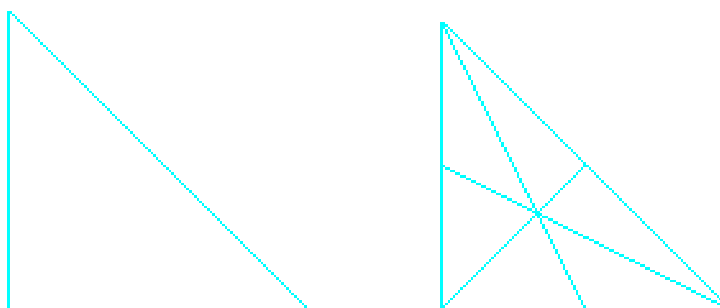
Rafał Balcerowski

# Teselacja

*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/*

## 1. Wstęp

Teselacja jest techniką zagęszczania siatki werteksów obiektów trójwymiarowych w grafice komputerowej. 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.

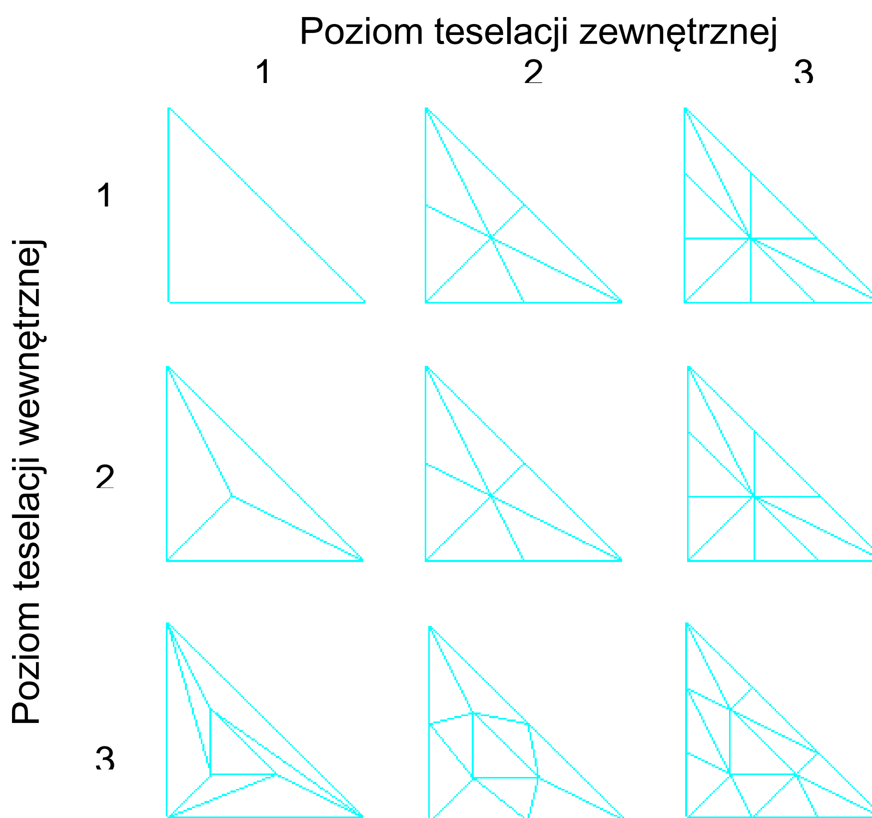


Rysunek 1.1 Trójkąt przed teselacją i po teselacji

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

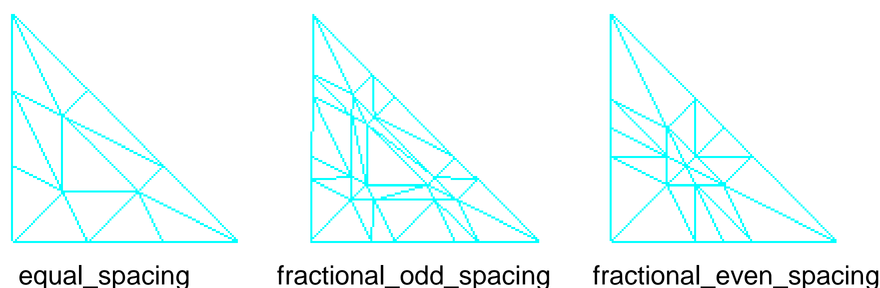
ustawić tutaj rozmiar płatu oraz poziom teselacji zewnętrznej i wewnętrznej. Z poziomów tych wynika, ile nowych wierzchołków zostanie wygenerowanych i w których miejscach. Teselacja zewnętrzna może być ustawiona oddzielnie dla każdego boku. Określa ona na ile części bok ma zostać podzielony poprzez dodanie wierzchołków. Natomiast teselacja wewnętrzna dodaje wierzchołki wewnątrz płata tak, aby liczba odcinków pomiędzy wejściowymi wierzchołkami była zgodna z ustawionym parametrem (por. rys. 1.2).




