX\_RGBA - tryb kolorów.

```
auxInitPosition (100, 100, 450, 450) ;
```

Pozycja i wymiary okna.

```
auxInitWindow ("Test2") ;
```

Nazwa okna

```
auxMainLoop (RenderScene) ;
```

Główna pętla renderowania obrazu.

## OpenGL (kod aux)

```
glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
```

Kolor używany do czyszczenia ekranu.

Krok koloru - 0.00006

```
glClear(GL_COLOR_BUFFER_BIT);
```

Wykonanie czyszczenia ekranu.

```
glColor3f(1.0f, 0.0f, 0.0f);
```

kolor rysowania.

```
glRectf(100.0f, 150.0f, 150.0f, 100.0f);
```

rysujemy prostokąt.

```
glFlush () ;
```

Przetwarzaj dotychczasowe komendy.

## **Nagłówki**

```
#include <windows.h>
```

```
#include <gl\gl.h>
```

```
#include <gl\glaux.h>
```

# Skalowanie okna wyświetlania.

Nasz rysunek jest w rzeczywistości w przestrzeni 3D  $z=0$ .

Przy zmianie wymiarów okna wyświetlania nasz obrazek też powinien być odpowiednio przeskalowany.

Do tego służy funkcja **auxReshapeFunc** (`far* func`), gdzie argumentem jest funkcja zwrotna o następującym prototypie.

**void CALLBACK** ChangeSize(`Glsizei w`, `Glsizei h`)  
funkcja ta otrzymuje wysokość i szerokość z okna macierzystego przy każdej próbie przeskalowania go.

# Skalowanie okna wyświetlania.

Możemy użyć tego do odwzorowania naszego układu współrzędnych na układ współrzędnych ekranu za pomocą funkcji;

## DEFINIOWANIE WIDOKU

**glViewport**(Glint x, Glint y, Glint w, Glint h);  
x,y - prawy dolny róg widoku  
w,h - szerokość, wysokość

# Skalowanie okna wyświetlania.

## DEFINIOWANIE BRYŁY OBCINANIA

Po zmianie rozmiaru okna musimy przeddefiniować bryłę obcinania aby stosunki współrzędnych zostały takie same.

*Stosunek współrzędnych (aspect ratio)* to stosunek ilości pixeli odpowiadający jednostce osi pionowej do ilości pikseli na osi poziomej. 1.0 równe ilości pikseli na osiach.

Rzutowanie równoległe

**glOrtho**(GLdouble lewa, GLdouble prawa, GLdouble dolna, GLdouble górna, GLdouble bliższa, GLdouble dalsza);

# Skalowanie okna wyświetlania.

## DEFINIOWANIE BRYŁY OBCINANIA

**glLoadIdentity()**; - macierz jdnostkowa.

**glOrtho()** modyfikuje istniejące obcinanie.

```
Void CALLBACK ChangeSize(Glsizei w, Glsizei h){
    if(h==0)
        h=1;
    glViewport(0,0,w,h);
    glLoadIdentity();
    if(w<=h)
        glOrtho(0.0f, 250.0f, 0.0f, 250.0f*h/w, 1.0,-1.0);
    else
        glOrtho(0.0f, 250.0f*w/h, 0.0f, 250.0f, 1.0,-1.0);
}
auxReshapeFunc(ChangeSize);
```

# Animacje.

Animacje w bibliotece pomocniczej osiągamy przez funkcje;

**auxIdleFunction()** gdzie argumentem jest funkcja o prototypie  
`void CALLBACK IdleFunction(void);`

Funkcja jest wywoływana w momencie bezczynności programu. Nie jest wywoływana w momencie skalowania okna.

Z animacją wiąże się problem wyświetlania. Najczęściej wyświetlanie nie nadąża za procesem rysowania co przejawiać się może w miganiu rysowanego obiektu.

# Animacje.

W celu uniknięcia takiego zjawiska stosuje się podwójne buforowanie.

