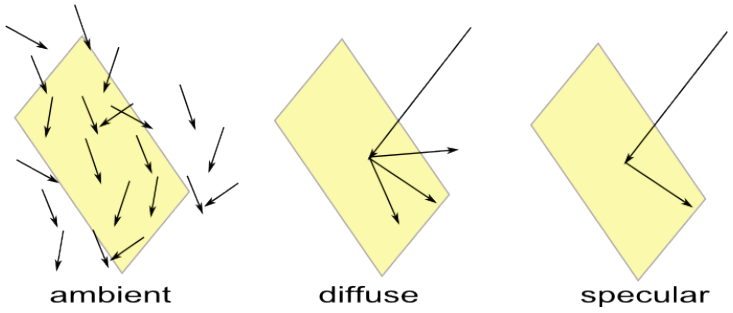
1 Podstawy modelu oświetlenia OpenGL

Zastosowanie odpowiedniego oświetlania obiektów przestrzennych w *OpenGL* decyduje o poziomie realizmu tworzonych scen. Efekty świetlne pozwalają na modelowanie powierzchni obiektów określając, czy mają być one np. matowe czy błyszczące. Tematem pierwszej części ćwiczeń laboratoryjnych będzie zdefiniowanie parametrów modelu oświetlenia globalnego sceny oraz wprowadzenie prostego, kierunkowego źródła światła w obszar widziany przez obserwatora.

1.1 Podstawy modelu oświetlenia, światło otaczające

Modelowanie efektu oświetlania obiektów przestrzennych składa się zasadniczo z dwóch faz: analizy emisji źródła oraz obliczenie odbicja (refleksu) obiektu, który jest przez źródło oświetlany. Biblioteka *OpenGL* oferuje trzy zasadnicze sposoby modelowania oświetlania.



Rysunek 1: Interpretacja podstawowych składowych modelu oświetlenia

Składowa światła otaczającego (*ambient*) reprezentuje światło silnie rozproszone. Promienie świetlne padają na scenę z różnych kierunków, a analiza efektów oświetlania nie pozwala na jednoznaczne wskazanie położenia źródła światła na scenie. Promienie składowej dyfuzyjnej *diffuse* pochodzą ze ściśle określonego źródła w przestrzeni sceny lecz po odbiciu od obiektu ulegają rozproszeniu. Składowa dyfuzyjna pozwala na modelowanie obiektów wykonanych z materiałów matowych. Trzecia składowa - światło odbite (*specular*) reprezentuje promieniowanie, które pada ze ściśle określonego punktu w przestrzeni (źródła),a po odbiciu się od obiektu promienie odbite podążają wzdłuż określonej prostej, której kierunek jest wyznaczony z prawa odbicia. Obecnie przećwiczmy sterowanie podstawową składową w modelu oświetlenia - składową światła otaczającego (ambient). Model oświetlania implementowany jest z wykorzystaniem shadera fragmentów. Przypomnijmy, że podstawowym zadaniem tego shadera jest wyliczenie koloru aktualnie przetwarzanego fragmentu prymitywu. W przypadku implementowania modelu oświetlenia należy założyć, że kolor wyjściowy obiektu zależeć będzie nie tylko od zdefiniowanego koloru rysowania C_m (nazywanego również kolorem materiału), ale także od panujących warunków oświetleniowych. W modelu światła otaczającego można wprowadzić kolor źrodła światła C_l oraz intensywność świecenia A_l rzutujące na wyjściowy kolor fragmentu C_f według równania:

$$C_f = C_m C_l A_l \tag{1}$$

Naturalnie, równanie powyższe należy traktować z uwzględnieniem wektorowego charakteru zmiennych opisujących kolory:

$$\left[C_f^R, C_f^G, C_f^B\right]^T = \left[C_m^R, C_m^G, C_m^B\right]^T \left[C_l^R, C_l^G, C_l^B\right]^T A_l \tag{2}$$

Powyższe równanie pozwala modelować znane z natury zjawisko oddawania barw oraz modulacji intensywności oświetlenia. Obecnie spróbujemy zaimplementować powyższy model światła otaczającego z wykorzystaniem shadera fragmentów.

