Uniwersytet Mikołaja Kopernika w Toruniu
Wydział Fizyki, Astronomii i Informatyki Stosowanej
Instytut Nauk Technicznych

Rafał Balcerowski
nr albumu: 289524

Praca inżynierska
na kierunku informatyka stosowana

Zaawansowana grafika 3D w OpenGL 4 i GLSL: shader teselacji i inne shadery oraz cieniowanie Fresnela i inne techniki oświetlenia

Opiekun pracy dyplomowej
dr hab. Jacek Matulewski, prof. UMK
Katedra Informatyki Stosowanej

Toruń 2023

Spis treści

Abstract

Title: *Advanced 3D graphics techniques in OpenGL 4 and GLSL: tessellation shaders, Fresnel shading and other lighting techniques.*

The purpose of this diploma thesis is to present the theory and implementation of some 3D graphics techniques in OpenGL and GLSL, in particular two of them. The first technique is tessellation which uses tessellation shaders. The second technique is Fresnel shading accompanied by environmental mapping, which requires the use of a fragment shader. In addition, several other techniques and effects were also presented. The text takes the form of a tutorial in which the issues are presented step-by-step with the provision of changes introduced to the code and the appropriate commentary.

Wstęp

Niniejsza praca dotyczy programowania grafiki trójwymiarowej w OpenGL 4, a w szczególności programowania shaderów karty graficznej w języku GLSL. Praca ta kontynuuje rozwój kodu z podręcznika "*Grafika 3D czasu rzeczywistego: Nowoczesny OpenGL"* autorstwa profesora Jacka Matulewskiego [1] i omawia następujące zagadnienia:

- shader teselacji i inne shadery,
- cieniowanie Fresnela, mapowanie środowiska i inne techniki związane z oświetleniem.

OpenGL (ang. *Open Graphics Library*) to biblioteka programistyczna, która zapewnia interfejs programistyczny do tworzenia grafiki. Jest jednym z najpopularniejszych rozwiązań w dziedzinie. Jest szeroko stosowana w aplikacjach wymagających renderowania grafiki trójwymiarowej (3D). Umożliwia także programowanie w różnych językach programowania. [2] W niniejszej pracy kod jest napisany w C++.

GLSL (ang. *OpenGL Shading Language*) jest językiem programowania stworzonym do programowania programów cieniujących w OpenGL. Program cieniujący (shader) to program wykonywany na procesorze karty graficznej, który wykonuje różne obliczenia i przekształcenia grafiki 3D, takie jak: przetwarzanie wierzchołków, teselacja, cieniowanie czy oświetlenie. Składnia tego języka jest wzorowana na języku C. [3]

Praca ma formę tutorialu. To oznacza, że dwa pierwsze rozdziały pracy zawierają wstęp teoretyczny do wybranego zagadnienia oraz przedstawienie krok po kroku sposobu jego implementacji, co umożliwi odtworzenie kodu i działającego programu przez osobę czytającą.

Pierwszy rozdział jest poświęcony teselacji. Rozpoczyna się od wprowadzenia w temat. Następnie zaprezentowana jest implementacja rozwiązania z wykorzystaniem wszystkich pięciu shaderów potoku graficznego. Technika teselacji jest przedstawiona na przykładzie sfery, choć z początku stworzonej z tak małej liczby punktów, że przypomina raczej wielościan. Dopiero dzięki działaniom przeprowadzonym w shaderach teselacji i geometrii przybiera kształt właściwej sfery.

Drugi rozdział opisuje cieniowanie Fresnela i mapowanie środowiska. Po krótkim wprowadzeniu w zagadnienie, implementowany jest efekt Fresnela, który imituje specyficzne efekty świetlne na granicy dwóch ośrodków. Następnie opisana jest obsługa wielu zestawów shaderów, co umożliwi utworzenie skyboksa i poprawienie wyglądu sceny. Następnie prezentowany jest efekt mapowania środowiska oraz efekt dyspersji chromatycznej.

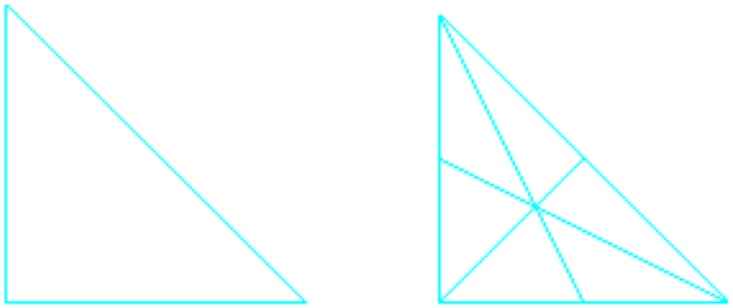
Natomiast trzeci rozdział zawiera zbiór pytań (z odpowiedziami) oraz zadań. Pytania kontrolne umożliwią sprawdzenie poziomu zaznajomienia z tematem, a zadania w praktyczny sposób rozwiną zrozumienie działania i implementacji omówionych w pracy technik oraz efektów.

Praca ma formę tutorialu, którego celem jest przedstawienie wymienionych wyżej technik grafiki 3D. Za najlepszy sposób prezentacji tych zagadnień została uznana implementacja krok-po-kroku z przedstawieniem zmian wprowadzanych do kodu oraz odpowiednim komentarzem. Aby możliwe było sprawdzenie nabytej wiedzy, przygotowany został również zbiór pytań oraz zadania do samodzielnego wykonania.

Rozdział 1.
Teselacja

Teoria

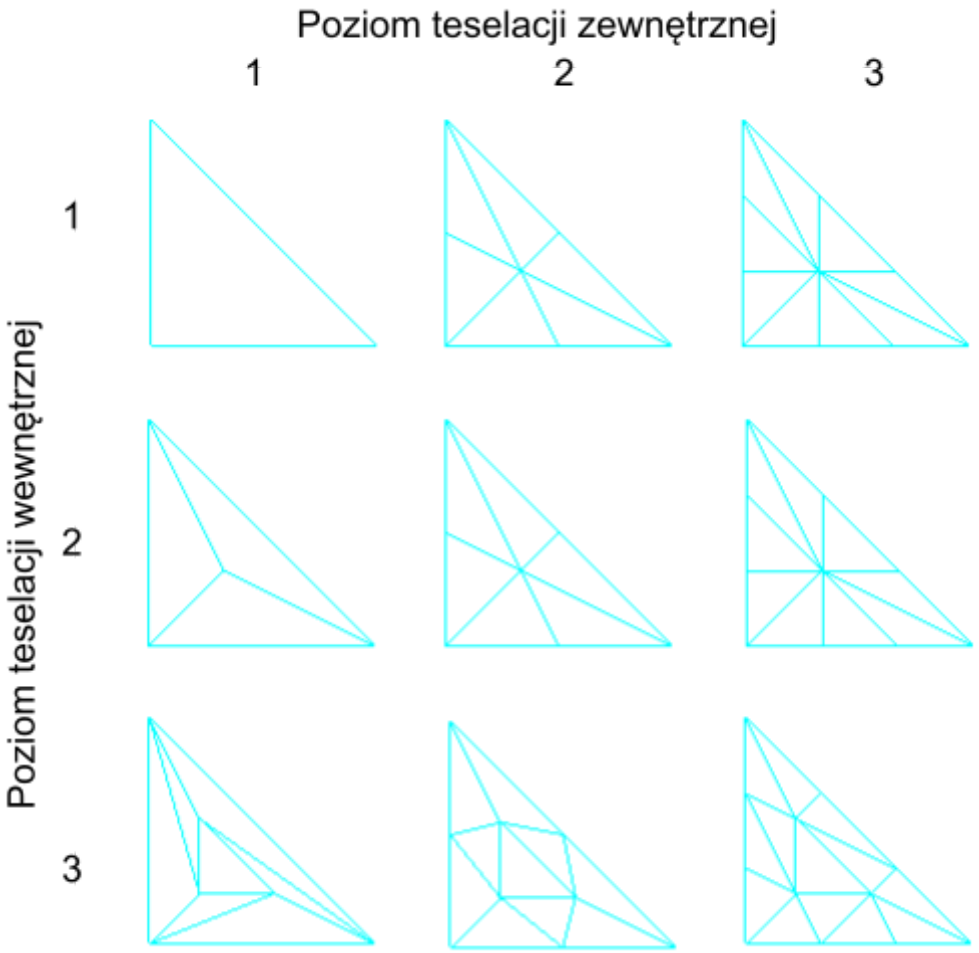
Teselacja jest techniką zagęszczania siatki werteksów obiektów trójwymiarowych w grafice komputerowej [4]. Polega na dzieleniu wielokątów, z których składa się siatka na mniejsze, co zapewnia poprawę odwzorowania szczegółów i wizualnej jakości sceny. Przykład teselacji widoczny jest na rysunku 1.1.



Rysunek 1.1 Trójkąt przed teselacją i po teselacji

Teselacja jest wykonywana przez kartę graficzną. W OpenGL odbywa się to w potoku renderowania, w trzech etapach. Wierzchołki dzielone są na prymitywy specjalnego typu — płaty (ang. *patches*). Liczba wierzchołków wchodzących w skład płatu może być zmieniana. Przy wyborze trzech otrzymywać będziemy prymitywy - trójkąty.

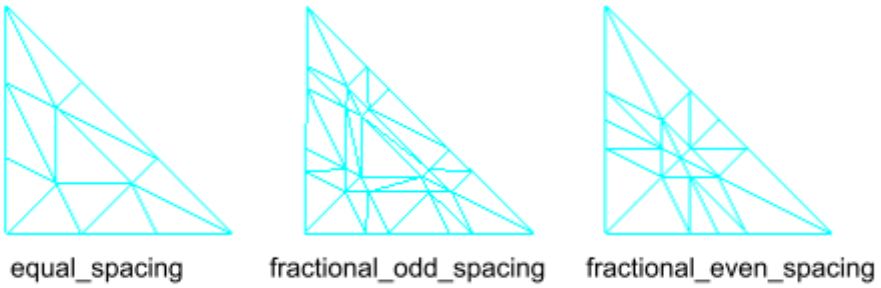
Pierwszym etapem teselacji jest shader kontroli teselacji (ang. *Tessellation Control Shader*). Jest on programowalny i określa liczbę teselacji, jaką należy wykonać. Programista może ustawić tutaj rozmiar płatu oraz poziom teselacji zewnętrznej i wewnętrznej. Z poziomów tych wynika, ile nowych wierzchołków zostanie wygenerowanych i w których miejscach. Teselacja zewnętrzna może być ustawiona oddzielnie dla każdego boku. Określa ona na ile części bok ma zostać podzielony poprzez dodanie wierzchołków. Natomiast teselacja wewnętrzna dodaje wierzchołki wewnątrz płata tak, aby liczba odcinków pomiędzy wejściowymi wierzchołkami była zgodna z ustawionym parametrem (por. rys. 1.2). [5]



Rys. 1.2. Otrzymane prymitywy w zależności od poziomów teselacji zewnętrznej i wewnętrznej.

Drugim etapem jest generator teselowanych prymitywów (ang. *tesselation primivite generator*), który otrzymuje na wejściu płat i generuje z niego konkretny prymityw (np. trójkąt). Ten etap nie jest programowalny, ale wpływ na jego działanie mają parametry ustawione w poprzedzającym go shaderze kontroli teselacji (poziomy teselacji) oraz w następującym po nim shaderze ewaluacji teselacji (typ prymitywu, rozstaw teselacji, kierunek nawijania).

Trzecim etapem teselacji jest shader ewaluacji teselacji (ang. *Tessellation Evaluation Shader*), który oblicza położenia i inne dane wygenerowanych wierzchołków. Etap ten również jest programowalny. Od niego zależy to, jakiego typu prymitywy będą generowane. Mogą to być izolinie, trójkąty lub czworokąty (ang. *isolines*, *triangles*, *quads*). Kolejnym ważnym parametrem ustawianym w tym shaderze jest rozstaw (ang. *spacing*) pomiędzy generowanymi w teselacji wierzchołkami. Dostępne są trzy opcje, których efekt działania można zobaczyć na rys. 1.3. Ostatnim opcjonalnym parametrem jest kierunek nawijania, który może oczywiście przyjąć dwie wartości: kierunek zgodny z kierunkiem wskazówek zegara albo odwrotny. Shader ewaluacji teselacji przyjmuje wygenerowane prymitywy, w których dla każdego wierzchołka ustawia położenie i inne dane. [6]



Rys. 1.3 Trójkąty teselowane z różnym ustawieniem parametru rozstawu wierzchołków.

Implementacja

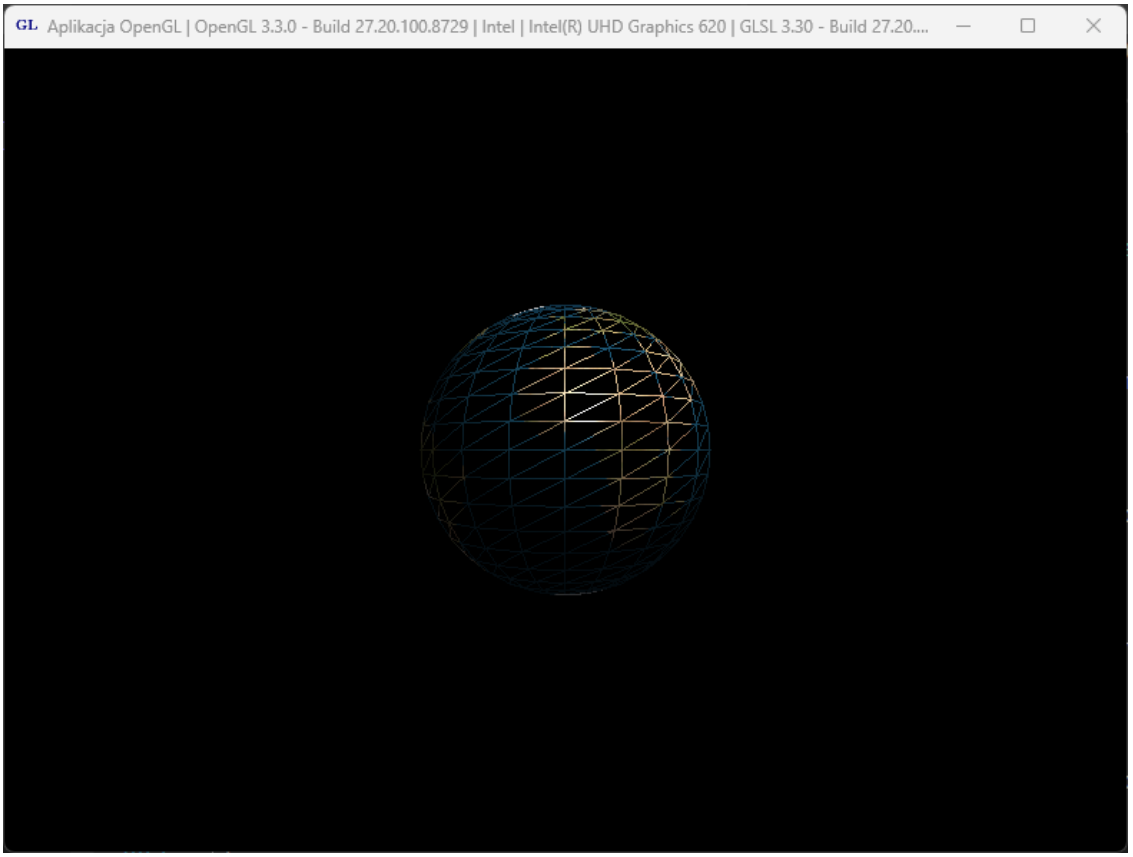
Aby zaimplementować własny przebieg teselacji, zmodyfikujemy i rozbudujemy kod projektu startowego[1], do którego dodamy dwa pliki shaderów.

W pierwszej kolejności przejdźmy do pliku *OknoGL.cpp*, do metody `UstawienieSceny`. W dowolnym jej miejscu dopiszmy wywołanie funkcji `glPolygonMode` z argumentami `GL_FRONT_AND_BACK` i `GL_LINE` (listing 1.1), co włącza tryb renderowania samych krawędzi. Tryb ten umożliwia wygodną obserwację efektów zmian wprowadzonych w procesie teselacji (por. rysunek 1.4).