Obraz naszej sceny nie jest rysowany na bufor ekranu tylko do pomocniczego bufora, a z niego dopiero a ekran.

```
auxDisplayMode (AUX_DOUBLE, AUX_RGBA)
```

```
auxSwapBuffers ();
```

# Klawiatura.

Biblioteka AUX pozwala nam modyfikować naszą scenę przez klawisze.

```
auxKeyFunc (Glint key, void(*func()));
```

```
void CALLBACK KeyFunc (void);
```

```
AUX_ESCAPE
```

```
AUX_SPACE
```

```
AUX_LEFT
```

```
AUX_RIGHT
```

```
AUX_UP
```

```
AUX_DOWN
```

# Mysz.

Biblioteka AUX pozwala nam na interakcje z naszą sceną przez mysz.

```
auxMouseFunc(Glint button, Glint mode void(*  
MouseFunc()));
```

PROTOTYP

```
void CALLBACK MouseFunc(AUX_EVENTREC *event);
```

```
typedef struct _AUX_EVENTREC{  
    Glint event;  
    Glint data[4];
```

```
}AUX_EVENTREC
```

```
event - AUX_MOUSEUP, AUX_MOUSEUP
```

```
data[AUX_MOUSEX]=pozioma współrzędna myszy
```

```
data[AUX_MOUSEY]=pionowa współrzędna myszy
```

```
data[MOUSE_STATUS]=przycisk myszy (button param)
```

# Prymitywy graficzne.

Rysowanie w 3 wymiarach.

Układ współrzędnych określa nam bryła obcinania (glOrtho).

(-100,100)

1. Trójwymiarowy punkt: wierzchołek

```
glVertex2f(10.0f, 10.0f);
```

```
glVertex3f(10.0f, 10.0f, 0.0f);
```

```
glVertex4f(10.0f, 10.0f, 0.0f, 1.0f);
```

w - współczynnik skalowania.

# Prymitywy graficzne.

Teraz mamy wierzchołek, wystarczy nam teraz określić czy ten wierzchołek będzie punktem, wierzchołkiem odcinka czy też jakiegoś wieloboku.

Zadania wierzchołków grupujemy w pętli.

```
glBegin (GLenum mode)
```

```
    ...
```

```
glEnd ()
```

Zestaw punktów w 3D.

```
glBegin (GL_POINTS) ;  
    glVertex3f (10.0f, 10.0f, 0.0f) ;  
    glVertex3f (10.0f, 1.0f, 0.0f) ;  
glEnd () ;
```

# Prymitywy graficzne.

Pamiętać trzeba o tym że punkt jest rysowany aktualnie wybranym kolorem.

W pętli glBegin/glEnd można zawrzeć dowolną liczbę prymitywów.

Ponawianie ustawiania prymitywu powoduje spowolnienie całego procesu tworzenia grafiki.

```
glBegin (GL_POINTS) ;  
    for (i=0 ; i<10 ; i++)  
    {  
        x+=0.1 ;  
        y+=0.1 ;  
        glVertex3f (x , y , 0.0f) ;  
    }  
glEnd () ;
```

# Prymitywy graficzne.

## Ustalenie rozmiaru punktu.

Domyślny rozmiar punktu to 1 pixel.

```
Void glPointSize(Glfloat size);
```

size - średnica w pixelach rysowanego punktu.

MS GL obsługuje wielkości od 0.05 do 10.0 co 0.125

```
Glfloat sizes[2];
```

```
Glfloat step;
```

```
glGetFloatv(GL_POINT_SIZE_RANGE, sizes);
```

```
glGetFlotav(GL_POINT_SIZE_GRANULARITY, &step);
```

Wywołania te dotyczą maszyny stanu OpenGL

# Prymitywy graficzne

## Rysowanie linii w 3 wymiarach.

```
glBegin(GL_LINES)
    glVertex3f(0.0,0.0,0.0);
    glVertex3f(50.0,50.0,50.0);
glEnd();
```

Musimy podawać parzystą liczbę wierzchołków.  
Inaczej ostatni wierzchołek będzie ignorowany.

## Łamana GL\_LINE\_STRIP

```
glBegin(GL_LINE_STRIP)
    glVertex3f(0.0,0.0,0.0);
    glVertex3f(50.0,50.0,50.0);
    glVertex3f(50.0,10.0,50.0);
glEnd();
```

# Prymitywy graficzne

```
Łamana zamknięta GL_LINE_LOOP  
glBegin(GL_LINE_LOOP)  
    glVertex3f(0.0,0.0,0.0);  
    glVertex3f(50.0,50.0,50.0);  
    glVertex3f(50.0,10.0,50.0);  
glEnd();
```

Po napotkaniu ostatniego punktu krzywa jest zamykana.

## Ustalenie grubości linii.

```
glLineWidth(Glfloat width)
```

Informacje o szerokości możemy uzyskać z maszyny stanu OpenGL.

# Prymitywy graficzne

```
glGetfloatv(GL_LINE_WIDTH_RANGE, sizes);  
glGetfloatv(GL_LINE_WIDTH_GGRANULARITY, &size);
```

Linie przerywane.

```
glEnable(GL_LINE_STIPPLE);
```

```
glLineStipple(Glint factor, Glushort pattern);
```

Wyłączamy za pomocą `glDisable()`;

pattern 16-bit

0000000011111111 = 0x00FF = 255;

factor służy jako mnożnik zwiększający szerokość wzoru.