1.1.1 Ćwiczenia

1. Utwórz nowy projekt programistyczny i podłącz do niego pliki źródłowe pobrane z serwera materiałów dydaktycznych

2. Skompiluj i uruchom program. W bieżącym stadium rozwoju aplikacja tworzy na scenie zieloną kostkę. Zwróć uwagę na sposób implementacji obiektu w metodzie `PrepareObjects` klasy implementującej scenę. Wszystkie ściany kostki rysowane są w tym samym odcieniu, przez co zlewają się przy krawędziach.

3. Uzupełnij kod rysowania sceny o polecenia tworzące macierz transformacji `mTransform` i włączające ją w postać macierzy model-widok. Dzięki takiemu rozdziałowi przekształceń macierzowych możliwe jest przechowywanie macierzy model-widok związanej z ustalonym położeniem obserwatora. W tym układzie odniesienia rysowany jest też statyczny układ współrzędnych

```
// narysuj bazowy uklad odniesienia
       Axes->Draw();

       // przygotuj transformacje ukladu do rysowania obiektu
       glm::mat4 mTransform = glm::mat4(4);
       mTransform = glm::rotate(mTransform, rot_x, glm::vec3(1.0f, 0.0f, 0.0f));
       mTransform = glm::rotate(mTransform, rot_y, glm::vec3(0.0f, 1.0f, 0.0f));
       glUniformMatrix4fv(_ModelView, 1, GL_FALSE, glm::value_ptr(mModelView*mTransform));

       // narysuj obiekt
       Cube->Draw();
```

4. Transformacja pozwala na rotowanie obiektu z użyciem klawiszy strzałkowych.

5. Wprowadź jako składową prywatną sceny zmienną typu skalarnego float `LightAmbient`. Będzie to zmienna przechowująca intensywność oświetlenia A_l.

6. W konstruktorze sceny zainicjuj zmienną wartością 0.5. Zapewnij sterowanie wartością zmiennej z wykorzystaniem klawiszy F1 oraz F2 (częściowo zaimplementowane w metodzie `KeyPressed`). Spraw, aby każdorazowe przyciśnięcie wymienionych klawiszy powodowało zmianę wartości zmiennej o +0.1

7. Zmodyfikuj kod shadera fragmentów, aby zaimplementować równanie modelujące światło ambient

```
#version 330

uniform vec3 LightColor;
uniform float LightAmbient;

in vec4 kolorek;

out vec4 outputColor;

void main()
{
       outputColor = kolorek*vec4(LightColor*(LightAmbient),1.0);
}
```

8. W kodzie rysowania sceny Zapewnij przekazanie wartości zmiennych jednorodnych z kodu aplikacji. Oczywiście musi ono nastąpić przed narysowaniem obiektu, który ma być poddany transformacji modelem oświetlenia

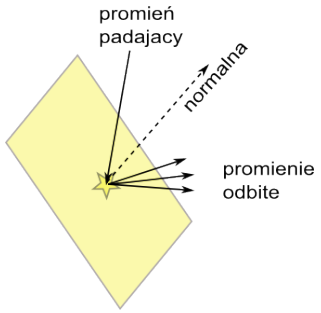
```
glm::vec3 LightColor = glm::vec3(1.0,1.0,1.0); // kolor swiatla
int _LightColor = glGetUniformLocation(program, "LightColor");
glUniform3fv(_LightColor,1,glm::value_ptr(LightColor));

int _LightAmbient = glGetUniformLocation(program, "LightAmbient");
glUniform1f(_LightAmbient,LightAmbient);
```

9. Skompiluj i uruchom program. Sprawdź, jak zmiana wartości zmiennej `LightAmbient` wpływa na uzyskiwany efekt graficzny

10. Wykonaj eksperyment polegający na zmianie koloru świecenia (zdefiniowane w wektorze `LightColor`. Czy zielony sześcian oświetlony w kolorze niebieskim będzie widoczny na scenie? Co jest przyczyną obserwowanych efektów?