Listing 1.1. Dodane wywołanie funkcji `glPolygonMode` na końcu metody `UstawienieSceny`. Plik: *OknoGL.cpp*.

```
        GLint parametrMacierzNormalnych = glGetUniformLocation(
            idProgramuShaderow, "macierzNormalnych");
        macierzNormalnych = macierzŚwiata.Odwrotna().Transponowana();
        macierzNormalnych.ZwiążZIdentyfikatorem(
            parametrMacierzNormalnych, true);

        UstawParametryŹródłaŚwiatła(
            Wektor3(1, 2, 2), Wektor4(1, 1, 1, 1),
            Wektor4(1, 1, 1, 1), Wektor4(1, 1, 1, 1));
        UstawParametryMateriału(
            Wektor4(0.1f, 0.1f, 0.1f, 1), Wektor4(1, 1, 1, 1),
            Wektor4(0, 1, 0, 1), 100);
        UstawParametryOsłabieniaOświetlenia(false, Wektor3(1, 0, 1));

        // (1) Włączenie trybu renderowania krawędzi.
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
}
```

[1] Projekt Visual Studio 2022 dostępny pod adresem
http://jacekmatulewski.fizyka.umk.pl/dydaktyka/3d/glsl/rbalcerowski/rbalcerowski_00_START_VS2019.zip



Rysunek 1.4. Bryła widoczna w trybie renderowania krawędzi.

Kolejnym krokiem jest dodanie klasy aktora reprezentującego sferę, która będzie rysowana z użyciem płatów zamiast trójkątów. Jak wspomniałem wyżej, płat to specjalny rodzaj prymitywu, który jest poddawany teselacji. Jednak jeszcze zanim zdefiniujemy nowego aktora, utwórzmy najpierw ogólną klasę dla aktorów podlegających teselacji. Będzie ona dziedziczyć z istniejącej już klasy bazowej `Aktor`. Zmiany wprowadzamy w plikach *Aktor.h* (listing 1.2) i *Aktor.cpp* (listing 1.3). Zmiany w kodzie wyróżniane są szarym tłem.

Listing 1.2. Dodanie deklaracji klasy bazowej dla aktorów złożonych z płatów. Plik: *Aktor.h*.

```
class TeselowanyAktor : public Aktor
{
protected:
	unsigned int liczbaIndeksów;
	unsigned int vbo_indeksy;

private:
	void InicjujBuforIndeksów();
	void UsuńBuforIndeksów();

protected:
	virtual unsigned int TwórzTabliceIndeksów(GLuint*& indeksy) = 0;

public:
	virtual void Inicjuj(GLuint atrybutPołożenie, GLuint atrybutNormalna,
GLuint atrybutWspółrzędneTeksturowania, GLuint atrybutKolor);
	virtual void Rysuj();
```

```
        TeselowanyAktor();
        virtual ~TeselowanyAktor();
};
```

Listing 1.3. Definicje metod klasy `TeselowanyAktor`. Plik: *Aktor.cpp*.

```
TeselowanyAktor::TeselowanyAktor()
:Aktor(), liczbaIndeksów(-1)
{
}

TeselowanyAktor::~TeselowanyAktor()
{
        Aktor::~Aktor();
        UsuńBuforIndeksów();
}

void TeselowanyAktor::Inicjuj(GLuint atrybutPołożenie, GLuint
atrybutNormalna, GLuint atrybutWspółrzędneTeksturowania, GLuint atrybutKolor)
{
        Aktor::Inicjuj(atrybutPołożenie, atrybutNormalna,
atrybutWspółrzędneTeksturowania, atrybutKolor);

        glPatchParameteri(GL_PATCH_VERTICES, 3); // Typ prymitywu i liczba
wierzchołków prymitywu.

        InicjujBuforIndeksów();
}

void TeselowanyAktor::InicjujBuforIndeksów()
{
        glGenBuffers(1, &vbo_indeksy);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo_indeksy);

        GLuint* indeksy = NULL;
        liczbaIndeksów = TwórzTabliceIndeksów(indeksy);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, liczbaIndeksów*sizeof(GLuint),
indeksy, GL_STATIC_DRAW);
        delete[] indeksy;
}

void TeselowanyAktor::UsuńBuforIndeksów()
{
        glDeleteBuffers(1, &vbo_indeksy);
}

void TeselowanyAktor::Rysuj()
{
        glBindVertexArray(vao);
        glBindBuffer(GL_ARRAY_BUFFER, vbo_indeksy);
        glDrawElements(GL_PATCHES, liczbaIndeksów, GL_UNSIGNED_INT, 0);
}
```

Wywołanie funkcji `glPatchParameteri` widoczne na listingu 1.3 jest jedną z dwóch najważniejszych zmian w kodzie C++. Określa ono, że typ prymitywu, za którego pomocą ma być renderowany obiekt, to płat o trzech wierzchołkach, z którego będzie można utworzyć trójkąty, których będzie więcej niż wejściowa liczba płatów. Druga ważna zmiana znajduje się w metodzie `Rysuj` i jest to wywołanie funkcji `glDrawElements` z pierwszym

argumentem równym `GL_PATCHES`, co służy przesłaniu wcześniej określonych (3-wierzchołkowych) płatów do potoku graficznego.

Następnie dodajemy do pliku *Aktor.h* deklarację właściwej już klasy `TeselowanaSfera` (listing 1.4) oraz definicję jej metod (listing 1.5).

Listing 1.4. Deklaracja klasy sfery złożonej z płatów. Plik: *Aktor.h*.

```
class TeselowanaSfera : public TeselowanyAktor
{
private:
	float promień;
	unsigned int liczbaSekcjiNaRównoleżnikach, liczbaSekcjiNaPołudnikach;
	unsigned int TwórzTabliceWerteksów(CWerteks*& werteksy);
	unsigned int TwórzTabliceIndeksów(GLuint*& indeksy);
	int indeks(int i, int j);

public:
	TeselowanaSfera(
		GLuint atrybutPołożenie, GLuint atrybutNormalna, GLuint
atrybutWspółrzędneTeksturowania, GLuint atrybutKolor,
		float promień,
		unsigned int liczbaSekcjiNaRównoleżnikach = 20, unsigned int
liczbaSekcjiNaPołudnikach = 20)
		:TeselowanyAktor(), promień(promień),
liczbaSekcjiNaRównoleżnikach(liczbaSekcjiNaRównoleżnikach),
liczbaSekcjiNaPołudnikach(liczbaSekcjiNaPołudnikach)
	{
		Inicjuj(atrybutPołożenie, atrybutNormalna,
atrybutWspółrzędneTeksturowania, atrybutKolor);
	}
};
```

Listing 1.5. Definicje metod klasy TeselowanaSfera. Plik: *Aktor.cpp*.

```
unsigned int TeselowanaSfera::TwórzTabliceWerteksów(CWerteks*& werteksy)
{
	const unsigned int całkowitaLiczbaWerteksów =
		(liczbaSekcjiNaRównoleżnikach + 1)
		* (liczbaSekcjiNaPołudnikach + 1);
	werteksy = new CWerteks[całkowitaLiczbaWerteksów];

	const double przyrostKątaTheta = M_PI / liczbaSekcjiNaPołudnikach;
	const double przyrostKątaPhi =
		2 * M_PI / liczbaSekcjiNaRównoleżnikach;

	for (unsigned int i = 0; i <= liczbaSekcjiNaPołudnikach; i++)
	{
		double kątTheta = i * przyrostKątaTheta;
		float _wysokość = (float)(promień * cos(kątTheta));
		float _promień = (float)(promień * sin(kątTheta));

		for (unsigned int j = 0;
			j <= liczbaSekcjiNaRównoleżnikach;
			j++)
		{
			double kątPhi = j*przyrostKątaPhi;
			int indeks = j + i*(liczbaSekcjiNaRównoleżnikach + 1);

			werteksy[indeks].x = _promień*(float)cos(kątPhi);
			werteksy[indeks].y = _promień*(float)sin(kątPhi);
```

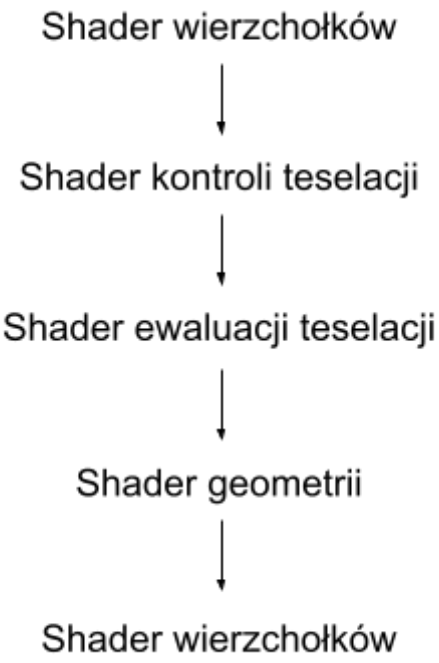
```
                    werteksy[indeks].z = _wysokość;

                    werteksy[indeks].r = i/(liczbaSekcjiNaPołudnikach-1);
                    werteksy[indeks].g = 0.5;
                    werteksy[indeks].b = 0.5;
                    werteksy[indeks].a = 1;
                    if (i != 0 && i != liczbaSekcjiNaPołudnikach)
                    {
                          werteksy[indeks].nx =
                                werteksy[indeks].x / _promień;
                          werteksy[indeks].ny =
                                werteksy[indeks].y / _promień;
                          werteksy[indeks].nz =
                                werteksy[indeks].z / _promień;
                    }
                    else
                    {
                          werteksy[indeks].nx = 0;
                          werteksy[indeks].ny = 0;
                          werteksy[indeks].nz =
                                werteksy[indeks].z
                                 / (float)fabs(werteksy[indeks].z);
                    }
                    werteksy[indeks].s = (float)(1 - kątPhi / 2 / M_PI);
                    werteksy[indeks].t = (float)(kątTheta / M_PI);
              }
        }

        return całkowitaLiczbaWerteksów;
}

unsigned int TeselowanaSfera::TwórzTabliceIndeksów(GLuint*& indeksy)
{
        const unsigned int liczbaIndeksówWPaśmie =
                          6 * liczbaSekcjiNaRównoleżnikach;
        const unsigned int całkowitaLiczbaIndeksów =
              liczbaIndeksówWPaśmie * liczbaSekcjiNaPołudnikach;
        indeksy = new GLuint[całkowitaLiczbaIndeksów];

        for (unsigned int i = 0; i < liczbaSekcjiNaPołudnikach; i++)
        {
              for (unsigned int j = 0; j < liczbaSekcjiNaRównoleżnikach; j++)
              {
                    int szeregowo = i*liczbaSekcjiNaRównoleżnikach+j;
                    indeksy[6 * szeregowo] =
                          i * (liczbaSekcjiNaRównoleżnikach+1) + j;
                    indeksy[6 * szeregowo + 1] =
                          i * (liczbaSekcjiNaRównoleżnikach+1) + j
                          + (liczbaSekcjiNaRównoleżnikach+1);
                    indeksy[6 * szeregowo + 2] =
                          i * (liczbaSekcjiNaRównoleżnikach+1) + j
                           + liczbaSekcjiNaRównoleżnikach + 1 + 1;
                    indeksy[6 * szeregowo + 3] =
                          i * (liczbaSekcjiNaRównoleżnikach+1) + j;
                    indeksy[6 * szeregowo + 4] =
                          i * (liczbaSekcjiNaRównoleżnikach+1) + j
                           + (liczbaSekcjiNaRównoleżnikach+1) + 1;
                    indeksy[6 * szeregowo + 5] =
                          i * (liczbaSekcjiNaRównoleżnikach+1) + j + 1;
              }
        }

        return całkowitaLiczbaIndeksów;
}
```

Wróćmy do pliku *OknoGL.cpp*, aby zastąpić oryginalną sferę przez tą utworzoną przed chwilą z płatów. W metodzie `PrzygotujAktorów`, zamiast tworzyć obiekt typu `SferaZBuforemIndeksów`, tworzymy obiekt typu `TeselowanaSfera`. Dodatkowo zmniejszamy do 5 liczbę sekcji, z których zrobiona jest sfera (listing 1.6). W efekcie generowana sfera będzie bardzo różniła się od rzeczywistego kształtu sfery (por. rys. 1.6). Ułatwi nam to jednak obserwowanie skutków teselacji, która w połączeniu z shaderem geometrii przywróci właściwy kształt sfery.

Listing 1.6. Podmiana typu bryły na scenie.

```
unsigned int COknoGL::PrzygotujAktorów()
{
	GLuint atrybutPołożenie = glGetAttribLocation(idProgramuShaderow,
"polozenie_in");
	if (atrybutPołożenie == (GLuint)-1) atrybutPołożenie = 0;

	GLuint atrybutNormalna = glGetAttribLocation(idProgramuShaderow,
"normalna_in");
	if (atrybutNormalna == (GLuint)-1) atrybutNormalna = 1;

	GLuint atrybutWspółrzędneTeksturowania =
glGetAttribLocation(idProgramuShaderow, "wspTekstur_in");
	if (atrybutWspółrzędneTeksturowania == (GLuint)-1)
atrybutWspółrzędneTeksturowania = 2;

	GLuint atrybutKolor = glGetAttribLocation(idProgramuShaderow,
"kolor_in");
	if (atrybutKolor == (GLuint)-1) atrybutKolor = 3;

	int liczbaAktorów = 1;
	aktorzy = new Aktor*[liczbaAktorów];

	//bryły
	// Utworzenie obiektu TeselowanaSfera.
	aktorzy[0] = new TeselowanaSfera(atrybutPołożenie, atrybutNormalna,
atrybutWspółrzędneTeksturowania, atrybutKolor, 0.75f, 5, 5);
	aktorzy[0]->MacierzŚwiata = Macierz4::ObrótY(90) *
Macierz4::ObrótX(90);
	aktorzy[0]->MateriałŚwiatłoOtoczenia = Wektor4(0.2f, 0.2f, 0.2f, 1);
	aktorzy[0]->MateriałŚwiatłoRozpraszane = Wektor4(1, 1, 1, 1);
	aktorzy[0]->MateriałŚwiatłoRozbłysku = Wektor4(1, 1, 1, 1);
	aktorzy[0]->MateriałWykładnikRozbłysku = 10;
	aktorzy[0]->IndeksTekstury = (teksturowanieWłączone) ?
indeksyTekstur[0] : -1;

	return liczbaAktorów;
}
```

Jeśli jednak uruchomimy teraz program, zobaczymy jedynie puste okno – konieczne są shadery teselacji, które poradzą sobie ze zmienionymi danymi wejściowymi. Shadery teselacji znajdują się w potoku renderowania pomiędzy shaderem wierzchołków a shaderem geometrii [7]. Kolejność wszystkich shaderów w potoku renderowania obecnych w OpenGL prezentuje rysunek 1.5.



Rysunek 1.5 Kolejność shaderów w potoku renderowania.

Stwórzmy wobec tego w projekcie cztery pliki shaderów: *Sfera.vert* (listing 1.7), *Sfera.tesc* (listing 1.8), *Sfera.tese* (listing 1.9) i *Sfera.frag* (listing 1.10). Są to kolejno shader wierzchołków, shader kontroli teselacji, shader ewaluacji teselacji i shader fragmentów. W shaderze werteksów poziomy teselacji ustawiliśmy na 1, co oznacza, że po teselacji otrzymamy dokładnie taką samą siatkę bryły, jaką przesyłamy do potoku graficznego (bez żadnych modyfikacji).

