Programowanie Grafiki Komputerowej

OpenGL Shading Language cz. 2 - podstawy języka GLSL

Grzegorz Pawiński

Katedra Informatyki, Politechnika Świętokrzyska w Kielcach



KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI



Politechnika Świętokrzyska

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY



Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

" Program Rozwojowy Potencjału Dydaktycznego Politechniki Świętokrzyskiej w Kielcach: *kształcenie w nowoczesnych obszarach techniki* "

Program Operacyjny Kapitał Ludzki, Priorytet IV Szkolnictwo wyższe i nauka, Działanie 4.1. Wzmocnienie i rozwój potencjału dydaktycznego uczelni oraz zwiększenie liczby absolwentów kierunków o kluczowym znaczeniu dla gospodarki opartej na wiedzy, Poddziałanie 4.1.1. Wzmocnienie potencjału dydaktycznego uczelni.

Umowa UDA–POKL.04.01.01-00395/09-00

Spis treści

1. Typy danych i zmienne
   - Proste typy danych
   - Złożone typy danych
   - Deklaracja zmiennych
   - Inicjalizacja zmiennych
   - Instrukcje sterujące i funkcje
2. Komunikacja OpenGL i GLSL
   - Przebieg komunikacji
   - Zmienna Uniform
   - Zmienna Attribute
   - Zmienna Varying
3. Proste shadery
   - Transformacje wierzchołków
   - Kolory wierzchołków
4. Bibliografia

Uwagi

- Animacje są uruchamine, gdy plik PDF jest wyświetlany na pełnym ekranie.
- Animacja jest uruchamiana dopiero po naciśnięciu spacji (program Adobe Reader).
- Klatki animacji wyświetlane są co 2 sekundy.
- Symulacje są uruchamiane po kliknięciu w ikonę z odpowiednim systemem operacyjnym. Przeglądarka wyświetli okno z ostrzeżeniem, należy go zaakceptować. W systemie Linux należy dodatkowo kliknąć dwukrotnie w nazwę programu.

Wymagania do uruchomienia symulacji

- Do uruchamiania symulacji z pliku PDF, wymagany jest program Adobe Reader w wersji 5.0 lub nowszej.
- W systemie Windows, należy umieścić bibliotekę *glut32.dll* w katalogu systemowym (zwykle *C:\Windows*), a w jego podkatalogu *System32* bibliotekę *glew32.dll*. Biblioteki można znaleźć TUTAJ.
- W systemie Linux, biblioteki należy umieścić w katalogu */usr/lib*. Biblioteki można znaleźć TUTAJ.
- Symulacje programów zostały napisane dla sprzętu komputerowego obsługującego OpenGL w wersji 2.0 lub nowszej.

Proste typy danych

Język GLSL ma podobną składnię do języka C/C++, jednak posiada mniejsza liczbę prostych i złożonych typów danych.

Typy proste

Typ	Opis
float	liczba zmiennoprzecinkowa ze znakiem
int	liczba całkowita ze znakiem
bool	typ logiczny, przyjmuje *true* lub *false*

Wektory złożone z typów prostych

Typ	Opis
vec{2,3,4}	wektor 2,3 lub 4 liczb typu *float*
ivec{2,3,4}	wektor 2,3 lub 4 liczb typu *int*
ivec{2,3,4}	wektor 2,3 lub 4 liczb typu *bool*

Macierze i samplery

Reprezentacja macierzy kwadratowych

Typ	Opis
mat2	macierz 2x2 liczb typu *float*
mat3	macierz 3x3 liczb typu *float*
mat4	macierz 4x4 liczb typu *float*

Specjalne typy danych zawierające dostęp do teksli

Typ	Opis
sampler{1,2,3}D	tekstury 1D, 2D lub 3D
samplerCube{2,3,4}	mapy sześcienne (ang. *cube maps*)
sampler{1,2}DShadow	mapy cieni (ang. *shadow maps*)

Typy złożone

W GLSL tablice oraz struktury są deklarowane przy użyciu tej samej składni co w języku C. W GLSL nie wolno inicjalizować tablic podczas deklaracji.

Deklaracja tablicy i struktury

```
float tab[4];
struct kierSwiatla {
   vec3 kierunek;
   vec3 kolor;
};
```

Deklaracja zmiennych

W GLSL nie ma niejawnej konwersji typów. Inicjalizacja, jak również rzutowanie typów wykonywane są za pomocą konstruktorów.