# Prymitywy graficzne

## Rysowanie wielokątów w przestrzeni 3D.

Trójkąt.

```
glBegin(GL_TRIANGLES)
    glVertex3f(0.0,0.0,0.0);
    glVertex3f(25.0,25.0,0.0);
    glVertex3f(50.0,0.0,0.0);
glEnd();
```

Kolejność określa nam kierunek rysowanego trójkąta. Zgodnie ze wskazówkami zegara lub przeciwnie.

OpenGL zakłada, że wielokąty skierowane przeciwnie do wskazówek zegara są do nas zwrócone przodem. Jeśli chcesz to zmienić to

```
glFrontFace(GL_CW); GL_CCW
```

# Prymitywy graficzne

Rysowanie paska trójkątów.

`GL_TRIANGLE_STRIP`

Ilość wierzchołków może być większa od 3.

Kolejność wierzchołków nie jest zachowana.

Anticlockwise - kierunek przeciwny do wskazówek zegara musi być zachowany.

Wachlarze trójkątów.

`GL_TRIANGLE_FAN`

Opcja ta służy do tworzenia zbioru trójkątów o wspólnym punkcie. Pierwszy wierzchołek wyznacza wspólny punkt dla wachlarza trójkątów.

# Prymitywy graficzne (jednolite obiekty)

Prze tworzeniu złożonych obiektów musimy się zastanowić nad tym jaki jest nasz obiekt.

O kolorze wielokąta decyduje pierwszy wierzchołek w wielokącie, przy modelu w którym kolor jest jednolity.

## Wybór modelu koloru.

`glShadeModel(GL_FLAT);` - jednolity kolor

`glShadeModel(GL_SMOOTH);` - interpolacja kolorów pośrednich.

```
glBegin(GL_TRIANGLES)
    glColor3f(1.0f,0.0f,0.0f);
    glVertex3f(0.0,0.0,0.0);
    glVertex3f(25.0,25.0,0.0);
    glVertex3f(50.0,0.0,0.0);
glEnd();
```

# Prymitywy graficzne (bufor głębokości)

W OpenGL obiekt rysowany jako drugi zawsze przesłania obiekt pierwszy. Problem ten można usunąć dzięki technice zwanej buforem głębokości.

```
glEnable(GL_DEPTH_TEST);
```

Testowane są pixele aby wyznaczyć te które są bliżej od tych które są dalej.

W celu poprawy wydajności programu stosuje się nie rysowanie tylnych powierzchni.

# Prymitywy graficzne (usuwanie niewidocznych).

Eliminacja wielokątów skierowanych tyłem nazywa się usuwaniem niewidocznych powierzchni.

```
glEnable(GL_CULL_FACE);
```

Wielokąty mogą być rysowane jako siatki lub wypełnione.

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);  
glPolygonMode(GL_BACK, GL_FILL);
```

Można to ustawiać zarówno dla tyłu jak i przodu.

# Prymitywy graficzne (usuwanie niewidocznych)

## Czworokąty.

GL\_QUADS

GL\_QUAD\_STRIP - pasek wielokątów.

Ogólne wielokąty.

GL\_POLYGON może być używany do rysowania wielokąta o dowolnej liczbie krawędzi.

Wypełnianie jednolitych wielokątów.

## Mapowanie tekstur.

Określanie desenia.

```
glEnable(GL_POLYGON_STIPPLE);
```

```
glPolygonStipple(pBitmap);
```

# Prymitywy graficzne (desenie)

Wskaźnik do obszaru danych zawierający dane desenia.

```
Glubyte *pBitmap;
```

```
pBitmap=new Glubyte[16]
```

```
32bit x 32bit
```

```
delete[] pBitmap;
```

Warunki konstruowania wielokątów to

1. Nie wolno produkować wygiętych wielokątów.
2. Tylko wielokąty wypukłe.

## Prymitywy graficzne (krawędzie).

Figury wklęsłe można także konstruować w OpenGL składając je z kilku figur wypukłych.

Gdy chcemy zachować krawędzie zewnętrzne jako widoczne to wystarczy podać czy ta krawędź należeć będzie do krawędzi zewnętrznych czy też nie.