Listing 1.7. Kod shadera wierzchołków.

```
#version 460 core

layout (location = 0) in vec3 polozenie_in;
layout (location = 3) in vec4 kolor_in;

const mat4 macierzJednostkowa = mat4(1.0);

uniform mat4 macierzSwiata = macierzJednostkowa;
uniform mat4 macierzWidoku = macierzJednostkowa;
uniform mat4 macierzRzutowania = macierzJednostkowa;
uniform mat4 macierzNormalnych = macierzJednostkowa;

mat4 macierzMVP = macierzRzutowania * macierzWidoku * macierzSwiata;

out vec4 kolorVertToTesc;

void main()
{
	vec4 polozenie = vec4(polozenie_in, 1.0);
	gl_Position = macierzMVP * polozenie;

	kolorVertToTesc = kolor_in;
}
```

Listing 1.8. Kod shadera kontroli teselacji.

```
#version 460 core

layout (vertices = 3) out;

in vec4 kolorVertToTesc[];

out vec4 kolorTescToTese[];

void main()
{
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
    kolorTescToTese[gl_InvocationID] = kolorVertToTesc[gl_InvocationID];

    // Ustawienie poziomów teselacji zewnętrznej dla każdego boku trójkąta.
    int tessLevelOuter = 1;
    gl_TessLevelOuter[0] = tessLevelOuter;
    gl_TessLevelOuter[1] = tessLevelOuter;
    gl_TessLevelOuter[2] = tessLevelOuter;

    // Ustawienie poziomów teselacji wewnętrznej.
    int tessLevelInner = 1;
    gl_TessLevelInner[0] = tessLevelInner;
}
```

Listing 1.9. Kod shadera ewaluacji teselacji.

```
#version 460 core

layout (triangles, equal_spacing , ccw) in;

in vec4 kolorTescToTese[];

out vec4 kolorTeseToFrag;

void main()
{
    gl_Position = gl_TessCoord.x * gl_in[0].gl_Position
                + gl_TessCoord.y * gl_in[1].gl_Position
                + gl_TessCoord.z * gl_in[2].gl_Position;

    kolorTeseToFrag = gl_TessCoord.x * kolorTescToTese[0]
                    + gl_TessCoord.y * kolorTescToTese[1]
                    + gl_TessCoord.z * kolorTescToTese[2];
}
```

Listing 1.10. Kod shadera fragmentów.

```
#version 460 core

in vec4 kolorTeseToFrag;

out vec4 FragColor;

void main()
{
      FragColor = kolorTeseToFrag;
}
```

Po uzupełnieniu shaderów GLSL, potrzebujemy metody w kodzie C++, która umożliwi ich kompilację, gdyż obecna wersja metody `PrzygotujShadery` obsługuje jedynie shadery wierzchołków i fragmentów. Dodajmy zatem do klasy `COknoGL` jej przeciążoną wersję mogącą skompilować wszystkie pięć shaderów obecnego potoku graficznego. Zacznijmy zatem od dodania w klasie `COknoGL` (plik nagłówka *OknoGL.h*) deklaracji tej przeciążonej metody (listing 1.11), a następnie w pliku *OknoGL.cpp* dodajemy jej definicję (listing 1.12). Metoda ta jest dość długa, ale w znacznej mierze powiela jedynie to, co robi jej oryginalna wersja[2].

Listing 1.11. Deklaracja przeciążonej metody `PrzygotujShadery`.

```
class COknoGL : public COkno
{
private:
	HGLRC uchwytRC; //uchwyt kontekstu renderingu
	HDC uchwytDC; //uchwyt prywatnego kontekstu urządzenia GDI
	bool UstalFormatPikseli(HDC uchwytDC) const;
	bool InicjujWGL(HWND uchwytOkna);
	void UsuńWGL();
	void UmieśćInformacjeNaPaskuTytułu(HWND uchwytOkna);

	unsigned int idProgramuShaderow;
	static unsigned int KompilujShader(const char* nazwaPliku, unsigned int
typ, bool trybDebugowania = false);
	static unsigned int PrzygotujShadery(const char* vsNazwaPliku, const
char* fsNazwaPliku, bool trybDebugowania = false);
	static unsigned int PrzygotujShadery(
		const char* fileNameVertexShader,
		const char* fileNameTesselationControlShader,
		const char* fileNameTesselationEvaluationShader,
		const char* fileNameGeometryShader,
		const char* fileNameFragmentShader,
		bool trybDebugowania = false);
	void ustawCzas();
```

Listing 1.12. Definicja przeciążonej metody `PrzygotujShadery`.

```
unsigned int COknoGL::PrzygotujShadery(
	const char* fileNameVertexShader,
	const char* fileNameTesselationControlShader,
	const char* fileNameTesselationEvaluationShader,
	const char* fileNameGeometryShader,
	const char* fileNameFragmentShader,
	bool trybDebugowania)
{

	//Tworzenie obiektu programu
	GLuint idProgramu = glCreateProgram();

	// 1. Kompilacja shadera werteksów
	GLuint idShaderaWerteksów = NULL;
	if (fileNameVertexShader != nullptr)
	{

		idShaderaWerteksów = KompilujShader(
```

[2] *Zasadne byłoby użycie wzorca metody szablonowej, ale ponieważ to wiązałoby się ze zbyt dużymi zmianami w kodzie, które odciągałyby uwagę czytelnika od zasadniczego tematu, nie zdecydowałem się na takie rozwiązanie. Szczegóły metody kompilującej shadery są omówione w rozdziale 6 "Grafiki 3D czasu rzeczywistego. Nowoczesny OpenGL" [1].*

```
                    fileNameVertexShader,
                    GL_VERTEX_SHADER,
                    trybDebugowania);

            if (idShaderaWierteksów == NULL)
            {
                  PokażKomunikat(
                       "Kompilacja shadera werteksów nie powiodła się",
                       MB_ICONERROR);
                  return NULL;
            }
            else if (trybDebugowania) PokażKomunikat(
                     "Kompilacja shadera werteksów zakończyła się sukcesem",
                     MB_ICONINFORMATION);

            //Przylaczanie shadera
            glAttachShader(idProgramu, idShaderaWierteksów);
      }

      // 2. Kompilacja shadera kontroli teselacji.
      GLuint idTesselationControlShader = NULL;
      if (fileNameTesselationControlShader != nullptr)
      {
            idTesselationControlShader =
                 KompilujShader(fileNameTesselationControlShader,
                                GL_TESS_CONTROL_SHADER,
                                trybDebugowania);
            if (idTesselationControlShader == NULL)
            {
                  PokażKomunikat("Kompilacja shadera kontroli teselacji nie
powiodła się", MB_ICONERROR);
                  return NULL;
            }
            else if (trybDebugowania) PokażKomunikat("Kompilacja shadera
kontroli teselacji zakończyła się sukcesem", MB_ICONINFORMATION);

            glAttachShader(idProgramu, idTesselationControlShader);
//Przylaczanie shadera
      }

      // 3. Kompilacja shadera ewaluacji teselacji.
      GLuint idTesselationEvaluationShader = NULL;
      if (fileNameTesselationEvaluationShader != nullptr)
      {
            idTesselationEvaluationShader =
               KompilujShader(fileNameTesselationEvaluationShader,
                              GL_TESS_EVALUATION_SHADER, trybDebugowania);
            if (idTesselationEvaluationShader == NULL)
            {
               PokażKomunikat(
                 "Kompilacja shadera ewaluacji teselacji nie powiodła się",
                 MB_ICONERROR);
               return NULL;
            }
            else if (trybDebugowania)
               PokażKomunikat(
                 "Kompilacja shadera ewaluacji teselacji"
                 " zakończyła się sukcesem",
                 MB_ICONINFORMATION);

            glAttachShader(idProgramu, idTesselationEvaluationShader);
//Przylaczanie shadera
      }

      // 4. Kompilacja shadera geometrii
```

```
	GLuint idGeometryShader = NULL;
	if (fileNameGeometryShader != nullptr)
	{
		idGeometryShader = KompilujShader(fileNameGeometryShader,
GL_GEOMETRY_SHADER, trybDebugowania);
		if (idGeometryShader == NULL)
		{
			PokażKomunikat("Kompilacja shadera geometrii nie powiodła
się", MB_ICONERROR);
			return NULL;
		}
		else if (trybDebugowania) PokażKomunikat("Kompilacja shadera
geometrii zakończyła się sukcesem", MB_ICONINFORMATION);

		glAttachShader(idProgramu, idGeometryShader); //Przylaczanie
shadera
	}

	// 5. Kompilacja shadera fragmentów
	GLuint idShaderaFragmentów = NULL;
	if (fileNameFragmentShader != nullptr)
	{
		idShaderaFragmentów = KompilujShader(fileNameFragmentShader,
GL_FRAGMENT_SHADER, trybDebugowania);
		if (idShaderaFragmentów == NULL)
		{
			PokażKomunikat("Kompilacja shadera fragmentów nie powiodła
się", MB_ICONERROR);
			return NULL;
		}
		else if (trybDebugowania) PokażKomunikat("Kompilacja shadera
fragmentów zakończyła się sukcesem", MB_ICONINFORMATION);

		glAttachShader(idProgramu, idShaderaFragmentów); //Przylaczanie
shadera
	}

	// Linkowanie.
	glLinkProgram(idProgramu);

	// Weryfikacja linkowania.
	GLint czyPowodzenie;
	glGetProgramiv(idProgramu, GL_LINK_STATUS, &czyPowodzenie);
	if (!czyPowodzenie)
	{
		const int maxInfoLogSize = 2048;
		GLchar infoLog[maxInfoLogSize];
		glGetProgramInfoLog(idProgramu, maxInfoLogSize, NULL, infoLog);
		char komunikat[maxInfoLogSize + 64] = "Uwaga! Linkowanie programu
shaderów nie powiodło się:\n";
		strcat_s(komunikat, (char*)infoLog);
		PokażKomunikat(komunikat, MB_ICONERROR);
		return NULL;
	}
	else if (trybDebugowania)
		PokażKomunikat("Linkowanie programu shaderów powiodło się",
MB_ICONINFORMATION);

	// Walidacja programu.
	glValidateProgram(idProgramu);

	// Weryfikacja walidacji.
	glGetProgramiv(idProgramu, GL_VALIDATE_STATUS, &czyPowodzenie);
	if (!czyPowodzenie)
	{
```

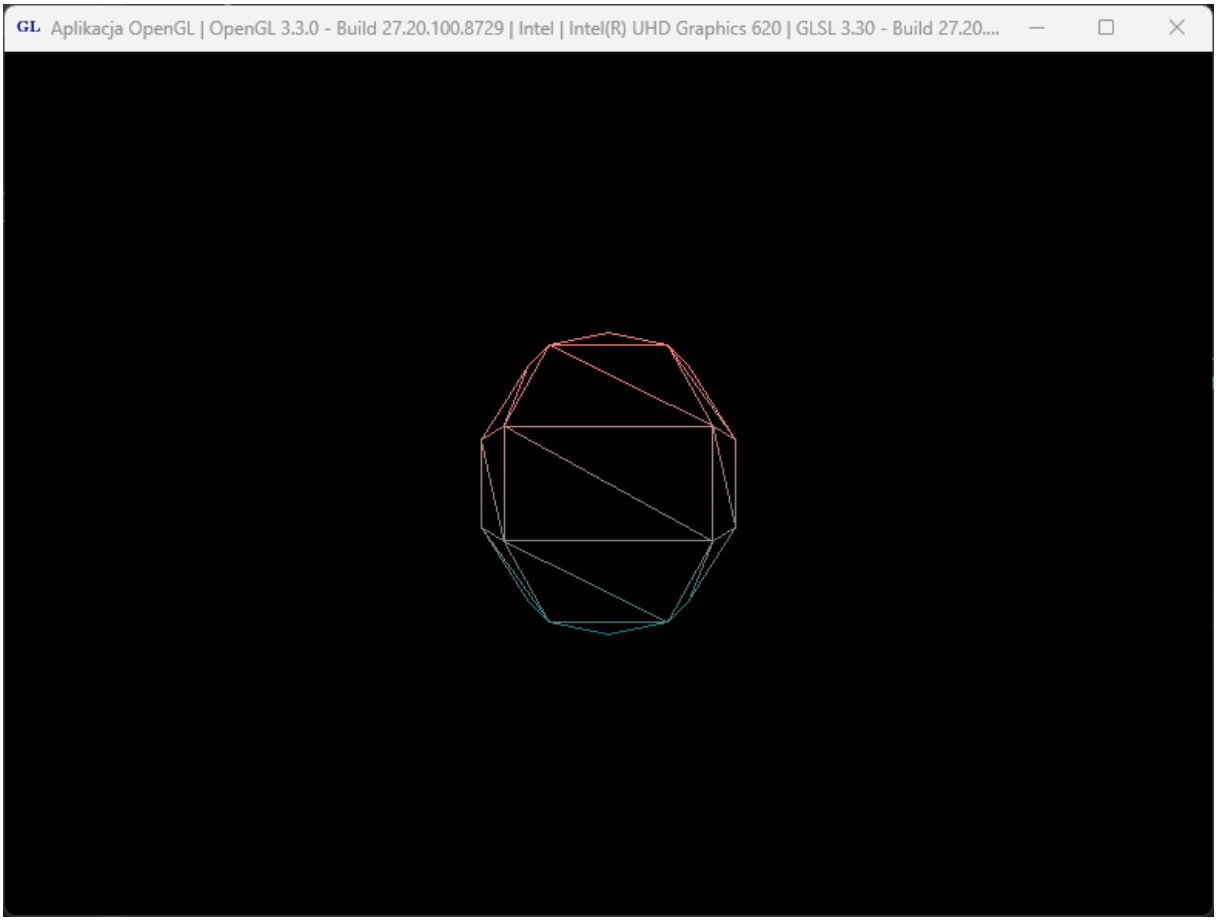
```
            const int maxInfoLogSize = 2048;
            GLchar infoLog[maxInfoLogSize];
            glGetProgramInfoLog(idProgramu, maxInfoLogSize, NULL, infoLog);
            char komunikat[maxInfoLogSize + 64] = "Uwaga! Walidacja programu
shaderów nie powiodła się:\n";
            strcat_s(komunikat, (char*)infoLog);
            PokażKomunikat(komunikat, MB_ICONERROR);
            return NULL;
      }
      else if (trybDebugowania)
            PokażKomunikat("Walidacja programu shaderów powiodła się",
MB_ICONINFORMATION);

      // Uzycie programu.
      glUseProgram(idProgramu);

      // Usuwanie niepotrzebnych obiektow shadera.
      if (idShaderaWerteksów != NULL)
            glDeleteShader(idShaderaWerteksów);
      if (idTesselationControlShader != NULL)
            glDeleteShader(idTesselationControlShader);
      if (idTesselationEvaluationShader != NULL)
            glDeleteShader(idTesselationEvaluationShader);
      if (idGeometryShader != NULL)
            glDeleteShader(idGeometryShader);
      if (idShaderaFragmentów != NULL)
            glDeleteShader(idShaderaFragmentów);

      return idProgramu;
}
```

W metodzie `WndProc` klasy `COknoGL` zastepujemy dotychczas wywoływaną metodę `PrzygotujShadery` jej nową wersją. Jako jej argumenty przekazujemy ścieżki do plików z naszymi shaderami w kolejności, w jakiej występują one w potoku graficznym. Jeśli jakiegoś shadera nie używamy (np. w tym wypadku shadera geometrii), to przekazujemy wartość `nullptr` (listing 1.13). Po tych zmianach możemy wreszcie uruchomić aplikację z nadzieją na zobaczenie bryły z ograniczoną liczbą wierzchołków (rysunek 1.6).

Listing 1.13. Wywołanie przeciążonej metody PrzygotujShadery.