Inicjalizacja zmiennych i rzutowanie

```
int i = 2;                          // inicjalizacja wartością początkową
float f = (float)2;                 // BŁĄD, wymagany jest konstruktor
float g = float(i);                 // rzutowanie przy pomocy konstruktora
vec3 v = vec3(1.0, 2.0, 3.0); // inicjalizacja przy pomocy konstruktora
```

Inicjalizacja zmiennych

GLSL jest bardzo elastyczny pod względem inicjalizacji zmiennych używając do tego celu innych zmiennych. Wymaga jedynie podania tej samej ilości zmiennych.

Inicjalizacja z użyciem innych zmiennych

vec2 a = **vec2**(1.0,2.0);
vec2 b = **vec2**(3.0,4.0);
float c = 3.0;
vec4 d = **vec4**(a,b);
vec3 e = **vec3**(a,c);

Inicjalizacja macierzy

Macierze również można inicjalizować w ten sposób. Występuje kilka konstruktorów powalających na inicjalizację macierzy.

Inicjalizacja macierzy różnymi konstruktorami

mat4 *m1* = **mat4**(1.0) // *ustawienie wartości 1.0 na przekątnej macierzy*

vec2 *a* = **vec2**(1.0,2.0);
vec2 *b* = **vec2**(3.0,4.0);

mat2 *m2* = **mat2**(*a*,*b*); // *kolumny macierzy*

mat2 *m3* = **mat2**(1.0,2.0,3.0,4.0); // *podanie wszystkich elementów*

Inicjalizacja struktur

Podobnie jest ze strukturami, posiadają one swoje konstruktory do inicjalizacji wartości.

Inicjalizacja struktury

```
struct kierSwiatla {
   vec3 kierunek;
   vec3 kolor;
};

vec3 v1 = vec3(1.0, 2.0, 2.0);
vec3 v2 = vec3(2.0, 3.0, 4.0);

kierSwiatla ks = kierSwiatla(v1, v2);
```

Dostęp do pól wektora

Dostęp do pól wektora jest możliwy przy użyciu liter lub nawiasów kwadratowych, tj. do tablicy. Do pola struktury odwołujemy się po jego nazwie. W przypadku struktur dostępny jest inny zestaw liter, w zależności od przechowywanych wartości:

- *współrzędne* – x, y, z, w.
- *kolory* – r, g, b, a.
- *tekstury* – s, t, p, q.

Dostęp do pól wektora

vec4 a = **vec4**(1.0,2.0,3.0,4.0);

float $posX$ = $a.x$;
float $posY$ = $a[1]$;

vec2 $posXY$ = $a.xy$;

float $depth$ = $a.w$;

Instrukcje sterujące

Instrukcje sterujące, poza małymi wyjątkami, mają składnię taką samą jak w języku C. Instrukcje warunkowe ***if-else*** oraz iteracje ***while*** i ***do-while***, wymagają tylko podania w warunku wyrażenia logicznego (typu ***bool***). Instrukcje skoku ***continue*** i ***break*** działają tak samo. Inaczej zachowuje się funkcja ***discard***, która powoduje natychmiastowe zakończenie działania shadera, bez możliwości zapisania danych. Można jej używać tylko we fragment shaderze.

Funkcje

Shader jest zbudowany z funkcji. Każdy typ shadera musi posiadać przynajmniej funkcję ***void main()***. Można tworzyć własne funkcje. Tak samo jak w języku C, używa się słowa kluczowego RETURN do zwracania wartości z funkcji (jeśli coś zwraca). Funkcja może zwracać każdy typ, oprócz tablic.

Dla parametrów funkcji ustawia się tryb przekazywania danych:

- ***in*** - parametr przekazywany do funkcji (domyślny tryb).
- ***out*** - parametr wychodzący z funkcji.
- ***inout*** - parametr wejściowy i wyjściowy.

Funkcje

Shader jest zbudowany z funkcji. Każdy typ shadera musi posiadać przynajmniej funkcję ***void main()***. Można tworzyć własne funkcje. Tak samo jak w języku C, używa się słowa kluczowego RETURN do zwracania wartości z funkcji (jeśli coś zwraca). Funkcja może zwracać każdy typ, oprócz tablic.

Dla parametrów funkcji ustawia się tryb przekazywania danych:

- ***in*** - parametr przekazywany do funkcji (domyślny tryb).
- ***out*** - parametr wychodzący z funkcji.
- ***inout*** - parametr wejściowy i wyjściowy.