```
glEdgeflag(TRUE/FALSE) ;  
glVertex2f(-20.0f, 0.0f) ;
```

# Przekształcenia w 3D

Umożliwiają rzutowanie trójwymiarowych współrzędnych na dwuwymiarowy ekran.

Najpierw musimy określić położenie obserwatora. Są to bezwzględne współrzędne ekranowe.

Przekształcenie punktu obserwacji.  
Odpowiada ustawieniu i skierowaniu kamery na scenę.

Przekształcenie modelu.

# Przekształcenia w 3D (rzutowanie)

Rzutowanie definiuje bryłę widzenia i definiuje bryłę obcinania.

## **Równoległe**

`glOrtho(..)` równoległa bryła rzutowania

-----

## **Perspektywiczne**

Ostrosłup widzenia w kierunku oka.

**glFrustum** (`Gldouble`

`left, right, bottom, top, near, far`) - mnoży bieżącą macierz przez macierz rzutowania perspektywicznego.

## Przekształcenia w 3D (rzutowanie)

Znacznie lepszym rozwiązaniem jest funkcja  
`void gluPerspective (Gldouble fovy, Gldouble  
aspect, Gldouble zNear, Gldouble zFar)`

`fovy` - kąt pola widzenia w kierunku pionowym.

`Aspect` - stosunek wysokości do szerokości bryły

`zNear, zFar` - odległości od bliższej i dalszej  
płaszczyzny.

Dodatkowo co nam oferuje utility to funkcja

`gluLookAt (okoX, okoY, okoZ, punktX, punktY, punktZ,  
goraX, goraY, goraZ) ;`

Pozwala ona na umieszczenie oka kamery w dowolnym  
punkcie.

# Przekształcenia w 3D (rzutowanie)

Rzutowanie definiuje bryłę widzenia i definiuje bryłę obcinania.

Macierz widoku modelu 4x4

$$V * M = V'$$

## Translacje.

```
void glTranslatef(GLfloat x, GLfloat y, GLfloat z)
```

```
glTranslatef(0.0, 10.0, 0.0);
```

## Obroty.

**glRotatef**(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);

Kierunek obrotu jest określany w stopniach.

## **Skalowanie.**

**glScalef**(GLfloat x, GLfloat y, GLfloat z);

Ściskanie i rozciąganie.

## Macierz tożsamościowa(jednostkowa).

**glMatrixMode (GL\_MODELVIEW)**

Macierz modelu widoku.

**glLoadIdentity () ;**

Ładuje macierz tożsamościową do bieżącej macierzy.

## **Stos macierzy .**

Maksymalna wysokość stosu.

`glGet(GL_MAX_MODEL_VIEW_STACK_DEPTH) ;`

`glGet(GL_MAX_PROJECTION_STACK_DEPTH) ;`

`GL_STACK_OVERFLOW` - przepełnienie stosu

`GL_STACK_UNDERFLOW` - zdejmowanie ze stosu pustego.

# Przekształcenia w 3D (rzutowanie)

```
glPushMatrix () ;
```

Rotacje, translacje.

```
glPopMatrix () ;
```

Własne macierze transformacji.

```
GLfloat m[]={1.0f, 0.0f, 0.0f, 0.0f,  
             0.0f, 1.0f, 0.0f, 0.0f,  
             0.0f, 0.0f, 1.0f, 0.0f,  
             0.0f, 0.0f, 0.0f, 1.0f}
```

```
glMatrixMode(GL_MODELVIEW)
```

```
glLoadMatrix(m) // ładowanie macierzy jako bieżącej.
```

```
glMatrixMode(GL_MODELVIEW)
```

```
glMultMatrixf(m) ; // mnożenie macierzy
```

# Kolory w OpenGL

Dwie metody RGBA i color index.

RGBA - (red, green, blue, alpha)

Color index - wybrane kolory z większej palety

```
glColor4f(1.0f,1.0f,0.0f,1.0f);
```

```
glColori(12);
```

# Światła w scenie (fotorealizm)

## Światła:

- otoczenia (ambient light)
- rozproszone (diffuse)
- odbłyśki (specular)

Światła w naszej scenie obliczamy załączając

```
glEnable(GL_LIGHTING);
```

Samo załączenie nic nie daje, trzeba jeszcze określić model oświetlenia. Bez tego nasz obiekt jest 2-wymiarowy.

Najpierw określamy światło otoczenia.

```
GLfloat ambient[] = {1.0f,1.0f,1.0f,1.0f}
```

```
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,ambient);
```

# Światła w scenie (fotorealizm)

Właściwości materiałów.

Nasze wielokąty muszą odbijać światło.

```
GLfloat green[]={0.0f,0.75f,0.0f,1.0f};
```

```
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE,  
green);
```

Przy załączonym oświetleniu tylko w ten sposób można określić kolor naszego trójkąta.