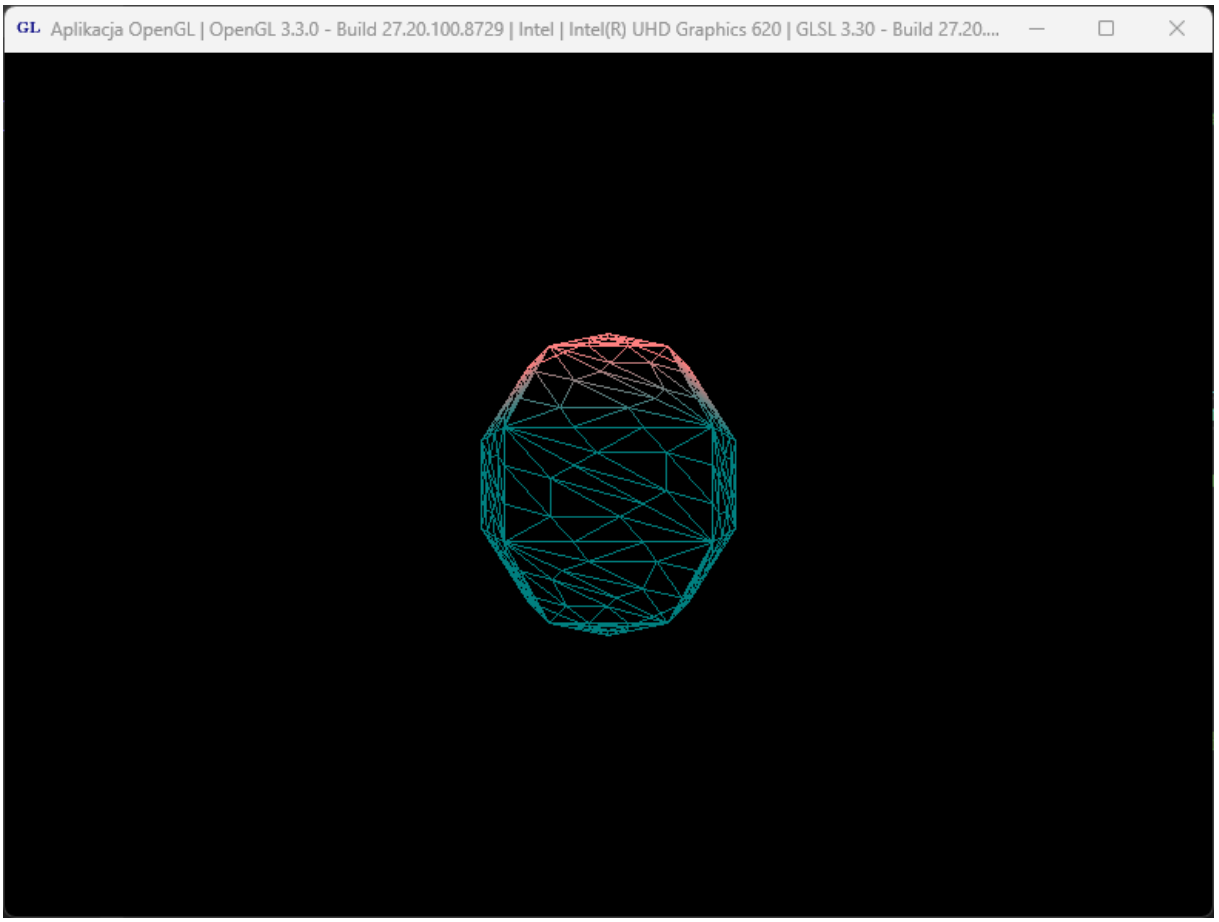
```
      switch (message)
      {
      case WM_CREATE: //Utworzenie okna
            //zmienna uchwytOkna nie jest jeszcze zainicjowana
            if (!InicjujWGL(hWnd))
            {
                  MessageBox(NULL, "Pobranie kontekstu renderowania nie
powiodło się", "Aplikacja OpenGL", MB_OK | MB_ICONERROR);
                  exit(EXIT_FAILURE);
            }

            // (13) Wywołanie przeciążonej metody PrzygotujShadery.
            //idProgramuShaderow = PrzygotujShadery("Basic.vsh", "Basic.fsh",
false);
            idProgramuShaderow = PrzygotujShadery(
                  "Sfera.vert",
                  "Sfera.tesc",
                  "Sfera.tese",
                  nullptr,
                  "Sfera.frag");

            if (idProgramuShaderow == NULL)
```

```
{
        MessageBox(NULL,
                   "Przygotowanie shaderów nie powiodło się",
                   "Aplikacja OpenGL",
                   MB_OK | MB_ICONERROR);
        exit(EXIT_FAILURE);
}
```



Rysunek 1.6 Bryła utworzona z użyciem shaderów teselacji.

Na rysunku 1.6 widać, że utworzyliśmy “sferę” zbudowaną dokładnie z takiej siatki, jaka powstaje w metodzie `TwórzTablicęWierzchołków`, czyli dość “kanciastą”, co pozwala pokazać korzyści płynące z użycia zastosowania teselacji. W tym celu ustawmy w shaderze kontroli teselacji zmienne `tessLevelOuter` i `tessLevelInner` równe 3. Wówczas otrzymamy efekt widoczny na rysunku 1.7. Jak widać siatka trójkątów zagęściła się. Nie ma to jednak wpływu na sam kształt bryły, ponieważ nowe wierzchołki znajdują się w tej samej płaszczyźnie, co wcześniejsze ściany utworzone z płatów. Chcąc uzyskać efekt wygładzenia sfery, będziemy musieli dodatkowo posłużyć się shaderem geometrii.



Rysunek 1.7 Bryła z zagęszczoną siatką trójkątów.

Wygładzenie sfery w shaderze geometrii

W przypadku sfery poprawimy jej kształt przesuwając wszystkie wierzchołki od środka sfery na odległość równą jej promieniowi. Do tego typu zadań najlepiej nadaje się shader geometrii [8]. Musimy jednak zmienić również shader wierzchołków (listing 1.14), ponieważ teraz nie będziemy mnożyć w nim położenia przez iloczyn macierzy *Model-View-Perspective*, ze względu na to, że na tym etapie potoku renderowania nie ma jeszcze wierzchołków tworzonych podczas teselacji. W związku z tym shader geometrii można obecnie traktować jak shader wierzchołków, który jest uruchamiany po etapie teselacji, a ściślej po shaderze ewaluacji teselacji [9].

Listing 1.14. Zmieniony kod shadera wierzchołków.

```
#version 460 core

layout (location = 0) in vec3 polozenie_in;

layout (location = 3) in vec4 kolor_in;

const mat4 macierzJednostkowa = mat4(1.0);

out vec4 kolorVertToTesc;
```

```
void main()
{
	vec4 polozenie = vec4(polozenie_in, 1.0);
	gl_Position = polozenie;

	kolorVertToTesc = kolor_in;
}
```

Dodajemy zatem do potoku renderowania shader geometrii. W tym celu tworzymy plik o nazwie *Sfera.geom* (listing 1.15) i dodajemy go do kompilowanych shaderów (listing 1.16). Następnie poprzez zmienne `uniform` przekazujemy do niego macierze, które wcześniej były wykorzystywane w shaderze wierzchołków. W taki sam sposób przekazujemy położenie środka sfery oraz jej promień (listing 1.17). Musimy także zmienić nazwy parametrów w shaderze kontroli teselacji (listing 1.18), shaderze ewaluacji teselacji (listing 1.19) oraz shaderze fragmentów (listingi 1.20).

Listing 1.15. Kod shadera geometrii.

```
#version 460 core

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

in vec4 kolorTeseToGeom[];

out vec4 kolorGeomToFrag;

uniform vec3 srodekSfery;
uniform float promienSfery;

const mat4 macierzJednostkowa = mat4(1.0);
uniform mat4 macierzSwiata = macierzJednostkowa;
uniform mat4 macierzWidoku = macierzJednostkowa;
uniform mat4 macierzRzutowania = macierzJednostkowa;
uniform mat4 macierzNormalnych = macierzJednostkowa;
mat4 macierzMVP = macierzRzutowania * macierzWidoku * macierzSwiata;

void main()
{

	for (int i = 0; i < 3; i++)
	{
		vec3 A = srodekSfery;
		vec4 B = gl_in[i].gl_Position;
		vec4 kierunekPrzesuniecia = B - vec4(A, 1);

		vec4 wektorPrzesuniecia =
		     normalize(kierunekPrzesuniecia) * promienSfery;

		gl_Position =
		     macierzMVP*(vec4(srodekSfery,1)+vec4(wektorPrzesuniecia));

		kolorGeomToFrag = kolorTeseToGeom[i];

		EmitVertex();
	}

	EndPrimitive();
}
```

Listing 1.16. W metodzie WndProc dodanie shadera geometrii do przygotowania.

```
      idProgramuShaderow = PrzygotujShadery(
            "Sfera.vert",
            "Sfera.tesc",
            "Sfera.tese",
            "Sfera.geom",
            "Sfera.frag");
```

Listing 1.17. Przekazanie zmiennych uniform na końcu metody UstawienieSceny.

```
void COknoGL::UstawienieSceny(bool rzutowanieIzometryczne)
{
...

GLint parametrSrodekSfery =
      glGetUniformLocation(idProgramuShaderow, "srodekSfery");
glUniform3f(parametrSrodekSfery, 0,0,0);

GLint parametrPromienSfery =
      glGetUniformLocation(idProgramuShaderow, "promienSfery");
glUniform1f(parametrPromienSfery, 0.75);
}
```

Listing 1.18. Zmieniony kod shadera kontroli teselacji.

```
#version 460 core

layout (vertices = 3) out;

in vec4 kolorVertToTesc[];

out vec4 kolorTescToTese[];

void main()
{
    gl_out[gl_InvocationID].gl_Position =
           gl_in[gl_InvocationID].gl_Position;
    kolorTescToTese[gl_InvocationID] =
           kolorVertToTesc[gl_InvocationID];

    int tessLevelOuter = 3;
    gl_TessLevelOuter[0] = tessLevelOuter;
    gl_TessLevelOuter[1] = tessLevelOuter;
    gl_TessLevelOuter[2] = tessLevelOuter;

    int tessLevelInner = 3;
    gl_TessLevelInner[0] = tessLevelInner;
}
```

Listing 1.19. Zmieniony kod shadera ewaluacji teselacji.

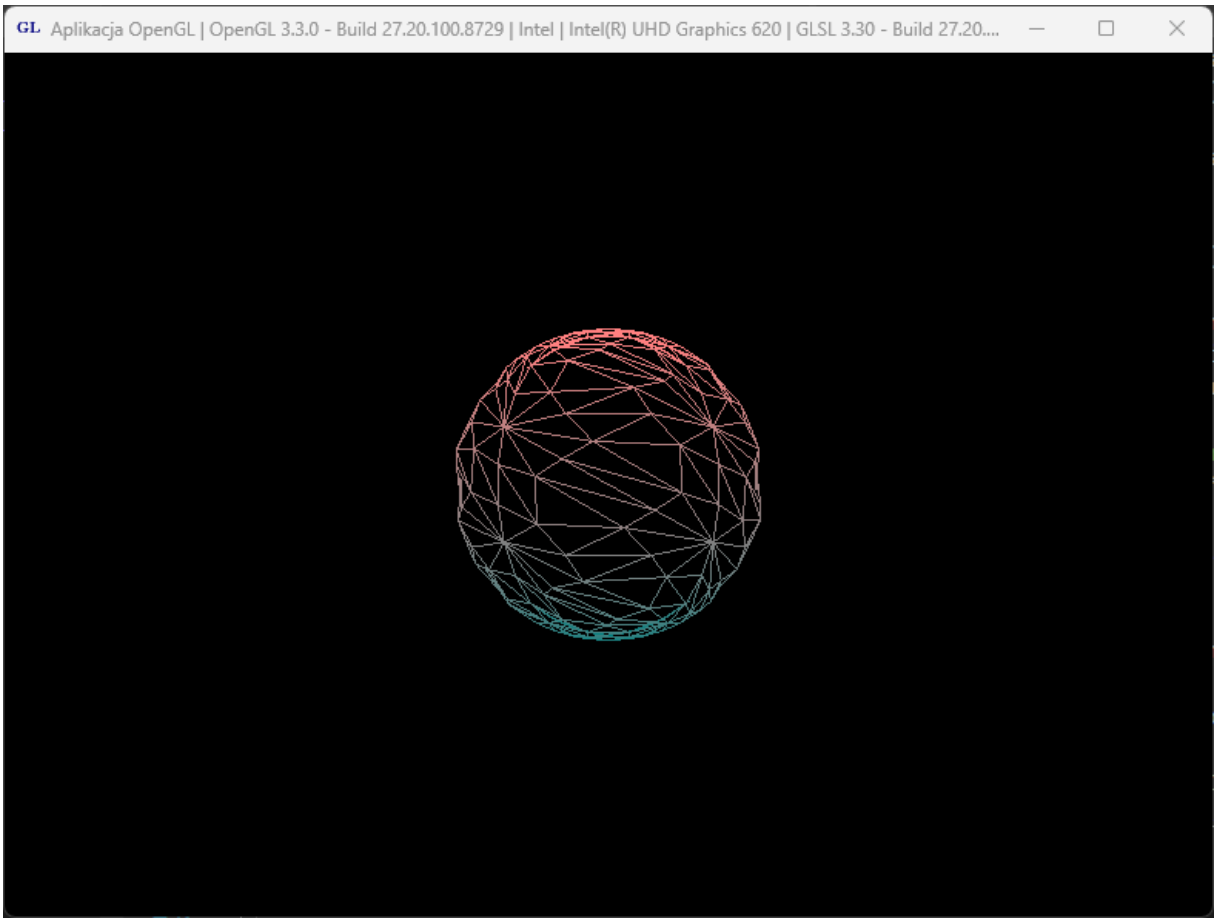
```
#version 460 core

layout (triangles, equal_spacing , ccw) in; // Ustawienie: typu prymitywu,
sposobu podziału krawędzi, nawijania.

in vec4 kolorTescToTese[];

out vec4 kolorTeseToGeom;
```

```
void main()
{
	gl_Position = gl_TessCoord.x * gl_in[0].gl_Position
		+ gl_TessCoord.y * gl_in[1].gl_Position
		+ gl_TessCoord.z * gl_in[2].gl_Position; // Wyjściowa pozycja
wierzchołka jest sumą iloczynów współrzędnych barycentrycznych i wejściowych
pozycji wierzchołka.

	kolorTeseToGeom = gl_TessCoord.x * kolorTescToTese[0]
			+ gl_TessCoord.y * kolorTescToTese[1]
			+ gl_TessCoord.z * kolorTescToTese[2];
}
```

Listing 1.20. Zmieniony kod shadera fragmentów.

```
#version 460 core

in vec4 kolorGeomToFrag;

out vec4 FragColor;

void main()
{
	FragColor = kolorGeomToFrag;
}
```

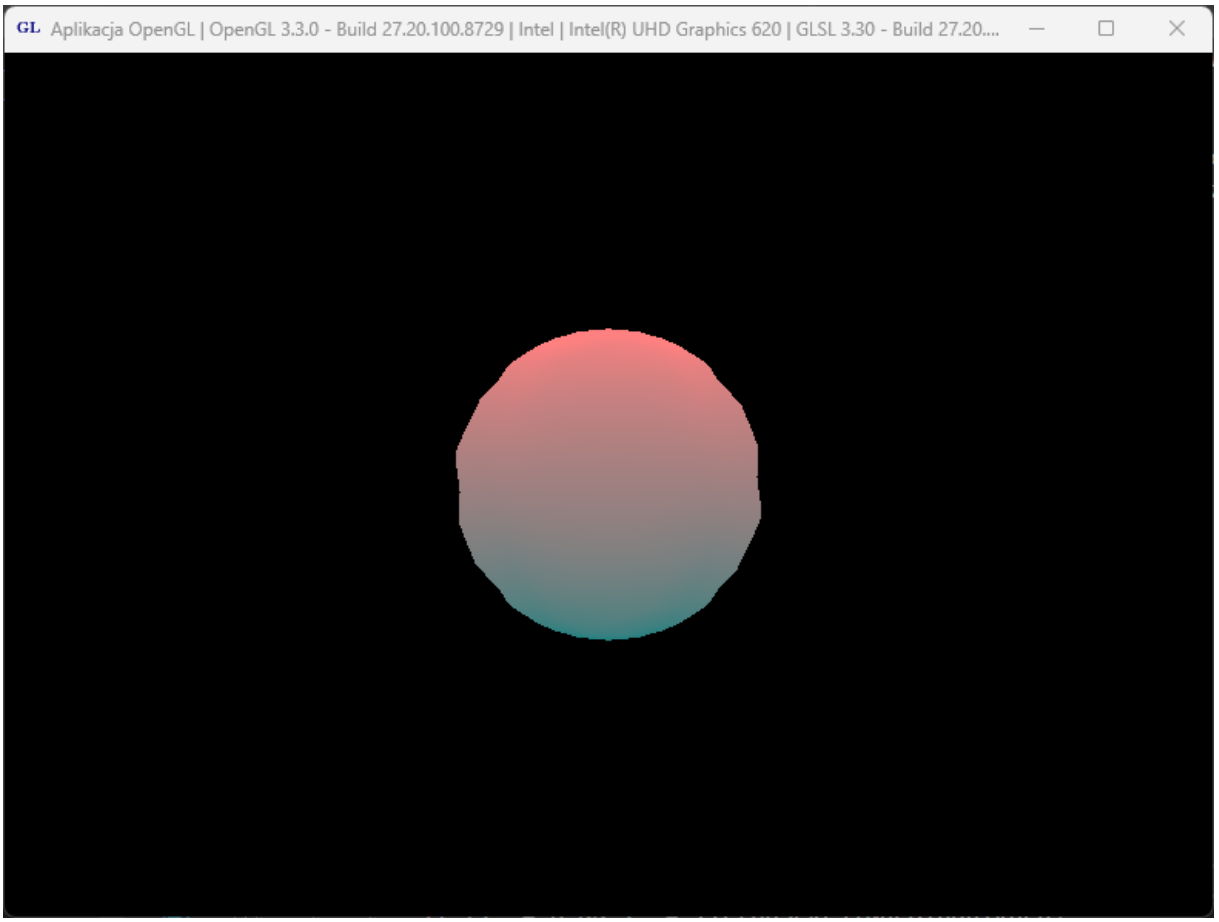
Gotowe! Teraz po uruchomieniu programu powinniśmy zobaczyć efekt widoczny na rysunku 1.8. Siatka jest zagęszczona, ale również wygładzona. Przywróćmy jeszcze tryb renderowania z wypełnieniem ścian (listing 1.21), co da finalny efekt widoczny na rysunku 1.9.



