Rafał Balcerowski

Cieniowanie Fresnela, mapowanie środowiska i dyspersja chromatyczna

Fragment pracy inżynierskiej Rafała Balcerowskiego z 2023 roku napisanej pod kierunkiem Jacka Matulewskiego.
Wersja z dnia 26 lipca 2023 roku.
Prezentowany projekt rozwija kod omówiony w podręczniku
„Grafika 3D czasu rzeczywistego. Nowoczesny OpenGL"
PWN 2014, 2021 (ISBN: 978-83-01-17966-3)
https://ksiegarnia.pwn.pl/Grafika-3D-czasu-rzeczywistego,68451737,p.html
oraz w kursach OpenGL i GLSL dostępnych na stronach:
https://jacekmatulewski.fizyka.umk.pl/dydaktyka/3d/
https://jacekmatulewski.fizyka.umk.pl/dydaktyka/3d/glsl/

Uwaga! *W tej części rozwijany jest kod przygotowany w części dotyczącej teselacji.*

1. 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. 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.

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

Wzór 2.1 Przybliżenie równania Fresnela

2. 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.

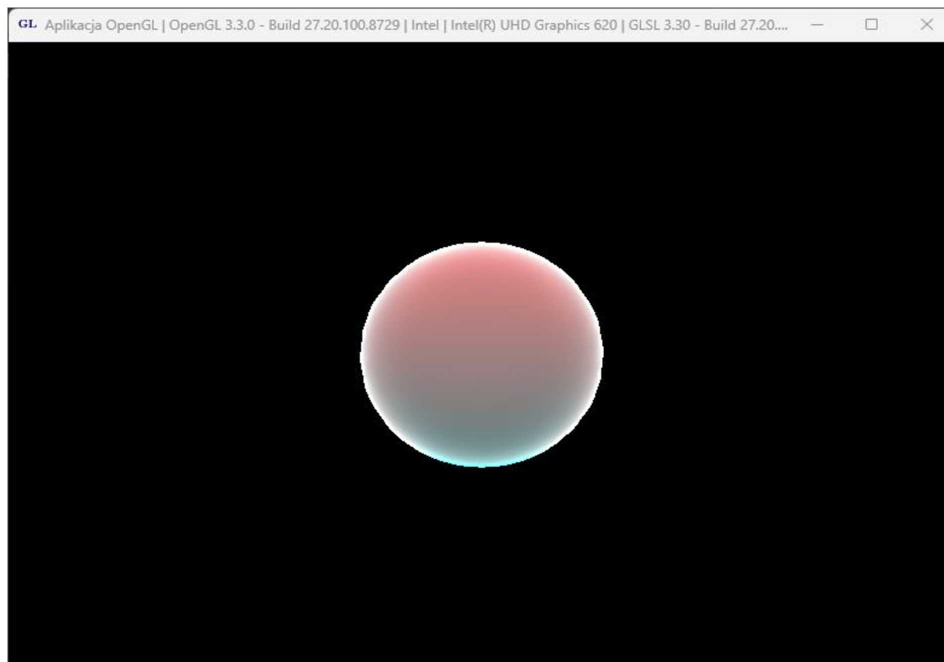
```
#version 460 core

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.

3. 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.

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 każdemu 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?

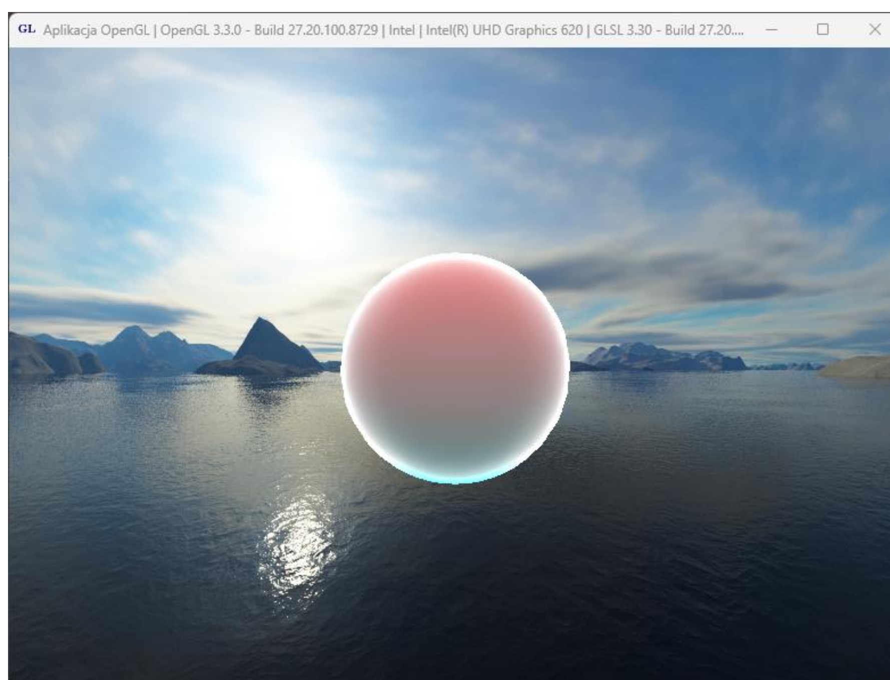
Listing 2.17. Nowa wersja metody `RysujAktorów`.