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.




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

# 2. Implementacja

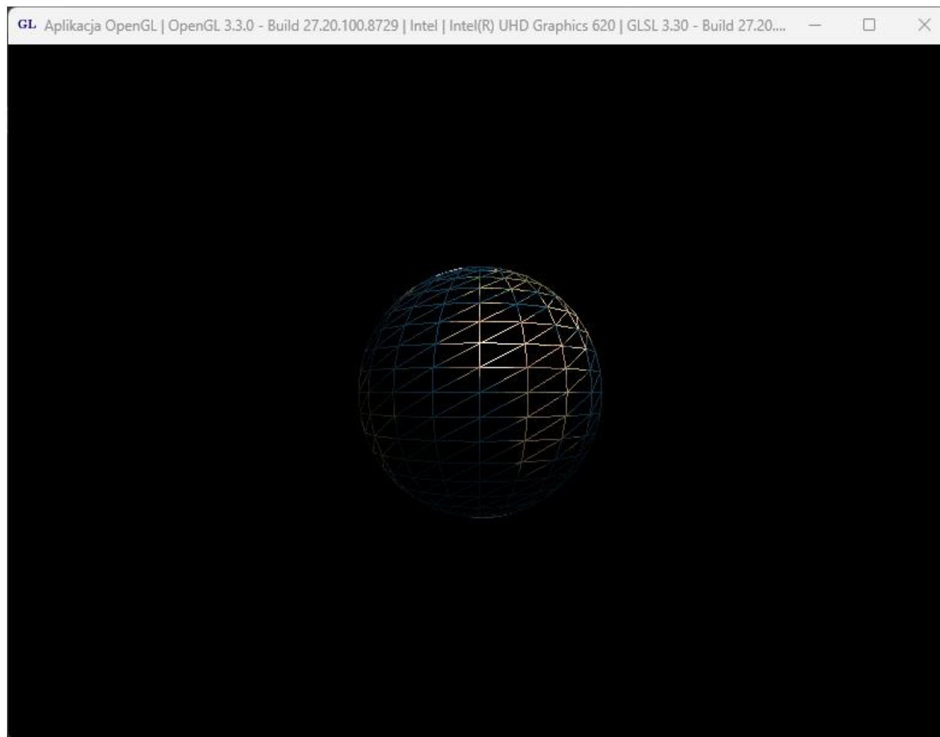
Aby zaimplementować własny przebieg teselacji, zmodyfikujemy i rozbudujemy kod projektu startowego[1], do którego dodamy dwa pliki shaderów.

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 TwórzTablicęIndeksów(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);

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

	InicjujBuforIndeksów();
}

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

	GLuint* indeksy = NULL;
	liczbaIndeksów = TwórzTablicęIndeksów(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 TwórzTablicęWerteksów(CWerteks*& werteksy);
	unsigned int TwórzTablicęIndeksów(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::TwórzTablicęWerteksów(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 =
```

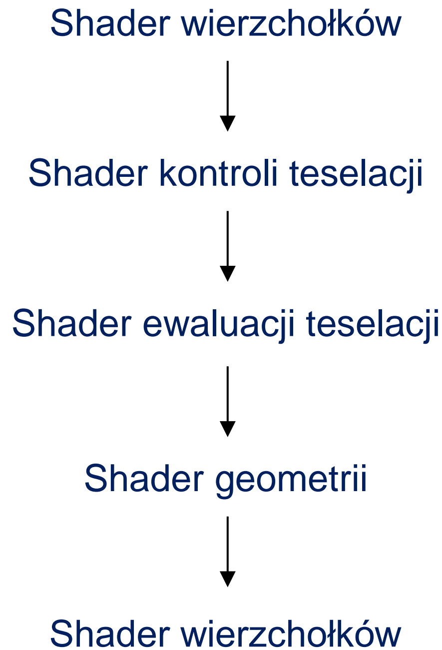
```
                                werteksy[indeks].x / _promień;
                        werteksy[indeks].ny =
                                werteksy[indeks].y / _promień;
                        werteksy[indeks].nz =
                                werteksy[indeks].z / _promień;
                }
                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::TwórzTablicęIndeksów(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. 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;
```

[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 podręcznika "Grafiki 3D czasu rzeczywistego. Nowoczesny OpenGL".*