1.2 Modelowanie świateł dyfuzyjnych

Światło rozpraszane (dyfuzyjne) posiada określony kierunek padania i odbija się od powierzchni ulegając rozproszeniu. Model światła dyfuzyjnego pozwala na uzyskiwanie powierzchni matowych, które rozjaśniają się, gdy są zwrócone bezpośrednio w stronę, z której padają promienie światła. Takie podejście wymaga uwzględnienia sposobu zorientowania powierzchni obiektu odbijającego światło względem promieni padania. Na rys. 2 zaprezentowano ideę modelowania z uwzględnieniem wektora normalnego do powierzchni. W praktyce to podejście jest rozszerzane poprzez definiowanie wektorów normalnych, prostopadłych do każdego z wierzchołka ograniczającego powierzchnię prymitywu. Dzięki temu możliwe jest naliczanie lokalnych gradientów w sposobie pocieniowania poszczególnych wierzchołków, co podnosi realizm uzyskiwanych scen.



Rysunek 2: Idea modelowania światła rozpraszanego

Równanie modelu oświetlenia dla składowej Ambient można zatem rozszerzyć do postaci

$$C_f = C_m C_l (A_l + |A_d|) \tag{3}$$

gdzie składowa dyfuzyjna A_d jest iloczynem skalarnym wektora kierunku padania światła oraz normalnej[1] do oświetlanej powierzchni. Oczywiście nic nie stoi na przeszkodzie, aby powyższy model rozbudować o źródło światła ambient i źródło światła diffuse o różnych barwach świecenia.

$$C_f = C_m (C_l A_l + C_d |A_d|) \tag{4}$$

[1] w tym rozumieniu normalna jest opisana wektorem kierunkowym o normie euklidesowej równej 1.

Równanie ogólne można rozbudowywać o kolejne czynniki związane z kierunkiem świecenia i barwą kolejnych źródeł światła. Obecnie, dla uproszczenia spróbujemy zaimplementować jedno kierunkowe źródło światła, które będzie wnosiło składową dyfuzyjną. Realizacja tego celu wymagać będzie dwóch zasadniczych kroków:

1. zdefiniowanie współrzędnych wektorów normalnych dla każdego z wierzchołków jako kolejnego atrybutu tego wierzchołka
2. wprowadzenie modelu równania do shadera wierzchołków, z uwzględnieniem faktu, że transformacje geometryczne wnoszone przez macierz model-widok mogą powodować denormalizację wektorów normalnych.

1.2.1 Przebieg ćwiczenia

1. Zapoznaj się ze sposobem zaimplementowania metody `SetNormal` w klasie implementującej obiekt `glObject`. Zwróć uwagę na sposób przekazywania atrybutów normalnych do tablic atrybutów wierzchołków
2. Uzupełnij kod metody `PrepareObjects` o definiowanie normalnych do poszczególnych wierzchołków na ścianach bocznych sześcianu
3. Rozbuduj kod shadera wierzchołków do następującej postaci

```
#version 330

uniform vec3 LightColor;
uniform vec3 LightDirection;
uniform float LightAmbient;

in vec4 kolorek;
smooth in vec3 vNormal;

out vec4 outputColor;

void main()
{
	float LightDiffuse = max(0.0, dot(normalize(vNormal), -LightDirection));
	outputColor = kolorek*vec4(LightColor*(LightAmbient+LightDiffuse),1.0);
}
```

Spowoduje to zaimplementowanie równania modelu Ambient+Diffuse. Polecenie `max` wybiera największą spośród zadanych liczb. Jakie jest jego przeznaczenie w kontekście ogólnego równania modelu? Polecenie `dot` implementuje iloczyn skalarny. Czy na podstawie rysunku 2 potrafisz wyjaśnić, dlaczego drugi z operandów jest zadawany ze znakiem minus?