Rysunek 1.8 Wygładzona sfera.

Listing 1.21. Ostatnie linie metody UstawienieSceny.

```
void COknoGL::UstawienieSceny(bool rzutowanieIzometryczne)
{
	...

	glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

	GLint parametrSrodekSfery =
		glGetUniformLocation(idProgramuShaderow, "srodekSfery");
	glUniform3f(parametrSrodekSfery, 0,0,0);

	GLint parametrPromienSfery =
		glGetUniformLocation(idProgramuShaderow, "promienSfery");
	glUniform1f(parametrPromienSfery, 0.75);
}
```



Rysunek 1.9 Finalny efekt teselacji.

Tekstura

Jeśli chcielibyśmy przywrócić teksturę Ziemi, która była nałożona na sferę w projekcie startowym, musimy dokonać dodatkowych zmian w shaderze geometrii (listing 1.22) oraz shaderze fragmentów (listing 1.23). W shaderze geometrii obliczamy współrzędne teksturowania dla wszystkich wierzchołków, które są obecne po wykonaniu teselacji i przekazujemy je do shadera fragmentów. W shaderze fragmentów za pomocą funkcji `texture` pobieramy z tekstury kolor i przypisujemy do fragmentu [10]. Efekt możemy zobaczyć na rysunku 1.10. W następnym rozdziale nie będziemy potrzebować tekstury, zatem po obejrzeniu uzyskanego efektu, możemy wycofać właśnie wprowadzone zmiany.

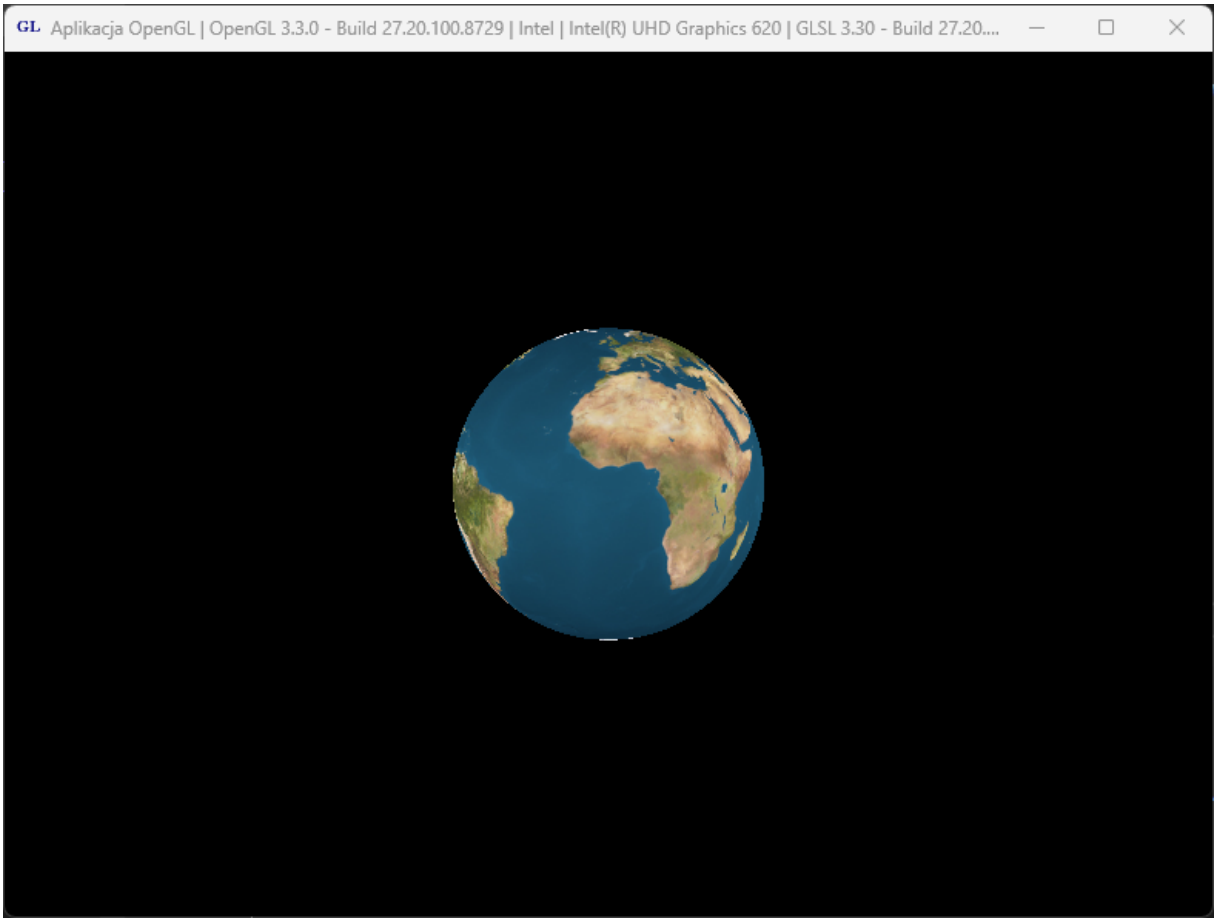
Listing 1.22. Modyfikacje dokonane w shaderze geometrii.

```
...
out vec2 wspTeksturGeomToFrag;
const float PI = 3.14159265358979323846;

void main()
{
    for (int i = 0; i < 3; i++)
    {
       vec3 A = srodekSfery;
       vec4 B = gl_in[i].gl_Position;
       vec4 kierunekPrzesuniecia = B - vec4(A, 1);

       vec4 wektorPrzesuniecia =
            normalize(kierunekPrzesuniecia) * promienSfery;
```

```
        float phi = atan(B.y, B.x);
        if (phi < 0)
              phi = 2 * PI + phi;
        float theta = acos(B.z / length(B.xyz));
        wspTeksturGeomToFrag.s = (1.0 - phi / 2.0 / PI);
        wspTeksturGeomToFrag.t = (theta / PI);

        gl_Position =
             macierzMVP * (vec4(srodekSfery, 1) + vec4(wektorPrzesuniecia));

        kolorGeomToFrag = kolorTeseToGeom[i];

        EmitVertex();
    }
    EndPrimitive();
}
```

Listing 1.23. Modyfikacje dokonane w shaderze fragmentów.

```
...
in vec2 wspTeksturGeomToFrag;
uniform sampler2D ProbnikTekstury0;

void main()
{
    FragColor = texture(ProbnikTekstury0, wspTeksturGeomToFrag);
}
```



Rysunek 1.10 Teselowana sfera z teksturą.

Rozdział 2.
Cieniowanie Fresnela i mapowanie środowiska

W tym rozdziale zajmiemy się implementacją cieniowania Fresnela. Następnie utworzymy skybox, co pozwoli nam wprowadzić mapowanie środowiska. Poprawi to znacznie wygląd całej sceny. Na koniec dodamy efekt dyspersji chromatycznej, czyli rozszczepienia światła. [11]

Teoria

Cieniowanie Fresnela to technika zapewniająca dokładniejsze oddanie zjawiska równoczesnego odbicia i załamywania się światła na różnych powierzchniach oraz zmiany koloru widzianego światła w zależności od kąta patrzenia i kąta padania światła na powierzchnię obiektu. [12]. Zjawisko to występuje — w różnym stopniu — na wszystkich obiektach. Można zaobserwować ten efekt patrząc na szybę. Jeżeli patrzymy na nią tak, że promienie światła dochodzące do oczu przechodzą przez szybę prostopadle do jej powierzchni, widzimy światło, które przez nie przenika. Natomiast jeżeli zmieniamy położenie oczu tak, że kąt promieni i powierzchni szyby zmniejsza się, w coraz większym stopniu widzimy światło, które jest odbijane przez szybę — szyba zaczyna przypominać lustro. To oznacza, że na granicy dwóch ośrodków część promieni światła, których kąt padania na obiekt jest niewielki, jest odbijana, i właśnie efekty związane z tym zjawiskiem są symulowane w cieniowaniu Fresnela. Dobrym przybliżeniem równań Fresnela wystarczającym do zastosowań w grafice komputerowej jest wzór 2.1.

$$współczynnik\ odbicia = max(0, min(1, błąd, skala * (1 + I \bullet N)^{potęga}$$

Wzór 2.1 Przybliżenie równania Fresnela

Implementacja cieniowania Fresnela

Kontynuujemy rozwijanie kodu z poprzedniego rozdziału. Zmiany wprowadzamy zarówno w shaderach, jak i w kodzie C++, choć w drugim przypadku są niewielkie. We wszystkich etapach potoku graficznego dodajemy zmienne wejściowe i wyjściowe (położenia, normalne i kolory werteksów) oraz przekazujemy ich wartości (listingi 2.1-2.5). Natomiast najważniejszy nowy element umieszczamy w shaderze fragmentów — jest to funkcja `CieniowanieFresnela` (listing 2.5).

Listing 2.1. Zmieniony shader werteksów

```
#version 460 core

layout (location = 0) in vec3 polozenie_in;
layout (location=1) in vec3 normalna_in;
```

```
layout (location = 3) in vec4 kolor_in;

const mat4 macierzJednostkowa = mat4(1.0);

uniform mat4 macierzSwiata = macierzJednostkowa;
uniform mat4 macierzNormalnych = macierzJednostkowa;

out vec4 kolorVertToTesc;

out vec3 polozenie_scena_vert;
out vec3 normalna_scena_vert;

void main()
{
	vec4 polozenie = vec4(polozenie_in, 1.0);
	gl_Position = polozenie;

	kolorVertToTesc = kolor_in;

	polozenie_scena_vert = mat3(macierzSwiata)*polozenie_in;
	normalna_scena_vert = mat3(macierzNormalnych)*normalna_in;
}
```

Listing 2.2. Zmieniony shader kontroli teselacji.

```
#version 460 core

layout (vertices = 3) out;

in vec4 kolorVertToTesc[];
in vec3 polozenie_scena_vert[];
in vec3 normalna_scena_vert[];

out vec4 kolorTescToTese[];
out vec3 polozenie_scena_tesc[];
out vec3 normalna_scena_tesc[];

void main()
{
    gl_out[gl_InvocationID].gl_Position =
            gl_in[gl_InvocationID].gl_Position;

    kolorTescToTese[gl_InvocationID] =
            kolorVertToTesc[gl_InvocationID];

    int tessLevelOuter = 6;
    gl_TessLevelOuter[0] = tessLevelOuter;
    gl_TessLevelOuter[1] = tessLevelOuter;
    gl_TessLevelOuter[2] = tessLevelOuter;

    int tessLevelInner = 6;
    gl_TessLevelInner[0] = tessLevelInner;

    polozenie_scena_tesc[gl_InvocationID] =
            polozenie_scena_vert[gl_InvocationID];

    normalna_scena_tesc[gl_InvocationID] =
            normalna_scena_vert[gl_InvocationID];
}
```

Listing 2.3. Zmieniony shader ewaluacji teselacji.

```
#version 460 core

layout (triangles, equal_spacing , ccw) in;

in vec4 kolorTescToTese[];
in vec3 polozenie_scena_tesc[];
in vec3 normalna_scena_tesc[];

out vec4 kolorTeseToGeom;
out vec3 polozenie_scena_tese;
out vec3 normalna_scena_tese;

void main()
{
    gl_Position = gl_TessCoord.x * gl_in[0].gl_Position
                + gl_TessCoord.y * gl_in[1].gl_Position
                + gl_TessCoord.z * gl_in[2].gl_Position;

    kolorTeseToGeom = gl_TessCoord.x * kolorTescToTese[0]
                    + gl_TessCoord.y * kolorTescToTese[1]
                    + gl_TessCoord.z * kolorTescToTese[2];

    polozenie_scena_tese = gl_TessCoord.x * polozenie_scena_tesc[0]
                    + gl_TessCoord.y * polozenie_scena_tesc[1]
                    + gl_TessCoord.z * polozenie_scena_tesc[2];
      normalna_scena_tese = gl_TessCoord.x * normalna_scena_tesc[0]
                    + gl_TessCoord.y * normalna_scena_tesc[1]
                    + gl_TessCoord.z * normalna_scena_tesc[2];
}
```

Listing 2.4. Zmieniony shader geometrii.

```
#version 460 core

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

in vec4 kolorTeseToGeom[];
in vec3 polozenie_scena_tese[];
in vec3 normalna_scena_tese[];

out vec4 kolorGeomToFrag;
out vec3 polozenie_scena_geom;
out vec3 normalna_scena_geom;

uniform vec3 srodekSfery;
uniform float promienSfery;

const mat4 macierzJednostkowa = mat4(1.0);
uniform mat4 macierzSwiata = macierzJednostkowa;
uniform mat4 macierzWidoku = macierzJednostkowa;
uniform mat4 macierzRzutowania = macierzJednostkowa;
uniform mat4 macierzNormalnych = macierzJednostkowa;
mat4 macierzMVP = macierzRzutowania * macierzWidoku * macierzSwiata;

void main()
{
      for (int i = 0; i < 3; i++)
      {
            vec3 A = srodekSfery;
            vec4 B = gl_in[i].gl_Position;
```

```
            vec4 kierunekPrzesuniecia = B - vec4(A, 1);

            vec4 wektorPrzesuniecia =
                 normalize(kierunekPrzesuniecia) * promienSfery;

            gl_Position =
                 macierzMVP*(vec4(srodekSfery,1)+vec4(wektorPrzesuniecia));

            kolorGeomToFrag = kolorTeseToGeom[i];

            polozenie_scena_geom = polozenie_scena_tese[i];
            normalna_scena_geom = polozenie_scena_tese[i];

            EmitVertex();
      }

      EndPrimitive();
}
```

Listing 2.5. Zmieniony shader fragmentów.