```
	if (fileNameVertexShader != nullptr)
	{

		idShaderaWerteksów = KompilujShader(
			fileNameVertexShader,
			GL_VERTEX_SHADER,
			trybDebugowania);

		if (idShaderaWerteksów == 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, idShaderaWerteksów);
	}

	//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);

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

	//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);
```

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

      //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);

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

      //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.
```

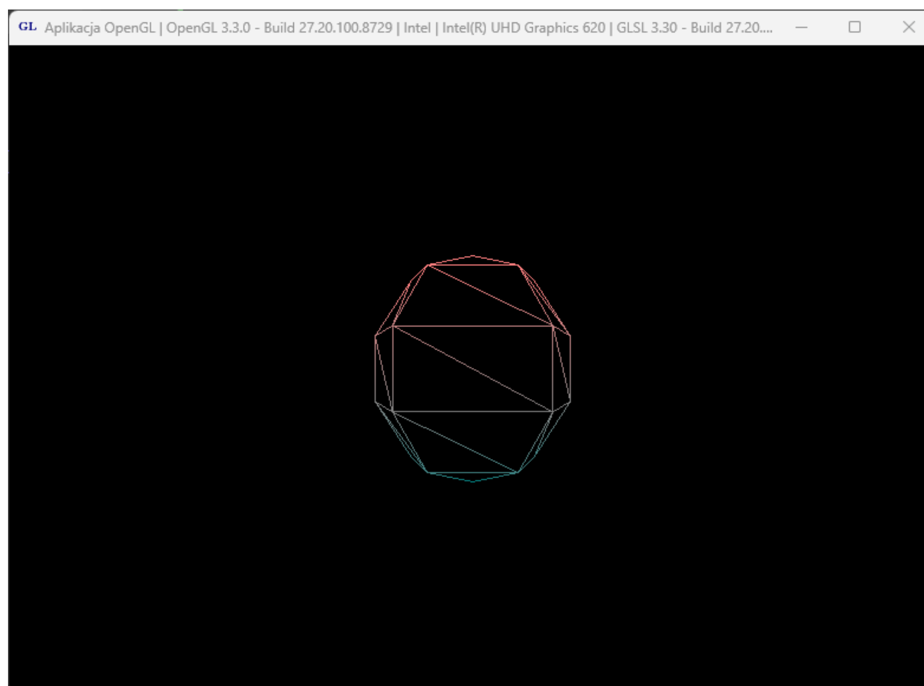
```
	glValidateProgram(idProgramu);

	//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.

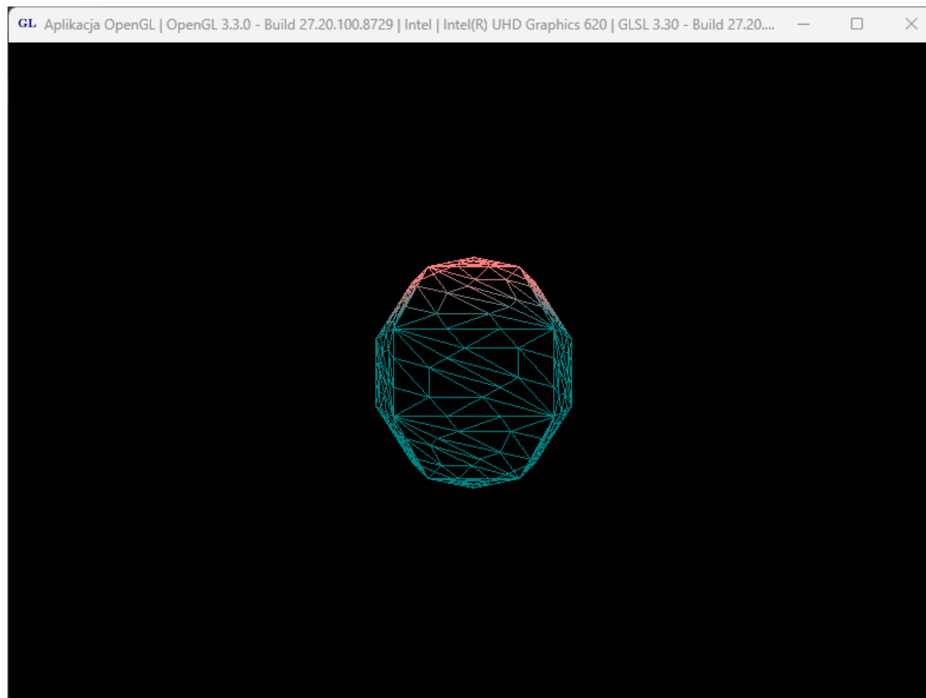
```
	switch (message)
	{
	case WM_CREATE: //Utworzenie okna
		//zmienna uchwytOkna nie jest jeszcze zainicjowana
		if (!InicjujWGL(hWnd))
		{
			MessageBox(NULL, "Pobranie kontekstu renderowania nie 
powiodło się", "Aplikacja OpenGL", MB_OK | MB_ICONERROR);
			exit(EXIT_FAILURE);
		}