```
glBegin(GL_TRIANGLES);
```

```
.....
```

```
glEnd();
```

# Światła w scenie (fotorealizm)

Generalnie teraz nie widać różnicy w naszej scenie. Faktyczna siła światła tkwi w rozpraszaniu i odbłyśkach.

Właściwości materiału można określać dla przedniej lub tylnej ściany.

`GL_FRONT, GL_BACK, GL_FRONT_AND_BACK`

Inna metoda to śledzenie kolorów z użyciem `glColor` jako funkcji określającej kolor.

```
glEnable(GL_COLOR_MATERIAL) ;  
glColorMaterial(GL_FRONT,  
GL_AMBIENT_AND_DIFFUSE) ;
```

# Światła w scenie (fotorealizm)

Do bardziej realistycznych scen potrzebne nam są światła umiejscowione.

Źródła te posiadają położenia, intensywności, kolory jak i kierunek padania.

Aby biblioteka wiedziała jak mocno rozświetlić daną płaszczyznę potrzebuje aby został określony wektor normalny do płaszczyzny naszego wielokąta.

# Wektor normalny

Mając trzy punkty w przestrzeni możemy obliczyć wektory określające przestrzeń 2D  $V_1$  i  $V_2$  następnie produkt  $V_1 \times V_2$  da nam wektor prostopadły do naszej płaszczyzny.

# Wektor normalny

```
void ReduceToUnit(float vector[3])
{
    float length;

    // Calculate the length of the vector
    length = (float)sqrt((vector[0]*vector[0]) +
                        (vector[1]*vector[1]) +
                        (vector[2]*vector[2]));

    if(length == 0.0f)
        length = 1.0f;

    vector[0] /= length;
    vector[1] /= length;
    vector[2] /= length;
}
```

# Wektor normalny

```
void calcNormal(float v[3][3], float out[3])
{
    float v1[3],v2[3];
    static const int x = 0, y=1, z=2;

    v1[x] = v[0][x] - v[1][x];
    v1[y] = v[0][y] - v[1][y];
    v1[z] = v[0][z] - v[1][z];

    v2[x] = v[1][x] - v[2][x];
    v2[y] = v[1][y] - v[2][y];
    v2[z] = v[1][z] - v[2][z];

    out[x] = v1[y]*v2[z] - v1[z]*v2[y];
    out[y] = v1[z]*v2[x] - v1[x]*v2[z];
    out[z] = v1[x]*v2[y] - v1[y]*v2[x];

    ReduceToUnit(out);
}
```

# Światła w scenie (fotorealizm)

Przygotowanie źródła światła.

```
GGLfloat ambient[]={0.3,0.3,0.3,1.0f}
```

```
GGLfloat diffuse[]={0.7,0.7,0.7,1.0}
```

```
glLightfv(GL_LIGHT0,GL_AMBIENT,ambient);
```

```
glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuse);
```

```
GGLfloat lpos[]={-50.0f,50.0f,100.0f,1.0f}
```

```
glLightfv(GL_LIGHT0,GL_POSITION,lpos);
```

```
glEnable(GL_LIGHT0);
```

Do ustalenia koloru możemy także użyć funkcji

```
glRGB(0,255,0);
```

# Światła w scenie (fotorealizm)

Przygotowanie źródła światła.

```
GLfloat specular[]={1.0,1.0,1.0,1.0f}
```

```
glLightfv(GL_LIGHT0, GL_SPECULAR, specular) ;
```

Ostatnia linia definiuje światło jasno białe dla odbłyśków.

## STOPIEŃ POŁYSKLIWOŚCI.

```
glMateriali(GL_FRONT, GL_SHININESS, 128) ;
```

Określa jak mała i skupiona będzie plama połysku.

# Światła w scenie (fotorealizm)

Domyślnie światło rozchodzi się w każdym kierunku. W celu przeniesienia światła do nieskończoności trzeba w ostatnim elemencie dać wartość zero.

Światło punktowe załączamy przez  
`glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 60.0f);`  
`glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 100.0f);`

`glPushAttrib(GL_LIGHTING_BIT);`  
Instrukcja zachowuje stan oświetlenia.  
`glPopAttrib();`  
Przywraca pierwotny stan oświetlenia.

# Cienie (fotorealizm)

Tworzenie cieni nie jest zaimplementowane w OpenGL. Można je jednak stworzyć programowo.