```
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.

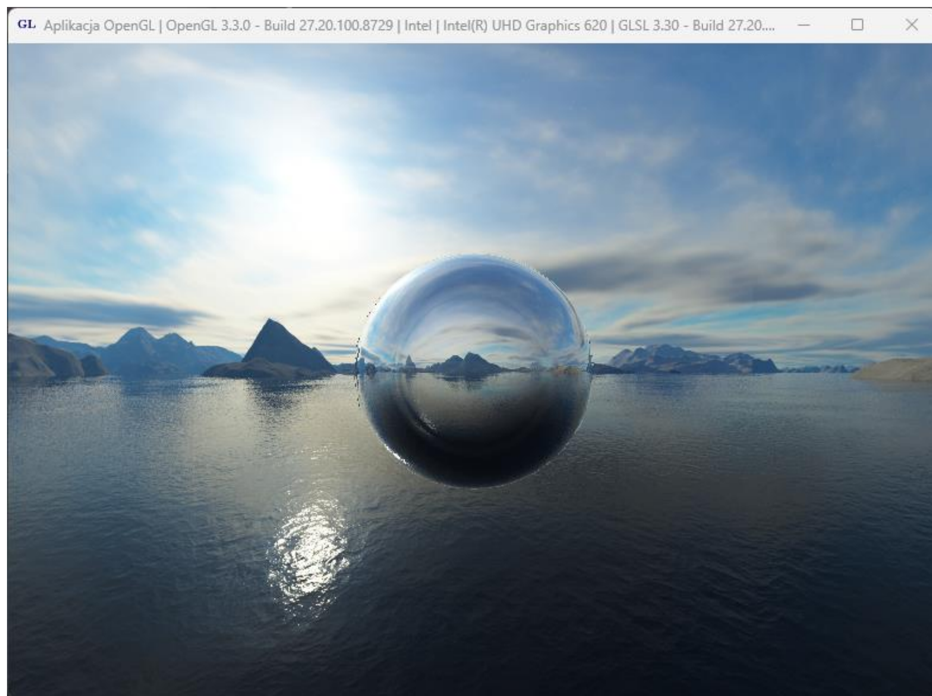
4. Mapowanie środowiska

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, 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.

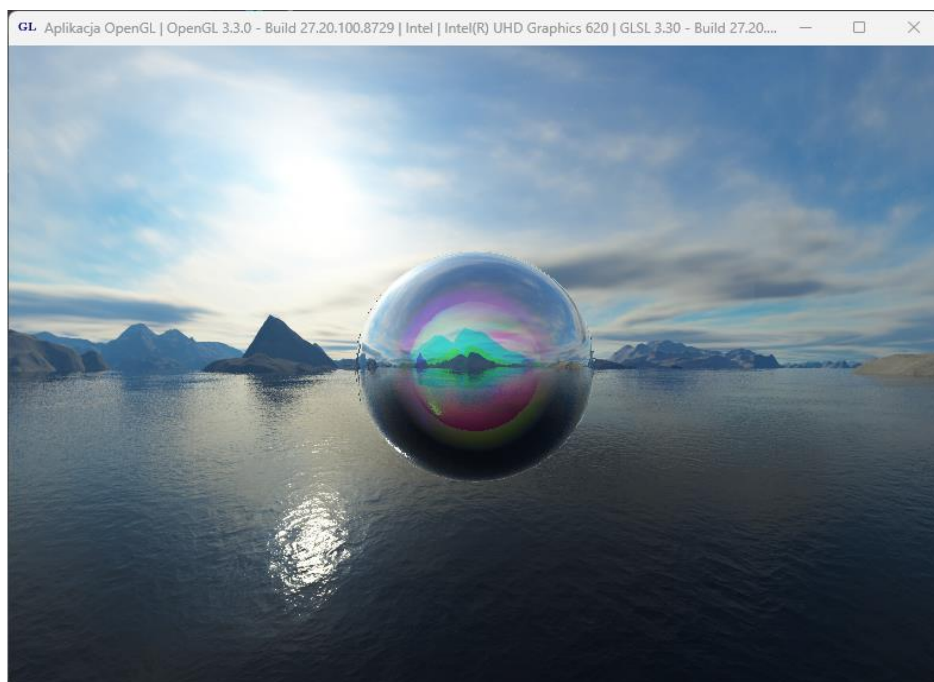
5. Dyspersja chromatyczna

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. W efekcie, różne długości fal świetlnych rozchodzą się pod różnymi kątami, co prowadzi do efektu tęczy lub kolorowych aureoli wokół obiektów.

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