		// (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 `TwórzTablicęWerteksów`, 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.

# 3. 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. 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.

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;
}
```

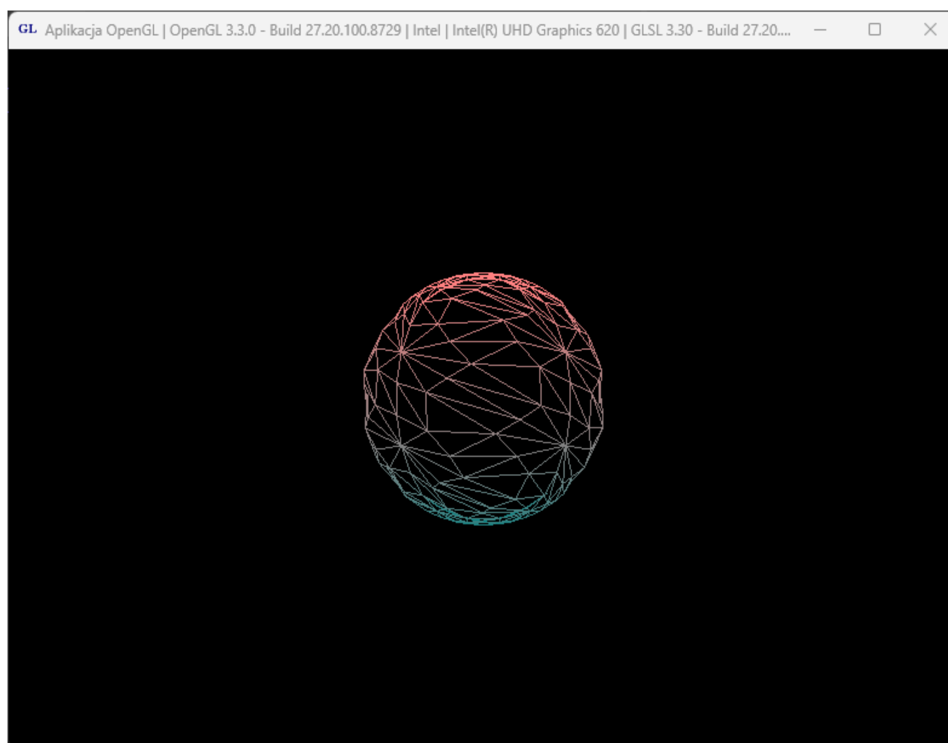
Listing 1.19. Zmieniony kod shadera ewaluacji teselacji.