Uwagi

- Funkcje można przeciążać, jeśli lista ich parametrów jest różna.
- Nie wolno stosować rekurencji.

Przykład

Funckja skalująca wierzchołek względem punktu [0,0,0]

```
float skaluj (out vec4 v, in float skala) {
   v.x *= skala;
   v.y *= skala;
   v.z *= skala;
   return (v.w);
}
```

Komunikacja OpenGL i GLSL

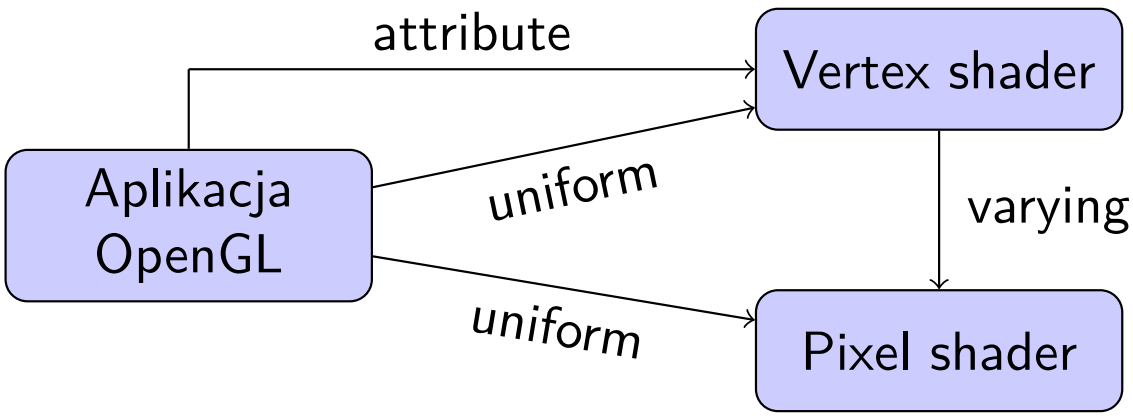
Aplikacja OpenGL może komunikować się z shaderami na kilka sposobów. Zawsze jest to komunikacja jednokierunkowa, ponieważ wynik działania shaderów jest przesyłany do renderowania. Shader ma dostęp do części zmiennych stanu OpenGL, dlatego np. światło lub kolory ustawione w aplikacji mogą zostać przez niego odczytane. Do tego celu wykorzystuje się kwalifikatory ***uniform*** i ***attribute***.

Kwalifikatory zmiennych

Kwalifikatory nadają zmiennym specjalne znaczenie. Podaje się je przed typem zmienej.
Wyróżniamy następujące kwalifikatory:

- ***const*** - wartość stała.
- ***attribute*** - zmienna globalna używana dla wierzchołków. Jest przekazywana z aplikacji OpenGL'a do vertex shadera, dla którego jest zmienną tylko do odczytu.
- ***uniform*** - zmienna globalna używana dla prymitywów. Umożliwia komunikację aplikacji OpenGL z vertex lub fragment shaderem. Jest tylko do odczytu.
- ***varying*** - używana do komunikacji pomiędzy vertex (zapis) i fragment (odczyt) shaderem.

Kierunki komunikacji



Zmienna uniform

Zmienna ***uniform*** nie może być zmieniana pomiędzy blokiem *glBegin/glEnd*, dlatego nie nadaje się do przechowywania atrybutów wierzchołków. Wykorzystywana jest dla wartości, które pozostają stałe dla prymitywu, ramki, a nawet dla całej sceny.

Zmienna deklarowana jest w shaderze, a w aplikacji OpenGL można uzyskać do niej referencję i przypisać wartość. W aplikacji OpenGL należy najpierw podlinkować program - *glLinkProgram()*, a następnie go użyć - *glUseProgram()*.

Pobranie lokalizacji zmiennej uniform

Aby uzyskać dostęp do zmiennej *uniform*, należy najpierw pobrać jej lokalizację w pamięci. Funkcja zwraca lokalizację zmiennej.

Pobranie lokalizacji zmiennej *uniform*

GLint **glGetUniformLocation**(GLuint *program*, char * *name*)

Parametry funkcji

- *program* – referencja do programu.
- *name* – nazwa zmiennej.