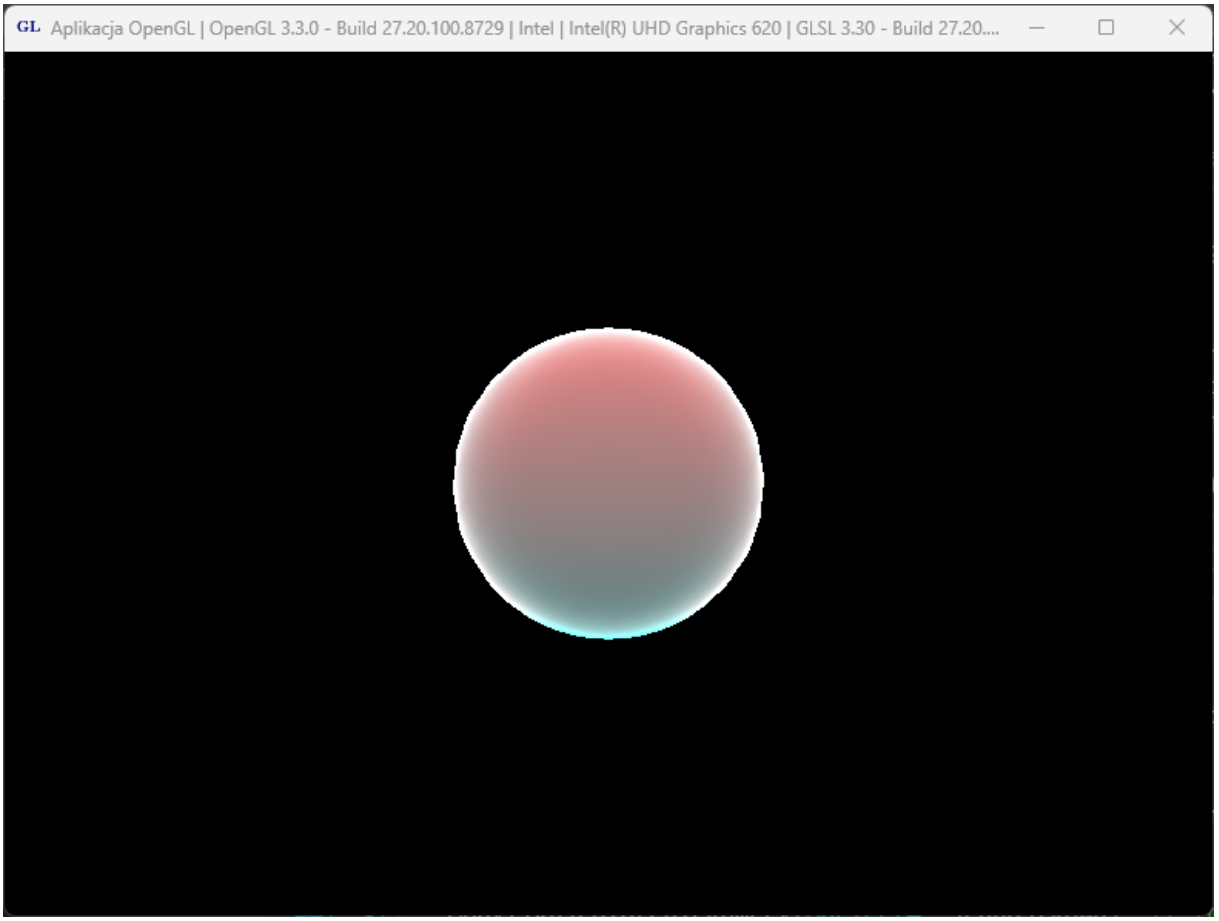
```
#version 460 core

in vec4 kolorGeomToFrag;
in vec3 polozenie_scena_geom;
in vec3 normalna_scena_geom;

uniform vec3 PolozenieKamery = vec3(0,0,1);

out vec4 FragColor;

vec4 CieniowanieFresnela(vec4 argkolor)
{
      vec3 I = normalize(polozenie_scena_geom - PolozenieKamery);
      vec3 N = normalize(normalna_scena_geom);

      float blad = 0;
      float skala = 2.2;
      float potega = 3;

      float wspolczynnikOdbicia =
               max(0, min(1, blad + skala * pow(1.0 + dot(I, N), potega)));

      vec4 kolorEfektu = clamp(argkolor*3, 0.0, 1.0);//

      vec4 kolorWynikowy = mix(argkolor, kolorEfektu, wspolczynnikOdbicia);

    return kolorWynikowy;
}

void main()
{
      FragColor = kolorGeomToFrag;
      FragColor = CieniowanieFresnela(FragColor);
}
```

Dodatkowo musimy wprowadzić niewielką zmianę w pliku *OknoGL.cpp*. W shaderze fragmentów wykorzystujemy zmienną `PolozenieKamery`, która z poziomu kodu C++ jest ustawiana w metodzie `ModyfikujPolozenieKamery`, wywoływanej w razie poruszania kamerą. Prowadzi to do sytuacji, w której zmienna `PolozenieKamery` nie jest na początku

zainicjalizowana poprawną wartością, co staje się widoczne w przypadku uruchomienia kodu odpowiedzialnego za implementację efektu Fresnela. Dodajemy zatem na końcu metody `UstawienieSceny` wywołanie metody `ModyfikujPolozenieKamery` (listing 2.6). Następnie możemy uruchomić projekt. Rysunek 2.1 przedstawia efekt końcowy, czyli białą otoczkę widoczną na obrzeżach sfery.

Listing 2.6. Kod dodany na koniec metody `UstawienieSceny`.

```
// Początkowe ustawienie zmiennej uniform PolozenieKamery w shaderze.
ModyfikujPołożenieKamery(Macierz4::Jednostkowa);
```



Rysunek 2.1 Sfera z cieniowaniem Fresnela.

Skybox

Efekt Fresnela dotyczy odbijania się światła od powierzchni renderowanego obiektu. W sposób oczywisty światło to zależy od źródeł światła w otoczeniu obiektu, w tym również innych obiektów odbijających światło. Otoczenie takie można symulować za pomocą tzw. skyboksa i odpowiedniego mapowania jego tekstury na renderowany obiekt. Sam skybox wykorzystywany jest do poprawienia wyglądu sceny poprzez pozorne zwiększenie jego rozmiaru [13] [14].

Skybox będzie wymagał oddzielnego zestawu shaderów, ze względu na to, że nie będzie on podlegał teselacji i przesuwania werteksów w shaderze geometrii. Zacznijmy zatem od utworzenia kompletu nowych shaderów. Będą to dwa pliki: *Skybox.vert* i *Skybox.frag*, które będą zawierały dwa proste programy cieniujące widoczne na listingach 2.7 i 2.8.

Listing 2.7. Shader *Skybox.vert*.

```
#version 330 core

layout (location = 0) in vec3 polozenie_in;

const mat4 macierzJednostkowa = mat4(1.0);
uniform mat4 macierzSwiata = macierzJednostkowa;
uniform mat4 macierzWidoku = macierzJednostkowa;
uniform mat4 macierzRzutowania = macierzJednostkowa;
mat4 macierzMVP = macierzRzutowania*macierzWidoku*macierzSwiata;

uniform mat4 macierzNormalnych = macierzJednostkowa;

out vec3 texCoords;

void main()
{
    texCoords = polozenie_in;

    vec4 pos = macierzMVP * vec4(polozenie_in, 1.0);
    gl_Position = vec4(pos.x, pos.y, pos.z, pos.w);
}
```

Listing 2.8. Shader *Skybox.frag*.

```
#version 330 core

uniform samplerCube ProbnikTeksturySkybox;

in vec3 texCoords;

out vec4 FragColor;

void main()
{
    FragColor = texture(ProbnikTeksturySkybox, texCoords);
}
```

Następnie dodajemy kod C++ wczytujący i ustawiający drugi zestaw shaderów. W pliku *OknoGL.h* potrzebujemy teraz nie jednej, a dwóch zmiennych przechowujących identyfikatory programów shaderów, zatem obok pola `idProgramuShaderow` definiujemy także drugie o nazwie `idProgramuShaderowSkybox`, przeznaczone dla shaderów skyboksa. Dodajemy również metodę włączającą dany zestaw shaderów oraz metodę wczytującą tekstury skyboksa.

Listing 2.9. Deklaracja nowych elementów klasy COknoGL.

```
unsigned int idProgramuShaderow;
unsigned int idProgramuShaderowSkybox;
void UzyjProgramuShaderow(
    unsigned int programShaderow,
    bool rzutowanieIzometryczne = false);
```

```
...
GLuint* indeksyTekstur;
GLuint indeksTeksturySkybox;
unsigned int liczbaTekstur;
virtual void PrzygotujTekstury(
    unsigned int liczbaTekstur,
    char** nazwyPlikówTekstur);
virtual void PrzygotujTeksturySkybox();
```

Zanim przejdziemy do modyfikacji pliku *OknoGL.cpp*, w sekcji publicznej klasy `Aktor` dodajmy nowe pole (listing 2.10). Modyfikacja klasy bazowej jest konieczna, ponieważ chcemy móc przypisać każdemy aktorowi na scenie osobny zestaw shaderów, który ma wykorzystywać do renderowania. Następnie tworzymy również klasę `Skybox` dziedziczącą z klasy `Aktor`. W pliku *Aktor.h* tworzymy deklarację nowej klasy (listing 2.11), natomiast w pliku *Aktor.cpp* definiujemy jej metody (listing 2.12).

Listing 2.10. Nowe pole w `Aktor`.

```
public:
    int ProgramShaderowId;
```

Listing 2.11. Deklaracja klasy `Skybox`.

```
class Skybox : public Aktor
{
private:
    float długośćKrawędzi;
    unsigned int TwórzTablicęWerteksów(CWerteks*& werteksy);

public:
    void Rysuj();

    Skybox(
       GLuint atrybutPołożenie,
       GLuint atrybutNormalna,
       GLuint atrybutWspółrzędneTeksturowania,
       GLuint atrybutKolor,
       float długośćKrawędzi);

    static Skybox* StwórzSkybox(
       GLuint atrybutPołożenie,
       float długośćKrawędzi)
    {
       return new Skybox(atrybutPołożenie, -1, -1, -1, długośćKrawędzi);
    }
};
```

Listing 2.12. Definicja metod klasy `Skybox`.

```
Skybox::Skybox(
    GLuint atrybutPołożenie, GLuint atrybutNormalna,
    GLuint atrybutWspółrzędneTeksturowania, GLuint atrybutKolor,
    float długośćKrawędzi)
:Aktor(), długośćKrawędzi(długośćKrawędzi)
{
    Inicjuj(atrybutPołożenie, atrybutNormalna,
       atrybutWspółrzędneTeksturowania, atrybutKolor);
}
```

```
unsigned int Skybox::TwórzTablicęWerteksów(CWerteks*& werteksy)
{
    const float x0 = długośćKrawędzi / 2.0f;
    const float y0 = długośćKrawędzi / 2.0f;
    const float z0 = długośćKrawędzi / 2.0f;

    werteksy = new CWerteks[24];

    float r = 1.0f; float g = 1.0f; float b = 1.0f;
    werteksy[0] = CWerteks(x0, -y0, -z0, 0, 0, -1, 0, 0, r, g, b);
    werteksy[1] = CWerteks(-x0, -y0, -z0, 0, 0, -1, 1, 0, r, g, b);
    werteksy[2] = CWerteks(x0, y0, -z0, 0, 0, -1, 0, 1, r, g, b);
    werteksy[3] = CWerteks(-x0, y0, -z0, 0, 0, -1, 1, 1, r, g, b);

    werteksy[4] = CWerteks(-x0, -y0, z0, 0, 0, 1, 0, 0, r, g, b);
    werteksy[5] = CWerteks(x0, -y0, z0, 0, 0, 1, 1, 0, r, g, b);
    werteksy[6] = CWerteks(-x0, y0, z0, 0, 0, 1, 0, 1, r, g, b);
    werteksy[7] = CWerteks(x0, y0, z0, 0, 0, 1, 1, 1, r, g, b);

    werteksy[8] = CWerteks(x0, -y0, z0, 1, 0, 0, 0, 0, r, g, b);
    werteksy[9] = CWerteks(x0, -y0, -z0, 1, 0, 0, 1, 0, r, g, b);
    werteksy[10] = CWerteks(x0, y0, z0, 1, 0, 0, 0, 1, r, g, b);
    werteksy[11] = CWerteks(x0, y0, -z0, 1, 0, 0, 1, 1, r, g, b);

    werteksy[12] = CWerteks(-x0, -y0, -z0, -1, 0, 0, 0, 0, r, g, b);
    werteksy[13] = CWerteks(-x0, -y0, z0, -1, 0, 0, 1, 0, r, g, b);
    werteksy[14] = CWerteks(-x0, y0, -z0, -1, 0, 0, 0, 1, r, g, b);
    werteksy[15] = CWerteks(-x0, y0, z0, -1, 0, 0, 1, 1, r, g, b);

    werteksy[16] = CWerteks(-x0, y0, z0, 0, 1, 0, 0, 0, r, g, b);
    werteksy[17] = CWerteks(x0, y0, z0, 0, 1, 0, 1, 0, r, g, b);
    werteksy[18] = CWerteks(-x0, y0, -z0, 0, 1, 0, 0, 1, r, g, b);
    werteksy[19] = CWerteks(x0, y0, -z0, 0, 1, 0, 1, 1, r, g, b);

    werteksy[20] = CWerteks(-x0, -y0, -z0, 0, -1, 0, 0, 0, r, g, b);
    werteksy[21] = CWerteks(x0, -y0, -z0, 0, -1, 0, 1, 0, r, g, b);
    werteksy[22] = CWerteks(-x0, -y0, z0, 0, -1, 0, 0, 0, r, g, b);
    werteksy[23] = CWerteks(x0, -y0, z0, 0, -1, 0, 1, 0, r, g, b);

    return 24;
}

void Skybox::Rysuj()
{
    glFrontFace(GL_CW);
    glBindVertexArray(vao);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    for (int i = 0; i < 6; ++i)
    {
       glDrawArrays(GL_TRIANGLE_STRIP, i * 4, 4);
    }
    glFrontFace(GL_CCW);
}
```

Teraz przechodzimy do pliku *OpenGL.cpp*, w którym zmiany rozpoczynamy od zdefiniowania dwóch nowych metod. Na listingu 2.13 widzimy metodę, która wczytuje sześć tekstur potrzebnych do pokrycia ścian skyboksa i ustawia ich parametry. Należy pamiętać, aby w głównym katalogu projektu znajdowały się pliki teksturami (są już w szablonie użytym w poprzednim rozdziale; chodzi o pliki o nazwach: *skybox-back.bmp*, *skybox-down.bmp*, *skybox-front.bmp*, *skybox-left.bmp*, *skybox-right.bmp*, *skybox-up.bmp*). Następnie przechodzimy do implementacji metody odpowiedzialnej za przełączanie zestawu shaderów (listing 2.14).

Listing 2.13. Przygotowanie tekstur skyboksa.

```
void COknoGL::PrzygotujTeksturySkybox()
{
    glGenTextures(1, &indeksTeksturySkybox);
    glBindTexture(GL_TEXTURE_CUBE_MAP, indeksTeksturySkybox);

    glTexParameteri(GL_TEXTURE_CUBE_MAP,
       GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,
       GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,
       GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,
       GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP,
       GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);

    char* nazwyTekstur[] =
    {
       "skybox-right.bmp", "skybox-left.bmp", "skybox-up.bmp",
       "skybox-down.bmp", "skybox-front.bmp", "skybox-back.bmp"
    };

    int szerokosc,
    int wysokosc;
    unsigned long* tekstura;

    for(unsigned int i = 0; i < 6; i++)
    {
       tekstura = WczytajObrazZPlikuBitmap(
             uchwytOkna, nazwyTekstur[i],
             szerokosc, wysokosc, false, 255);
       glTexImage2D(
             GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
             0, GL_RGBA, szerokosc, wysokosc, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, tekstura);
    }
}
```

Listing 2.14. Definicja metody ustawiającej potrzebny zestaw shaderów.

```
void COknoGL::UzyjProgramuShaderow(
    unsigned int programShaderow,
    bool rzutowanieIzometryczne)
{
    glUseProgram(programShaderow);

    //Macierze
    GLint parametrMacierzŚwiata =
       glGetUniformLocation(programShaderow, "macierzSwiata");
    macierzŚwiata.ZwiążZIdentyfikatorem(
       parametrMacierzŚwiata, true);

    GLint parametrMacierzWidoku =
       glGetUniformLocation(programShaderow, "macierzWidoku");
    macierzWidoku.ZwiążZIdentyfikatorem(
       parametrMacierzWidoku, true);

    GLint parametrMacierzRzutowania =
       glGetUniformLocation(programShaderow, "macierzRzutowania");
    macierzRzutowania.ZwiążZIdentyfikatorem(
```

```
        parametrMacierzRzutowania, false);
    macierzRzutowania.PrześlijWartość();

    GLint parametrMacierzNormalnych =
        glGetUniformLocation(programShaderow, "macierzNormalnych");
    macierzNormalnych.ZwiążZIdentyfikatorem(
        parametrMacierzNormalnych, true);
}
```

Teraz możemy przejść do modyfikacji pozostałych metod z klasy `COknoGL`. W metodzie `COknoGL::WndProc` wczytamy i przygotujemy drugi zestaw shaderów. Wywołamy również przedstawioną wyżej metodę przygotowującą tekstury dla skyboksa. W tym celu zmienimy kod instrukcji `switch`, a konkretnie kod wykonywany w przypadku komunikatu `WM_CREATE` (tj. w momencie utworzenia okna) (listing 2.15). Następnie należy wprowadzić zmiany w metodzie `COknoGL::PrzygotujAktorów` zgodnie ze wzorem z listingu 2.16.