```
#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;
}
```

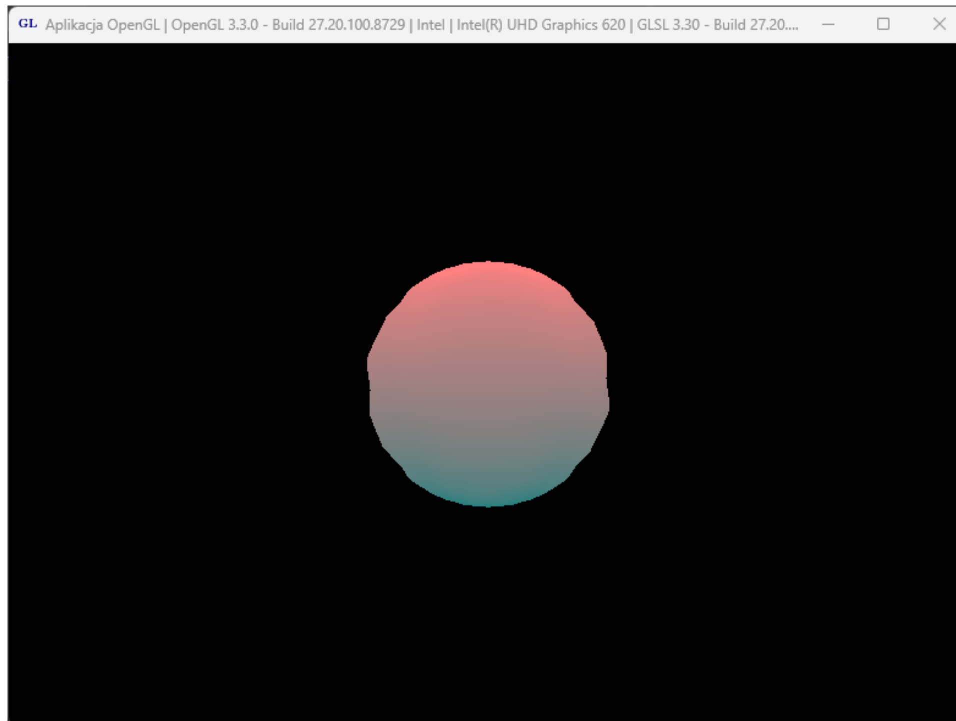
Gotowe! Teraz po uruchomieniu programu powinniśmy zobaczyć efekt widoczny na rysunku 1.8. Siatka jest zagęszczona, ale również wygładzona. Przywróćmy jeszcze tryb renderowania z wypełnieniem ścian (listing 1.21), co da finalny efekt widoczny na rysunku 1.9.




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.

# 4. 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. Efekt możemy zobaczyć na rysunku 1.10. W dalszej części dotyczącej cieniowania Fresnela i mapowania środowiska nie będziemy potrzebować tekstury, zatem po obejrzeniu uzyskanego efektu, możemy wycofać właśnie wprowadzone zmiany.

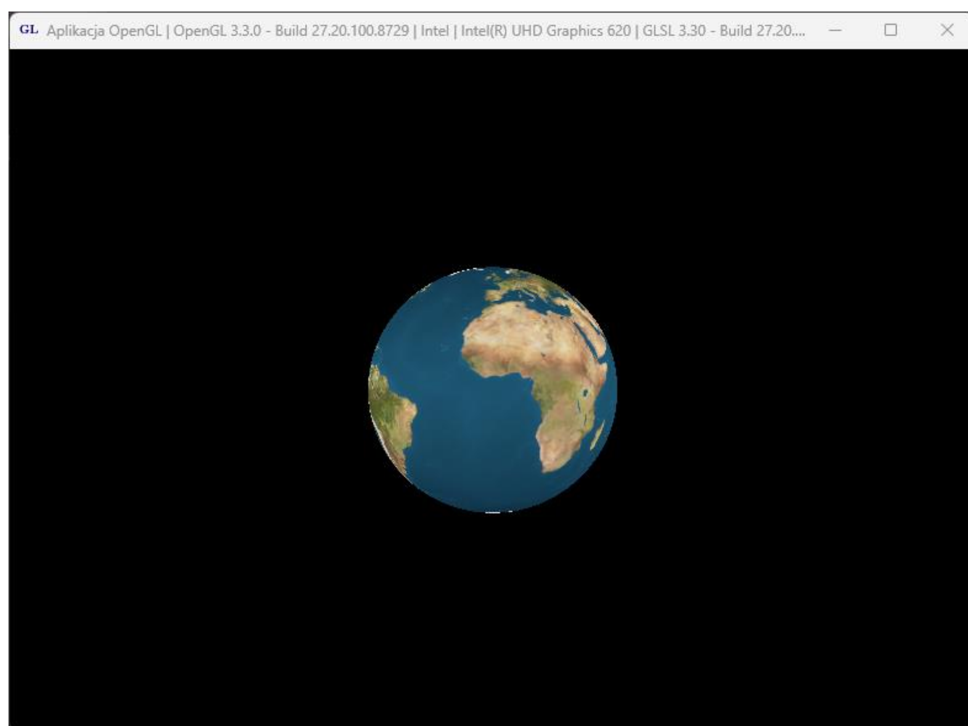
Listing 1.22. Modyfikacje dokonane w shaderze geometrii.

```
...

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.