Przypisywanie wartości zmiennej uniform

Po uzyskaniu lokalizacji zmiennej *uniform*, można jej przypisać wartość. Do ustawienia wartości dostępna jest rodzina funkcji, uzależniona od typu (f, i) oraz ilości (1, 2, 3, 4) zmiennych.

Przypisywanie wartości zmiennej uniform

VOID **glUniform1f**(GLINT *location*, GLFLOAT *v0*)
VOID **glUniform2f**(GLINT *location*, GLFLOAT *v0*, GLFLOAT *v1*)
VOID **glUniform3f**(GLINT *location*, GLFLOAT *v0*, GLFLOAT *v1*, ..)
VOID **glUniform3f**(GLINT *location*, GLFLOAT *v0*, GLFLOAT *v1*, .. , ..)
GLINT **glUniform**{1,2,3,4}**fv**(GLINT *location*, GLSIZEI *count*, GLFLOAT *v)

Parametry funkcji

- *location* – lokalizacja zmiennej.
- *v0, v1, v2, v3* – nazwa zmiennej.
- *count* – liczba elementów w tablicy.
- *v* – tablica wartości typu float.

Przypisywanie wartości dla macierzy

Przypisywanie wartości dla macierzy

GLint **glUniformMatrix**{1,2,3,4}**fv**(GLint *location*, GLsizei *count*, GLboolean *transpose*, GLfloat *v);

Parametry funkcji

- *location* – lokalizacja zmiennej.
- *count* – liczba modyfikowanych macierzy (zwykle 1).
- *transponse* – przyjmuje wartość 1, jeżeli macierz jest transponowana (w kolejności wierszy), 0 w przeciwnym razie.
- *v* – tablica wartości typu float.

Uwagi

Jeśli program jest ponownie linkowany, przypisane wartości zostaną utracone. Zmienne otrzymają wartość 0.

Przykład użycia zmiennych uniform

Kod źródłowy shadera

```
uniform float intensywnosc;
uniform vec4 kolor;
uniform float p[2];
uniform vec4 tabKolorow[3];
```

Kod źródłowy aplikacji OpenGL

```
//lokalizacje
GLint loc1,loc2,loc3,loc4;
float inten = 0.92;
float k[4] = {0.7,0.7,0.7,1.0};
float prog[2] = {0.25,0.25};
float kolory[12] = {0.1,0.2,0.3,4.0,
                    0.2,0.3,0.4,5.0,
                    0.3,0.4,0.5,6.0};

//pobranie lokalizacji
loc1 = glGetUniformLocation(p,"intensywnosc");
//przypisanie do zmiennej
glUniform1f(loc1,inten);

loc2 = glGetUniformLocation(p,"kolor");
glUniform4fv(loc2,1,k);

loc3 = glGetUniformLocation(p,"p");
glUniform1fv(loc3,2,prog);

loc4 = glGetUniformLocation(p,"tabKolorow");
glUniform4fv(loc4,3,kolory);
```

Zmienna attribute

Zmienne ***attribute*** są używane, gdy wymagane są zmiany wartości zmiennej dla każdego wierzchołka. Mogą być modyfikowane w dowolnej chwili. Przechowują dane o wierzchołkach, dlatego dostępne są tylko wewnątrz *vertex shadera*, w trybie do odczytu. Ich lokalizację można pobrać dopiero po użyciu programu - *glUseProgram()*.

Pobranie lokalizacji zmiennej attribute

Dostęp do zmiennej *attribute* uzyskiwany jest poprzez pobranie jej lokalizacji w pamięci. Funkcja zwraca lokalizację zmiennej.

Pobranie lokalizacji zmiennej *attribute*

GLint **glGetAttribLocation**(GLuint *program*, char * *name*)

Parametry funkcji

- *program* – referencja do programu.
- *name* – nazwa zmiennej.

Przypisywanie wartości zmiennej attribute

Mając lokalizację zmiennej *attribute*, można jej przypisać wartość dla każdego wierzchołka. Do ustawienia wartości dostępna jest rodzina funkcji, uzależniona od typu (f, i) oraz ilości (1, 2, 3, 4) zmiennych.