```
// Na podstawie równania płaszczyzny i położenia światła tworzy
// macierz rzutu cienia.
// Obliczona macierz jest umieszczona w tablicy destMat[][]
void MakeShadowMatrix(GLfloat points[3][3], GLfloat lightPos[4],
    GLfloat destMat[4][4])
{
    GLfloat planeCoeff[4];
    GLfloat dot;

    // Znalezienie współczynników równania płaszczyzny
    // Wyszukanie trzech pierwszych współczynników tak samo
    // jak przy znajdowaniu normalnej
    calcNormal(points, planeCoeff);

    // Znalezienie ostatniego współczynnika przez zastępowanie wstecz
    planeCoeff[3] = - (
        (planeCoeff[0]*points[2][0]) + (planeCoeff[1]*points[2][1]) +
        (planeCoeff[2]*points[2][2]));
}
```

# Cienie (fotorealizm)

```
// Iloczyn skalarny płaszczyzny i położenia światła
dot = planeCoeff[0] * lightPos[0] +
      planeCoeff[1] * lightPos[1] +
      planeCoeff[2] * lightPos[2] +
      planeCoeff[3] * lightPos[3];

// A teraz rzutowanie
// Pierwsza kolumna

destMat[0][0] = dot - lightPos[0] * planeCoeff[0];
destMat[1][0] = 0.0f - lightPos[0] * planeCoeff[1];
destMat[2][0] = 0.0f - lightPos[0] * planeCoeff[2];
destMat[3][0] = 0.0f - lightPos[0] * planeCoeff[3];

// Druga kolumna
destMat[0][1] = 0.0f - lightPos[1] * planeCoeff[0];
destMat[1][1] = dot - lightPos[1] * planeCoeff[1];
destMat[2][1] = 0.0f - lightPos[1] * planeCoeff[2];
destMat[3][1] = 0.0f - lightPos[1] * planeCoeff[3];
```

# Cienie (fotorealizm)

```
// Trzecia kolumna
    destMat[0][2] = 0.0f - lightPos[2] * planeCoeff[0];
    destMat[1][2] = 0.0f - lightPos[2] * planeCoeff[1];
    destMat[2][2] = dot - lightPos[2] * planeCoeff[2];
    destMat[3][2] = 0.0f - lightPos[2] * planeCoeff[3];

    // Czwarta kolumna
    destMat[0][3] = 0.0f - lightPos[3] * planeCoeff[0];
    destMat[1][3] = 0.0f - lightPos[3] * planeCoeff[1];
    destMat[2][3] = 0.0f - lightPos[3] * planeCoeff[2];
    destMat[3][3] = dot - lightPos[3] * planeCoeff[3];
}
```

# Listy wyświetlania

Listy wyświetlania przyspieszają wykonywanie programu OpenGL w przypadku dużej liczby wielokątów.

## PROCEDURA

```
glNewList(1, GL_COMPILE);  
... Kod OpenGL  
glEndList();
```

Następnie wywołanie tak przygotowanej listy wygląda następująco

```
glCallList(1);
```

Zamiast 1 można zastosować dowolny identyfikator  
Gluint list

# Grafika rastrowa

BITMAPY.

Obrazki 2 - kolorowe.

Kolor 0 - przezroczysty

1 - bieżący.

Do rysowania bitmap służy funkcja glBitmap

```
glBitmap(Glsizei width, Glsizei height,  
Glfloat xorig, Glfloat yorig, Glfloat xmove,  
Glfloat ymove, const Glubyte *bits)
```

xorig, yorig - położenie środka

xmove, ymove - przesunięcie w bitmapie.

Do ustalenia pozycji obrazka na naszym oknie służy.

```
glRasterPos2i(x, y);
```

# Grafika rastrowa

CZCIONKI BITMAPOWE.

Aby w OpenGL wyświetlić jakieś teksty na ekranie trzeba przygotować odpowiedni zestaw czcionek.

```
GluInt base;
```

```
HDC hdc;
```

```
base=glGenLists(96);
```

```
wglUseFontBitmaps(hdc, 32, 96, base);
```

Tworzy 96 bitmap znaków poczynając od kodu 32

```
glListBase(font-32);
```

```
glCallLists(strlen(s), GL_UNSIGNED_BYTE, s);
```

# Grafika rastrowa

Funkcja `glListBase` ustawia wartość bazową list wyświetlania.

`glCallLists` - wyświetla nam nasze czcionki wg. zadanego tekstu.

Czcionkę możemy ustawić funkcjami Win32

```
font=CreateFont(...)  
SelectObject(hdc, font);
```

# Grafika rastrowa.

## PIXMAPY

Obrazy zawierające więcej niż 2 kolory nazywamy pixmapami. Do rysowania pixmap służy funkcja