Listing 2.15. Wczytanie shaderów.

```
case WM_CREATE: //Utworzenie okna
    //zmienna uchwytOkna nie jest jeszcze zainicjowana
    if (!InicjujWGL(hWnd))
    {
        MessageBox(NULL, "Pobranie kontekstu renderowania nie powiodło się",
"Aplikacja OpenGL", MB_OK | MB_ICONERROR);
        exit(EXIT_FAILURE);
    }

    idProgramuShaderowSkybox = PrzygotujShadery(
        "Skybox.vert",
        nullptr,
        nullptr,
        nullptr,
        "Skybox.frag");

    idProgramuShaderow = PrzygotujShadery(
        "Sfera.vert",
        "Sfera.tesc",
        "Sfera.tese",
        "Sfera.geom",
        "Sfera.frag");


    if (idProgramuShaderowSkybox == NULL || idProgramuShaderow == NULL)
    {
        MessageBox(NULL, "Przygotowanie shaderów nie powiodło się", "Aplikacja
OpenGL", MB_OK | MB_ICONERROR);
        exit(EXIT_FAILURE);
    }

    UmieśćInformacjeNaPaskuTytułu(hWnd);
    if (teksturowanieWłączone)
        PrzygotujTekstury(3, new char*[3] { "tekstura1.bmp", "tekstura2.bmp",
"tekstura3.bmp" });
    else
    {
        glUniform1i(glGetUniformLocation(idProgramuShaderowSkybox,
"Teksturowanie"), false);
        glUniform1i(glGetUniformLocation(idProgramuShaderow, "Teksturowanie"),
false);
    }

    PrzygotujTeksturySkybox();
```

```
    liczbaAktorów = PrzygotujAktorów();
    UstawienieSceny();
    if (swobodneObrotyKameryMożliwe)
    {
       if (SetTimer(hWnd,
                    identyfikatorTimeraSwobodnychObrotówKamery,
                    okresTimeraSwobodnychObrotówKamery,
                    NULL) == 0)
             MessageBox(hWnd, "Nie udało się ustawić timera swobodnych
obrotów kamery", "Błąd", MB_OK | MB_ICONERROR);
    }
    if (animacjaMożliwa)
    {
       if (SetTimer(hWnd,
                    identyfikatorTimeraAnimacji,
                    okresTimeraAnimacji,
                    NULL) == 0)
             MessageBox(hWnd,
                        "Nie udało się ustawić timera animacji sceny",
                        "Błąd",
                        MB_OK | MB_ICONERROR);
    }
    break;
```

Listing 2.16. Nowa wersja metody `PrzygotujAktorów`.

```
unsigned int COknoGL::PrzygotujAktorów()
{
      GLuint atrybutPołożenie =
        glGetAttribLocation(idProgramuShaderow, "polozenie_in");
      if (atrybutPołożenie ==
        (GLuint)-1) atrybutPołożenie = 0;

      GLuint atrybutNormalna =
        glGetAttribLocation(idProgramuShaderow, "normalna_in");
      if (atrybutNormalna == (GLuint)-1)
        atrybutNormalna = 1;

      GLuint atrybutWspółrzędneTeksturowania =
        glGetAttribLocation(idProgramuShaderow, "wspTekstur_in");
      if (atrybutWspółrzędneTeksturowania == (GLuint)-1)
        atrybutWspółrzędneTeksturowania = 2;

      GLuint atrybutKolor =
        glGetAttribLocation(idProgramuShaderow, "kolor_in");
      if (atrybutKolor == (GLuint)-1)
        atrybutKolor = 3;

      GLuint atrybutPołożenieSkybox =
        glGetAttribLocation(idProgramuShaderowSkybox, "polozenie_in");
      if (atrybutPołożenieSkybox == (GLuint)-1)
        atrybutPołożenieSkybox = 0;

      const int liczbaAktorów = 2;
      aktorzy = new Aktor*[liczbaAktorów];

      // Bryły.
      aktorzy[0] = Skybox::StwórzSkybox(atrybutPołożenie, 9);
      aktorzy[0]->ProgramShaderowId = idProgramuShaderowSkybox;

      aktorzy[1] = new TeselowanaSfera(atrybutPołożenie, atrybutNormalna,
        atrybutWspółrzędneTeksturowania, atrybutKolor, 1.0f, 5, 5);
      aktorzy[1]->MacierzŚwiata =
```

```
        Macierz4::ObrótY(90) * Macierz4::ObrótX(90) *
Macierz4::Przesunięcie(0,0,0);
      aktorzy[1]->MateriałŚwiatłoOtoczenia = Wektor4(0.2f, 0.2f, 0.2f, 1);
      aktorzy[1]->MateriałŚwiatłoRozpraszane = Wektor4(1, 1, 1, 1);
      aktorzy[1]->MateriałŚwiatłoRozbłysku = Wektor4(1, 1, 1, 1);
      aktorzy[1]->MateriałWykładnikRozbłysku = 10;
      aktorzy[1]->IndeksTekstury =
        (teksturowanieWłączone) ? indeksyTekstur[0] : -1;
      aktorzy[1]->ProgramShaderowId = idProgramuShaderow;

      return liczbaAktorów;
}
```

Kolejnym krokiem jest modyfikacja metody `COknoGL::RysujAktorów` polegająca na dodaniu wywołania funkcji włączającej odpowiedni zestaw shaderów oraz dodaniu wiązania tekstury skyboksa z obydwoma zestawami shaderów (listing 2.17). Natomiast na końcu metody `UsuńTekstury` dodajemy polecenie usuwające tekstury użyte do skyboksa (listing 2.18). Po uruchomieniu projektu powinniśmy zobaczyć widok jak na rysunku 2.2. Oto skybox! Czyż nie poprawia znakomicie wyglądu sceny?

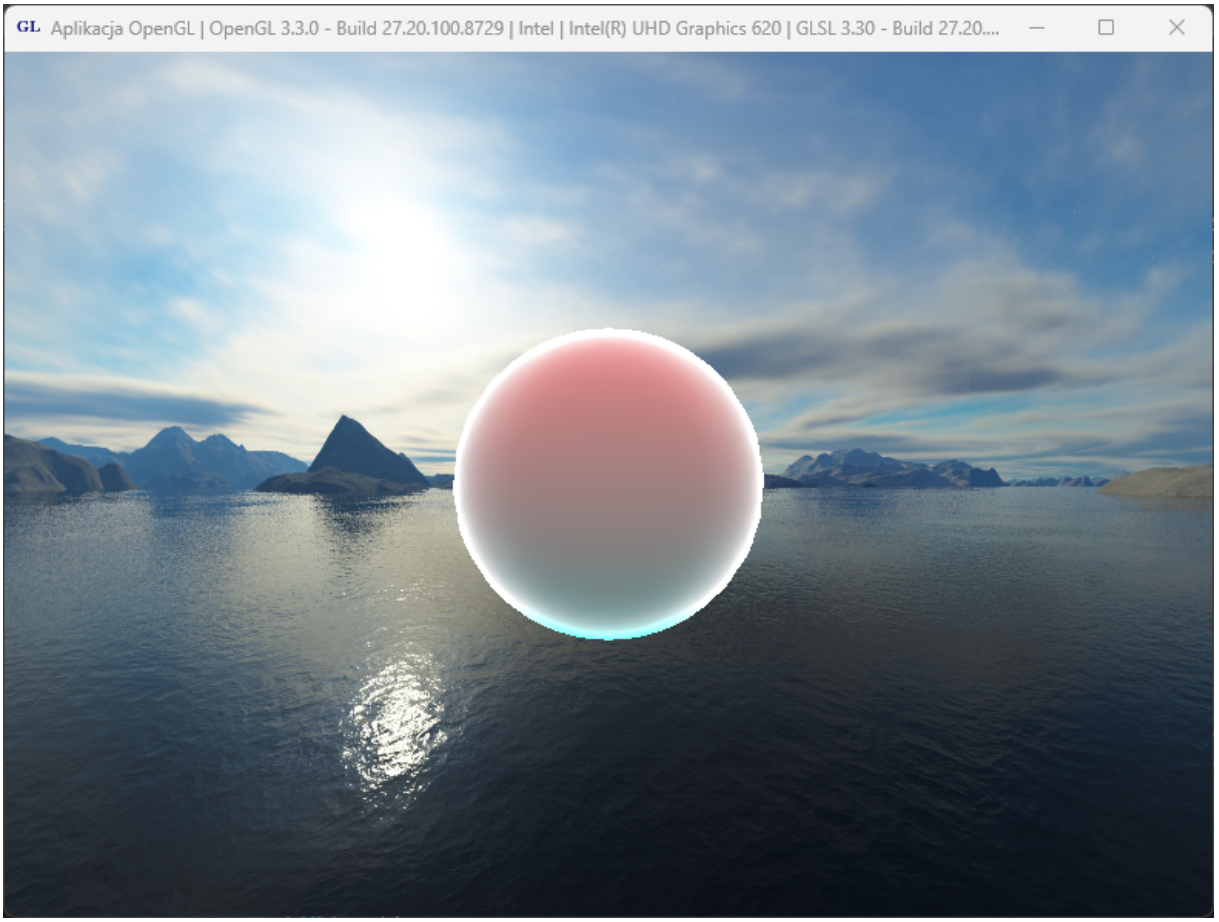
Listing 2.17. Nowa wersja metody `RysujAktorów`.

```
void COknoGL::RysujAktorów()
{
    for (unsigned int i = 0; i < liczbaAktorów; ++i)
    {
        UzyjProgramuShaderow(aktorzy[i]->ProgramShaderowId);

        //zachowuje związek z macierzą shaderów,
        //z tego powodu nie należy używać operatora =
        macierzŚwiata.Ustaw(aktorzy[i]->MacierzŚwiata);
        macierzŚwiata.PrześlijWartość();
        try
        {
              macierzNormalnych.Ustaw(
                    macierzŚwiata.Odwrotna().Transponowana());
        }
        catch (std::exception exc)
        {
              MessageBeep(0);
              macierzNormalnych.Ustaw(macierzŚwiata);
        }
        macierzNormalnych.PrześlijWartość();
        UstawParametryMateriału(
              aktorzy[i]->MateriałŚwiatłoOtoczenia,
              aktorzy[i]->MateriałŚwiatłoRozpraszane,
              aktorzy[i]->MateriałŚwiatłoRozbłysku,
              aktorzy[i]->MateriałWykładnikRozbłysku);

        if (teksturowanieWłączone)
        {
              glActiveTexture(GL_TEXTURE0);
              glBindTexture(GL_TEXTURE_2D, indeksyTekstur[0]);
              glUniform1i(glGetUniformLocation(
                    idProgramuShaderow, "ProbnikTekstury0"), 0);

              glActiveTexture(GL_TEXTURE1);
              glBindTexture(GL_TEXTURE_2D, indeksyTekstur[1]);
              glUniform1i(glGetUniformLocation(
                    idProgramuShaderow, "ProbnikTekstury1"), 1);

              glActiveTexture(GL_TEXTURE2);
              glBindTexture(GL_TEXTURE_2D, indeksyTekstur[2]);
```

```
                glUniform1i(glGetUniformLocation(
                      idProgramuShaderow, "ProbnikTekstury2"), 2);

                // Do mapowania środowiska dla sfery.
                glActiveTexture(GL_TEXTURE3);
                glBindTexture(GL_TEXTURE_CUBE_MAP, indeksTeksturySkybox);
                glUniform1i(glGetUniformLocation(
                      idProgramuShaderow, "ProbnikTeksturySkybox"), 3);

                // Do oteksturowania skyboksa.
                glActiveTexture(GL_TEXTURE4);
                glBindTexture(GL_TEXTURE_CUBE_MAP, indeksTeksturySkybox);
                glUniform1i(glGetUniformLocation(
                      idProgramuShaderowSkybox, "ProbnikTeksturySkybox"), 4);
        }

        aktorzy[i]->Rysuj();
    }
}
```

Listing 2.18. Nowa wersja metody `UsuńTekstury`.

```
void COknoGL::UsuńTekstury()
{
    glBindTexture(GL_TEXTURE_2D, NULL);
    glDeleteTextures(liczbaTekstur, indeksyTekstur);
    delete[] indeksyTekstur;
    liczbaTekstur = 0;

    glDeleteTextures(1, &indeksTeksturySkybox);
}
```



Rysunek 2.2 Scena ze skyboksem.

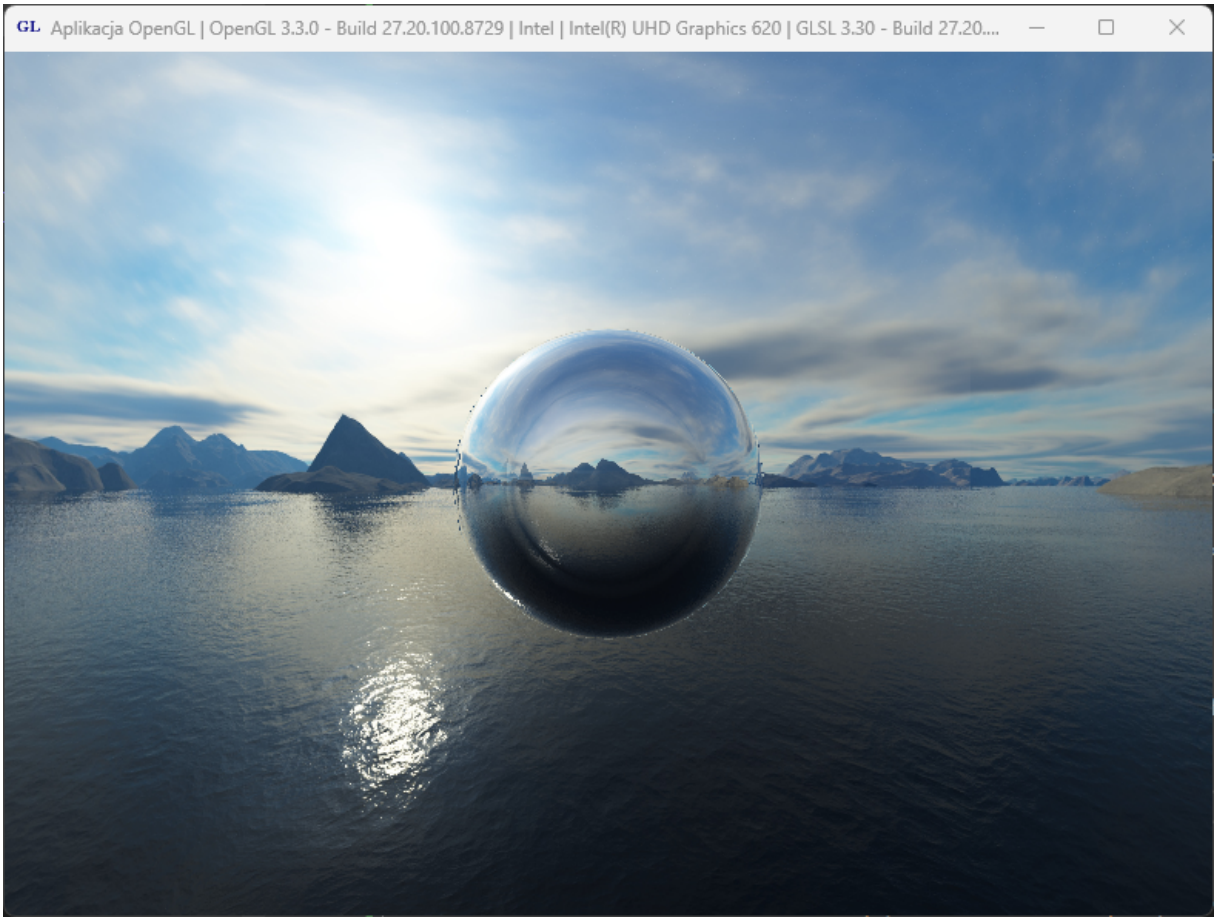
Mapowanie środowiska

Możemy jednak pójść jeszcze o krok dalej. Załóżmy, że sfera wisząca nad wodą jest przezroczystą gładką kulą. Tak jak patrząc na szybę z równoległej pozycji zauważamy zjawisko lustra, tak tutaj — dzięki zaimplementowanemu efektowi Fresnela — dostrzeżemy analogiczne zjawisko.. Powinien odbijać się w kuli obraz otoczenia. Taki efekt możemy uzyskać dzięki technice nazywanej mapowaniem środowiska. [15], do czego wystarczy jedynie zmienić shader fragmentów dla sfery zgodnie ze wzorem z listingu 2.19. Dodajemy dwie funkcje: `reflection`, która zwraca kolor otoczenia, który jest odbijany przez obiekt oraz `refraction`, która zwraca kolor otoczenia, który jest załamywany podczas przechodzenia przez obiekt. Następnie wykorzystujemy obliczone kolory w funkcji odpowiedzialnej za cieniowanie Fresnela, która miesza je w odpowiednich proporcjach. Powinniśmy uzyskać efekt widoczny na rysunku 2.3.