Przypisywanie wartości zmiennej attribute

VOID **glVertexAttrib1f**(GLINT *location*, GLFLOAT *v0*)
VOID **glVertexAttrib2f**(GLINT *location*, GLFLOAT *v0*, GLFLOAT *v1*)
VOID **glVertexAttrib3f**(GLINT *location*, GLFLOAT *v0*, GLFLOAT *v1*, ..)
VOID **glVertexAttrib3f**(GLINT *location*, GLFLOAT *v0*, GLFLOAT *v1*, .. , ..)
GLINT **glVertexAttrib**{1,2,3,4}**fv**(GLINT *location*, GLFLOAT *v)

Parametry funkcji

- *location* – lokalizacja zmiennej.
- *v0, v1, v2, v3* – nazwa zmiennej.
- *v* – tablica wartości typu float.

Przykład użycia zmiennej attribute

Kod źródłowy vertex shadera

```
attribute float wysokosc;
```

Kod źródłowy aplikacji OpenGL

```
//pobranie lokalizacji zmiennej
GLint loc = glGetAttribLocation(p,"wysokosc");

glBegin(GL_LINES);

    //nadanie wartosci
    glVertexAttrib1f(loc,1.0);
    glVertex2f(-1,1);

    //nadanie wartosci
    glVertexAttrib1f(loc,2.0);
    glVertex2f(1,1);

glEnd();
```

Zmienna varying

Zmienna ***varying*** jest wykorzystywana do przekazywania danych z *vertex shadera* do *pixel shadera*. *Vertex shader* oblicza i zapisuje dane w zmiennej. Dane są interpolowane dla każdego piksela, a następnie *pixel shader* może je odczytać. Nie może do zmiennej nic zapisać.

Zmienne stosowane są np. podczas obliczania oświetlenia dla każdego piksela. Aby obliczyć oświetlenie, *pixel shader* potrzebuje parametrów normalnej, które są dostępne tylko z poziomu *vertex shadera*.

Deklaracja zmiennej varying

Zmienne muszą być identycznie zadeklarowane w obu shaderach. W ten sposób zostaną ze sobą automatycznie połączone i będzie można odczytywać dane w *pixel shaderze*. OpenGL posiada zestaw predefiniowanych zmiennych *varying* – np. kolor każdego wierzchołka.

Deklaracja zmiennej varying

VARYING FLOAT *normalna*;

Transformacja wierzchołka

Do transformacji wierzchołków w OpeGL'u wykorzystuje się ***macierz rzutowania*** i ***widok modelu***. Wyjściowa pozycja wierzchołka obliczana jest według następującego wzoru:

Oblcizenie transformacji wierzchołka

$$wierzcholek_{trans} = macierz_{rzutowania} * widok_{modelu} * wierzcholek$$

GLSL ma dostęp do *macierzy rzutowania*, *widoku modelu*, a także do współrzędnych wierzchołka za pomocą predefiniowanych zmiennych:

Dostęp do *widoku modelu* i *macierzy rzutowania* z GLSL

```
UNIFORM MAT4 gl_ModelViewMatrix; //widok modelu
UNIFORM MAT4 gl_ProjectionMatrix; // macierz rzutowania
UNIFORM MAT4 gl_ModelViewProjectionMatrix // ich iloczyn
```

Współrzędne aktualnie analizowanego wierzchołka

```
ATTRIBUTE VEC4 gl_Vertex
```

Przekazanie współrzędnych wierzchołka

Wynikowa pozycja wierzchołka musi zostać zapisana w predefiniowanej zmiennej. W przeciwnym wypadku, obliczenia nie będą miały wpływu na wierzchołek. Jest to minimum, jakie musi wykonać każdy *vertex shader*.

Transofrmacje wierzchołka w *vertex shaderze*

```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Uwagi

Nie jest konieczne mnożenie *widoku modelu* i *macierzy rzutowania* dla każdego wierzchołka, dlatego lepszym rozwiązaniem jest skorzystanie ze zmiennej *gl_ModelViewProjectionMatrix*.

Przekazanie współrzędnych wierzchołka

Proces transformacji wierzchołków jest zadaniem kluczowym dla kart graficznych. W GLSL dostępna jest funkcja, która go optymalizuje i gwarantuje, że wynik transformacji będzie zawsze taki sam, jak dla oryginalnych funkcji OpenGL'a.

Optymalizacja transofrmacji wierzchołka w *vertex shaderze*

```
void main()
{
    gl_Position = ftransform();
}
```

Uwagi

Operacje, które nie zostaną zaimplementowane w *vertex shaderze* (np. wyliczenie oświetlenia), nie będą wykonane przez standardowe funkcje OpenGL'a.

Ustawienie wynikowego koloru piksela

Wynikowy kolor piksela jest ustawiany w *pixel shaderze*. Wykorzystywana jest do tego celu predefiniowana zmienna *gl_FragColor*.

Ustawienie wynikowego koloru piksela w *pixel shaderze*