```
glDrawPixels(Glsizei width, Glsizei height,  
             GLenum format, GLenum type,  
             GLvoid *pixels)
```

format	type
GL_COLOR_INDEX	GL_BYTE
GL_RGB	GL_UNSIGNED_BYTE
GL_LUMINANCE	GL_BITMAP

## REMAPOWANIE KOLORÓW

```
glPixelTransfer(GL_RED_SCALE, 1.1);  
glPixelTransfer(GL_GREEN_SCALE, 1.1);  
glPixelTransfer(GL_BLUE_SCALE, 1.1);  
Kod rozjaśnia obraz RGGB o 10%
```

## SKALOWANIE PIXMAP

```
glPixelZoom(1.0,1.0);brak skalowania  
glPixelZoom(-1.0,1.0);odbicie lustrzane w poziomie  
glPixelZoom(0.2,0.2);zmniejszenie obrazka
```

## WYKRAWANIE OBSZARÓW

```
glPixelStore(GL_UNPACK_ROW_LENGTH, 640);  
glPixelStore(GL_UNPACK_SKIP_PIXELS, 100);  
glPixelStore(GL_UNPACK_SKIP_ROWS, 100);  
glDrawPixels(300, 300, GL_RGB, GL_UNSIGNED_BYTE, BitmapBits);
```

## ODCZYTYWANIE PIXMAP Z EKRANU

```
glReadPixels(GLint x, GLint y, GLsizei width,  
GLsizei height, GLenum format, GLenum type,  
const GLvoid *pixels);
```

# Mapowanie tekstur

Nakładanie obrazów na wielokąty w scenie.  
Akceleratory 3D same mapują tekstury.

## Definiowanie tekstur 1D

```
glTexImage1D(Glenum target, Glint level, Glint  
components, Glsizei width, Glint border, Glenum  
format, Glenum type, const Glvoid *pixels);
```

target - określa jaka tekstura powinna być  
zdefiniowana (GL\_TEXTURE\_1D)

level - poziom szczegółów obrazu zwykle 0 jak  
nie ma mipmapy.

Components - określa ilość wartości koloru dla  
jednego pixela RGB=3, RGBA=4

# Mapowanie tekstur

Width - długość ciągu pixeli musi stanowić potęgę 2.

border - pixle ramki

format - określa w jakim formacie zapisane są kolory (GL\_COLOR\_INDEX, GL\_RGB, GL\_RGBA, GL\_LUMINANCE);

## FILTRY POWIĘKSZENIA I POMNIEJSZENIA.

```
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

GL\_NEAREST, GL\_LINEAR, GL\_NEAREST\_MIPMAP\_NEAREST,

GL\_NEAREST\_MIPMAP\_LINEAR - liniowo interpolowana mipmapa

GL\_LINEAR\_MIPMAP\_LINEAR - liniowo interpolacja interpolowanych mipmap.

GL\_NEAREST - najlepsza wydajność.

# Mapowanie tekstur

## Definiowanie tekstur 2D

```
glTexImage2D(Glenum target, Glint level, Glint  
components, Glsizei width, Glsizei height,  
Glint border, Glenum format, Glenum type, const  
Glvoid *pixels)
```

Dodany jedynie height czyli wysokość mapowanego obrazka.

Width i height muszą być roamiaru potęg 2  
2, 4, 8, 16, 32, 64, 128, 256, 512, 1024

# Mapowanie tekstur

Rysowanie wielokątów nałożoną teksturą.

Teksturowanie trzeba załączyć

```
glEnable(GL_TEXTURE_nD);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,  
GL_DECAL);
```

Ustawienie teksturowania w tryb kafelków.

GL\_MODULATE - piksele tekstury filtrują kolory istniejących pikseli na ekranie

GL\_DECAL

GL\_BLEND - łączenie ze stałym kolorem.

Symulacja chmur.

# Mapowanie tekstur

## MIPMAPY

Pozwalają zastosować zmienny poziom szczegółów w zależności od odległości od obserwatora.

Użycie mniejszych obrazków zwiększa wiarygodność sceny jak i wydajność kodu.

Wprowadzamy mipmapy przez funkcje `glTexImage2D` podając `level` i odpowiadającą mu mapę.

```
glTexImage2D(GL_TEXTURE_2D, 0, 3, 128, 128, GL_RGB, GL_UNSIGNED_BYTE, imageBits0);
```

```
glTexImage2D(GL_TEXTURE_2D, 1, 3, 128, 128, GL_RGB, GL_UNSIGNED_BYTE, imageBits1);  
GL_LINEAR_MIPMAP_LINEAR;
```

# Kwadryki

Powierzchnie 2 stopnia.