Listing 2.19. Shader fragmentów dla sfery realizujący mapowanie środowiska.

```
#version 460 core

in vec4 kolorGeomToFrag;
in vec3 polozenie_scena_geom;
in vec3 normalna_scena_geom;
in vec2 wspTekstur_geom;

uniform vec3 PolozenieKamery = vec3(0,0,1);
uniform sampler2D ProbnikTeksturySfery;
uniform samplerCube ProbnikTeksturySkybox;

out vec4 FragColor;

vec4 reflection(vec3 I, vec3 N)
{
    vec3 R = reflect(I, N);
    vec4 environmentColor = texture(ProbnikTeksturySkybox, R);
    return environmentColor;
}

vec4 refraction(vec3 I, vec3 N)
{
    const float etaRatio = 1.5;
    vec3 T = refract(I,N, etaRatio);
    vec4 environmentColor = texture(ProbnikTeksturySkybox, T);

    return environmentColor;
}

vec4 fresnel(vec3 I, vec3 N, vec4 reflectedColor, vec4 refractedColor)
{
    float bias = 0.2;
    float scale = 3;
    float power = 1.3;

    float reflectionFactor = clamp(bias+scale*pow(1.0+dot(I,N),power),0,1);


    vec4 finalColor = mix(refractedColor,reflectedColor,reflectionFactor);
    return finalColor;
}

vec4 CieniowanieFresnela(vec4 argkolor)
{
    vec3 I = normalize(polozenie_scena_geom - PolozenieKamery);
    vec3 N = normalize(normalna_scena_geom);
```

```
    vec4 reflectedColor = reflection(I, N);
    vec4 refractedColor = refraction(I, N);

    return fresnel(I, N, reflectedColor, refractedColor);
}

void main()
{
    FragColor = CieniowanieFresnela(FragColor);
}
```



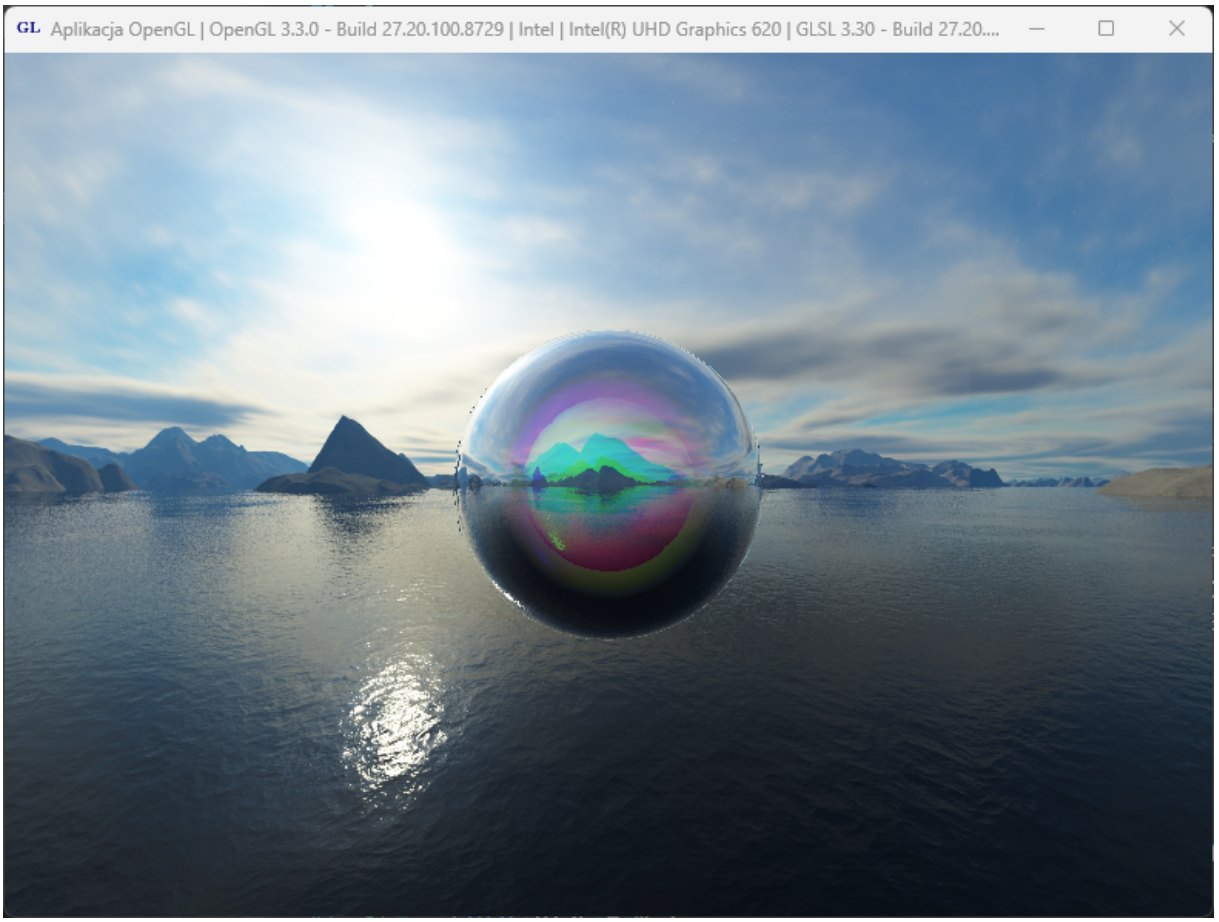
Rysunek 2.3 Widoczne mapowanie środowiska.

Dyspersja chromatyczna

Jednak to nadal nie koniec. Możemy dodać jeszcze jeden efekt, który będzie imitował rozpraszanie światła i jego wpływ na wygląd obiektów. Mowa tutaj o dyspersji chromatycznej. Dyspersja chromatyczna jest zjawiskiem, w którym światło białe jest rozszczepiane na składowe barwne (w grafice komputerowej na składowe: czerwoną, zieloną i niebieską) podczas przechodzenia przez jakieś medium [16]. W efekcie, różne długości fal świetlnych rozchodzą się pod różnymi kątami, co prowadzi do efektu tęczy lub kolorowych aureoli wokół obiektów. [12]
To będzie już bardzo prosta zmiana. Wystarczy zmodyfikować funkcję `refraction` w shaderze *Sfera.frag* (listing 2.20). Oddzielnie obliczamy wektory załamania światła dla każdej składowej. Kąty załamania składowych zależą od współczynników przechowywanych w wektorze `etaRatio`, które mogą być dowolnie dostosowywane. Gotowe! Finalny efekt możemy podziwiać na rysunku 2.4.

Listing 2.20. Zmodyfikowana funkcja w celu uzyskania rozszczepienia chromatycznego.

```
vec4 refraction(vec3 I, vec3 N)
{
    const vec3 etaRatio = vec3(1.1, 1.7, 1.4);
    vec3 TRed   = refract(I, N, etaRatio.r);
    vec3 TGreen = refract(I, N, etaRatio.g);
    vec3 TBlue  = refract(I, N, etaRatio.b);

    vec4 environmentColor;
    environmentColor.r = texture(ProbnikTeksturySkybox, TRed).r;
    environmentColor.g = texture(ProbnikTeksturySkybox, TGreen).g;
    environmentColor.b = texture(ProbnikTeksturySkybox, TBlue).b;
    environmentColor.a = 1.0;

    return environmentColor;
}
```



Rysunek 2.4 Widoczna dyspersja chromatyczna.

Rozdział 3.
Pytania i zadania

Na koniec chciałbym zaproponować krótki zestaw pytań oraz kilka zadań praktycznych. Zadania testowe pozwolą ocenić czytelnikowi poziom zrozumienia zagadnień programowania grafiki 3D poruszonych w pracy. Natomiast zadania praktyczne umożliwią przećwiczenie nabytej wiedzy poprzez modyfikację i dalszy rozwój kodu programu. Zachęcam do odpowiedzenia na pytania i rozwiązania zadań.

Pytania testowe

Poniżej umieszczam zbiór pytań zamkniętych. Dotyczą one zarówno szczegółowych kwestii poruszonych w pracy, jak również aspektów bardziej ogólnych. W każdym pytaniu jest tylko jedna poprawna odpowiedź. Poprawne odpowiedzi są wymienione na koniec rozdziału.

1. Co to jest teselacja?
    a. Technika zwiększania liczby trójkątów.
    b. Technika zmniejszania liczby trójkątów.
    c. Metoda implementacji cieniowania Fresnela.
    d. Technika realizacji realistycznego oświetlenia sceny.

2. Jaka jest kolejność shaderów w potoku renderowania?
    a. kontroli teselacji; ewaluacji teselacji; wierzchołków; geometrii; fragmentów.
    b. wierzchołków; kontroli teselacji; ewaluacji teselacji; geometrii; fragmentów.
    c. kontroli teselacji; geometrii; fragmentów; wierzchołków; ewaluacji teselacji.
    d. geometrii; wierzchołków; kontroli teselacji; ewaluacji teselacji; fragmentów.

3. W jakim języku programuje się shadery w OpenGL?
    a. HLSL
    b. C
    c. GLSL
    d. C++

4. Czym jest dyspersja chromatyczna?
    a. Zaawansowaną techniką oświetlania sceny.
    b. Sposobem wydajnego programowania shaderów.
    c. Metodą teselacji.
    d. Rozszczepianiem światła na składowe barwne podczas przechodzenia przez ośrodek.

5. Do czego służy skybox?
    a. Do tworzenia chmur i obłoków na niebie w grach komputerowych.
    b. Do symulowania odległego otoczenia sceny.
    c. Do ograniczenia obszaru renderowania w celu zwiększenia wydajności.
    d. Do renderowania realistycznych cieni.

6. Co to jest efekt Fresnela?
    a. Zjawisko, w którym światło jest pochłaniane przez obiekt.
    b. Zjawisko, w którym odbijane i załamywane światło zależy od kąta padania światła i kąta patrzenia.
    c. Zjawisko rozszczepienia światła na składowe barwne.
    d. Zjawisko, w którym światło nie jest załamywane podczas przechodzenia przez ośrodek.

7. W którym shaderze ustawiamy poziomy teselacji?
    a. Shader wierzchołków.
    b. Shader geometrii.
    c. Shader ewaluacji teselacji.
    d. Shader kontroli teselacji.

8. Czy shader geometrii jest obowiązkowy?
    a. Tak — zawsze.
    b. Tak jeśli obecne są shadery teselacji.
    c. Nie.
    d. Tak jeśli obecny jest shader wierzchołków.

9. Co to jest mapowanie środowiska?
    a. Uwzględnienie otoczenia w wyglądzie obiektu.
    b. Jest to inna nazwa na skybox.
    c. Jest to inna nazwa na cieniowanie Fresnela.
    d. Uwzględnienie obiektów w wyglądzie otoczenia.

10. Z ilu wierzchołków składa się płat?
    a. Z czterech.
    b. Z trzech
    c. Liczba ta zależy od programisty.
    d. Płaty nie posiadają wierzchołków.

Zadania praktyczne

Poniżej umieszczam zestaw praktycznych zadań, które wymagają zrozumienia i modyfikacji kodu programu. Część z nich dotyczy *dostrajania* różnorodnych parametrów renderowania, a część wymaga większych zmian zarówno w głównym kodzie programu C++, jak i w programach shaderów GLSL.

1. W shaderze kontroli teselacji przetestuj różne ustawienia poziomów teselacji zewnętrznej i wewnętrznej.

2. W shaderze fragmentów wypróbuj różne ustawienia parametrów cieniowania Fresnela: `bias`, `scale`, `power`.

3. W shaderze fragmentów zmodyfikuj współczynniki załamania światła `etaRatio`.

4. Stwórz na scenie dwie sfery: jedną przy użyciu teselacji, drugą w tradycyjny sposób. Będą one wykorzystywać różne zestawy shaderów.

5. Utwórz aktora złożonego z czworokątnych płatów oraz zestaw shaderów, który dokona teselacji takiej figury.

Odpowiedzi do pytań testowych

Lista odpowiedzi do pytań zamkniętych.

1. a
2. b
3. c
4. d
5. b
6. b
7. d
8. c
9. a
10. c

Podsumowanie

Niniejsza praca była poświęcona programowaniu grafiki trójwymiarowej w bibliotece OpenGL i języku shaderów GLSL. Głównymi celami pracy było zaprezentowanie dwóch zaawansowanych technik grafiki 3D: teselacji oraz efektu fresnela. Cele te zostały osiągnięte poprzez opis teoretyczny zagadnień oraz przedstawienie procesu ich implementacji. Ponadto omówione zostało kilka dodatkowych technik, które można zrealizować za pomocą shaderów GLSL.

W rozdziale dotyczącym teselacji omówiona została teoria dotycząca tego zagadnienia, a następnie dokładnie omówiliśmy implementację, wykorzystując pięć shaderów potoku graficznego. Przykładem zastosowania teselacji było stworzenie prostego wielościanu, który zmieniliśmy w rzeczywistą sferę z użyciem shaderów teselacji i geometrii.

Drugi rozdział przedstawia cieniowanie Fresnela, ale również tworzenie skyboksa, mapowanie środowiska oraz dyspersję chromatyczną. Po wprowadzeniu w zagadnienie, zaprezentowaliśmy główny efekt tj. cieniowanie Fresnela, który symuluje sposób odbijania i załamywania światła w zależności od kąta padania i kąta patrzenia. Następnie rozszerzyliśmy kod, aby obsługiwał wiele zestawów shaderów, co umożliwiło utworzenie skyboksa i poprawę wyglądu sceny. Przedstawiliśmy również efekt mapowania środowiska, tj. wpływ wyglądu środowiska na kolor obiektu. Na koniec zaimplementowaliśmy efekt dyspersji chromatycznej.

Trzeci rozdział umożliwia czytelnikowi sprawdzenie swojej wiedzy oraz przećwiczenie pracy z kodem. W tym celu zaproponowano zbiór pytań zamkniętych oraz zestaw zadań do wykonania.

Wszystkie zagadnienia zostały przedstawione w sposób praktyczny i łatwy do zrozumienia. Zmiany w kodzie były prezentowane na listingach. Natomiast otrzymywane efekty wizualne były przedstawiane na zrzutach ekranu. Praca ta rozwinęła kod aplikacji tworzonej w książce *"Grafika 3D czasu rzeczywistego: Nowoczesny OpenGL"* i pozostawia zachętę ku dalszemu rozwojowi programu.

Literatura

[1] J. Matulewski, *Grafika 3D czasu rzeczywistego. Nowoczesny OpenGL*, Wydawnictwo Naukowe PWN, Warszawa 2014

[2] *https://registry.khronos.org/OpenGL-Refpages/gl4* – Dokumentacja GLSL 4

[3] *https://www.khronos.org/opengl/wiki/OpenGL_Reference* – Dokumentacja OpenGL

[4] *https://en.wikipedia.org/wiki/Tessellation_(computer_graphics)* – Teselacja

[5] P. Kiciak, *OpenGL i GLSL (nie taki krótki kurs): Część 3*, Wydawnictwo Naukowe PWN, Warszawa 2019, 551-593

[6] *https://www.khronos.org/opengl/wiki/Tessellation* – Teselacja w dokumentacji OpenGL.

[7] P. Kiciak, *OpenGL i GLSL (nie taki krótki kurs): Część 1*, Wydawnictwo Naukowe PWN, Warszawa 2019, 10-13

[8] *https://www.khronos.org/opengl/wiki/Geometry_Shader* – Shader geometrii w dokumentacji OpenGL

[9] K. Sobiesiak, P. Sydow, *Shadery: zaawansowane programowanie w GLSL*, Wydawnictwo Naukowe PWN, Warszawa 2015, 213-219

[10] *https://www.khronos.org/opengl/wiki/Texture* – Teksturowanie w dokmentacji OpenGL.

[11] R. Fernando, M. Kilgard, *Język Cg. Programowanie grafiki w czasie rzeczywistym*, Helion, Gliwice 2003, 169-194

[12] *https://pl.wikipedia.org/wiki/R%C3%B3wnania_Fresnela* – Efekt Fresnela

[13] R. Wright, N. Haemel, G. Sellers, B. Lipchak. *OpenGL. Księga eksperta. Wydanie 5*, Helion, Gliwice 2011, 296-308

[14] *https://ogldev.org/www/tutorial25/tutorial25.html* – Tworzenie skyboksa

[15] *https://learnopengl.com/Advanced-OpenGL/Cubemaps* – Mapowanie środowiska

[16] *https://en.wikipedia.org/wiki/Dispersion_(optics)* – Dyspersja chromatyczna