```
void main()
{
    gl_FragColor = vec4(0.4,0.4,0.8,1.0);
}
```

Przekazanie koloru do pixel shadera

Pixel shader nie ma dostępu do źródłowego koloru wierzchołka. Kolor wierzchołka musi zostać przekazany z *vertex shadera*.

Odbywa się to w kilku krokach:

1. Aplikacja OpenGL ustawia kolor wierzchołka funkcją *glColor()*.
2. *Vertex shader* otrzymuje go w zmiennej *attribute gl_Color*.
3. Następnie wylicza kolor dla przedniej oraz tylniej strony i zapisuje go w zmiennych *varying gl_FrontColor* i *varying gl_BackColor*.
4. *Fragment shader* może odczytać interpolowany kolor danego piksela (z automatycznym określeniem strony przedniej i tylnej) za pomocą zmiennej *varying gl_Color* i na jego podstawie ustawia zmienną *gl_FragColor*.

Predefiniowane zmienne związane z kolorem

Predefiniowane zmienne związane z kolorem

ATTRIBUTE VEC4 *gl_Color; //kolor przekazywany z OpenGL'a*

VARYING VEC4 *gl_FrontColor; // vertex shader zapisuje kolor przedniej strony*
VARYING VEC4 *gl_BackColor; // vertex shader zapisuje kolor tylniej strony*

VARYING VEC4 *gl_Color; // fragment shader może odczytać wartość koloru*

Uwagi

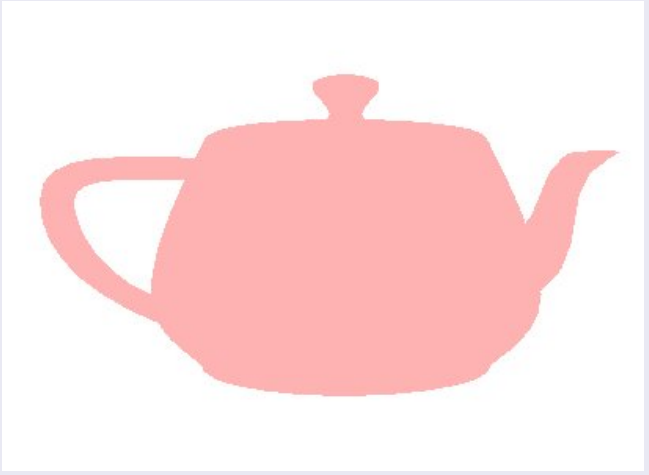
Wyjątkowo dopuszcza się występowanie w *vertex shaderze* dwóch zmiennych o tej samej nazwie, ponieważ zmienna *attribute gl_Color* jest widoczna tylko w *vertex shaderze*, a *varying gl_Color* w *pixel shaderze*.

Przykład

Przykładowy program demonstruje etapy przekazywania koloru z aplikacji OpenGL oraz dostęp do zmiennych związanych z kolorem w *vertex shaderze* i *pixel shaderze*. Efektem jest wyświetlenie obiektu 3D, dla którego kolor został ustawiony wewnątrz aplikacji OpenGL.

Przykładowy program





Bibliografia


R. Fernando, Mark J. Kilgard:
Język Cg. Programowanie grafiki w czasie rzeczywistym..
Wydawnictwo Helion, Gliwice, 2003.


OpenGL Shading Language.
http://www.opengl.org/documentation/glsl


Lighthouse 3D - GLSL Tutorial
http://www.lighthouse3d.com/opengl/glsl



KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI



Politechnika Świętokrzyska

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY



Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

" Program Rozwojowy Potencjału Dydaktycznego Politechniki Świętokrzyskiej w Kielcach: *kształcenie w nowoczesnych obszarach techniki* "

Program Operacyjny Kapitał Ludzki, Priorytet IV Szkolnictwo wyższe i nauka, Działanie 4.1. Wzmocnienie i rozwój potencjału dydaktycznego uczelni oraz zwiększenie liczby absolwentów kierunków o kluczowym znaczeniu dla gospodarki opartej na wiedzy, Poddziałanie 4.1.1. Wzmocnienie potencjału dydaktycznego uczelni.

Umowa UDA–POKL.04.01.01-00395/09-00