GLU32.DLL - proste 3D kształty geometryczne.  
Cylindry, sfery, walce, dyski.

```
GLUquadricObj * quadObj;  
quadObj = gluNewQuadric();  
gluQuadricOrientation(quadObj, GLU_OUTSIDE);  
gluQuadricNormals (quadObj, GLU_SMOOTH);  
gluQuadricDrawStyle (quadObj, GLU_LINE);  
gluQuadricTexture (quadObj, GLU_TRUE);  
gluSphere (quadObj, sizeX, gridX, gridY);
```

grid - rozdzielczość elementów.

# Kwadryki

## Powierzchnie 2 stopnia.

`gluCylinder (quadObj, Rbase, Rtop, height, slices, stacks) ;`

Rbase - promień podstawy.

Rtop - promień wierzchołka.

Height - wysokość

slices, stacks - określają z ilu elementów składa się obiekt.

`gluDisk (quadObj, Rinner, Router, slices, loops) ;`

Rinner - promień wewnętrznego dysku.

Router - promień zewnętrznego dysku.

slices, loops - określają z ilu elementów składa się obiekt.

`gluDeleteQuadric (quadObj) ;`

# Przezroczystość

## Przesroczystość (BLENDING) .

Polega na łączeniu kolorów czyli kontrolowaniu wartości RGBA poszczególnych pixeli.

```
glEnable(GL_BLEND) .
```

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA  
);
```

Parametry określają sposoby łączenia kolorów źródłowego i docelowego.

Dla tych ustawień kolor alpha będzie określał przezroczystość.

# Mgła

## Mgła (FOG) .

Dodanie efektu atmosferycznego czyli zależnego od głębokości cieniowania każdego elementu sceny polega na wymieszaniu pewnego koloru z każdym wierzchołkiem lub obiektem tekstury.

```
glEnable (GL_FOG) ;
```

### Wybieramy rodzaj mgły.

```
glFogi (GL_FOG_MODE, GL_LINEAR) ;  
GL_EXP, GL_EXP2
```

### Wybieramy kolor mgły.

```
GLfloat fogColor[4] = {0.7F, 0.8F, 1.0F, 1.0F} ;  
glFogfv (GL_FOG_COLOR, fogColor) ;
```

# Mgła

## Mgła (FOG) .

```
glFogi (GL_FOG_MODE, GL_EXP);  
glFogf (GL_FOG_DENSITY, 0.005);
```

GL\_EXP - tego rodzaju mgła posiada jeszcze gęstość

```
glHint (GL_FOG_HINT, GL_DONT_CARE);
```

GL\_NICEST - powoduje że mgła jest obliczana dla każdego wierzchołka.

# Krzywe

## Krzywe

```
glMap1f(GLenum target, GLfloat u1, GLfloat u2,  
GLint stride, GLint order, const GLfloat *points)
```

### target-**typ**. współrzędnych

- `GL_MAP1_VERTEX_3` - wierzchołki (x, y, z).
- `GL_MAP1_COLOR_4` - kolory punktów w postaci (r, g, b, a).
- `GL_MAP1_TEXTURE_COORD_2` - współrzędne w obszarze tekstury (s, t).
- `GL_MAP1_NORMAL` - współrzędne wektora normalnego (nx, ny, nz).

**u1, u2** - to dziedzina funkcji

**Stride** - oznacza liczbę współrzędnych składających się na każdy punkt kontrolny

**order** - liczba punktów kontrolnych

# Krzywe

## Krzywe

```
glEnable(target) ;
```

```
void glMapGrid1f(GLint n, GLfloat u1, GLfloat  
u2)
```

N - określa ilość punktów krzywej do obliczeń, u1,u2 dziedzina

```
void glEvalMesh1(GLenum mode, GLint p1, GLint  
p2)
```

**mode** - GL\_POINTS, GL\_LINE

p1 i p2 oznaczają numery próbek, od których zaczynamy i na których kończymy obliczenia.

# Powierzchnie

## Powierzchnie

```
void glMap2f(GLenum target, GLfloat u1, GLfloat  
u2, GLint ustride, GLint uorder, GLfloat v1,  
GLfloat v2, GLint vstride, GLint vorder, const  
GLfloat *points);
```

MAP1 -> MAP2

```
void glMapGrid2f(GLint nu, GLfloat u1,  
GLfloat u2, GLint nv, GLfloat v1, GLfloat v2)
```

```
void glEvalMesh2(GLenum mode, GLint i1, GLint  
i2, GLint j1, GLint j2).
```

## Przykłady

$$f(x, y) = 1 / [(x + 1) (y + 1)]$$