4. Zmodyfikuj kod shadera wierzchołków, aby zapewniał przekazywanie do shadera wierzchołków atrybutów normalnych.

```
#version 330

uniform mat4 projectionMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 normalMatrix;

layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec3 inColor;
```

```
layout (location = 2) in vec3 inNormal;

smooth out vec3 vNormal;
smooth out vec4 kolorek;

void main()
{
       gl_Position = projectionMatrix*modelViewMatrix*vec4(inPosition, 1.0);
       vec4 vRes = normalMatrix*vec4(inNormal, 0.0);
       vNormal = vRes.xyz;
       kolorek = vec4(inColor,1.0);
}
```

Zmienna `vNormal` przekazywana do shadera fragmentów jest budowana z wykorzystaniem dodatkowej macierzy `normalMatrix`. Zadaniem tej macierzy jest zapewnienie prawidłowej transformacji wektorów normalnych przy operacjach rotacji i skalowania. Zauważmy, że nie jest ona potrzebna dla operacji translacji. Jeśli macierz MV zawiera wspomniane operacje transformacji rotacji lub skali ujęte w macierzy MT, wówczas macierz normalizująca NM może być wyrażona poprzez

$$NM = \left[MT^{-1}\right]^T \tag{5}$$

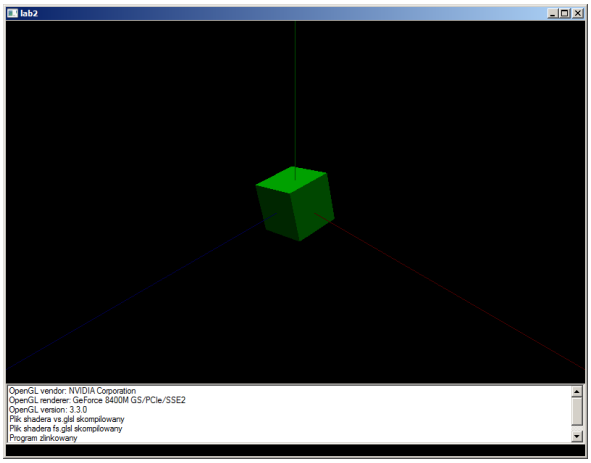
Na bieżącym etapie pozostanie jedynie zaimplementować równanie macierzowego przekształcenia normalnych. Realizacja tego kroku wiąże się z rozbudowaniem kodu rysującego scenę o naliczenie w/w przekształcenia i przekazanie wyników (w postaci macierzy transformacji normalnych) do shadera.

1. Uzupełnij kod rysowania sceny o dyrektywy zapewniające tworzenie i aktualizację macierzy normalizacyjnej

```
void Scene::Draw()
{
// czyscimy bufor kolorow
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

int _ModelView = glGetUniformLocation(program, "modelViewMatrix");
int _Projection = glGetUniformLocation(program, "projectionMatrix");
glUniformMatrix4fv(_Projection, 1, GL_FALSE, glm::value_ptr(mProjection));

glm::mat4 mModelView = glm::lookAt(glm::vec3(5.0f, 5.0f, 5.0f),
                                                    glm::vec3(0.0f),
                                                    glm::vec3(0.0f, 1.0f, 0.0f));
glUniformMatrix4fv(_ModelView, 1, GL_FALSE, glm::value_ptr(mModelView));

glm::vec3 LightDirection = glm::vec3(0.0,-1.0,0.0); // kierunek swiatla
int _LightDirection = glGetUniformLocation(program, "LightDirection");
glUniform3fv(_LightDirection,1,glm::value_ptr(LightDirection));

glm::vec3 LightColor = glm::vec3(1.0,1.0,1.0); // kolor swiatla

int _LightColor = glGetUniformLocation(program, "LightColor");
glUniform3fv(_LightColor,1,glm::value_ptr(LightColor));
```

```
int _LightAmbient = glGetUniformLocation(program, "LightAmbient");
glUniform1f(_LightAmbient,LightAmbient);

int _NormalMatrix = glGetUniformLocation(program, "normalMatrix");

glm::mat4 mTransform = glm::mat4(1.0);
glUniformMatrix4fv(_NormalMatrix, 1, GL_FALSE,
                     glm::value_ptr(glm::transpose(glm::inverse(mTransform))));

Axes->Draw();

mTransform = glm::rotate(glm::mat4(1.0), rot_x, glm::vec3(1.0f, 0.0f, 0.0f));
mTransform = glm::rotate(mTransform, rot_y, glm::vec3(0.0f, 1.0f, 0.0f));

glUniformMatrix4fv(_NormalMatrix,
                             1,
                             GL_FALSE,
                             glm::value_ptr(glm::transpose(glm::inverse(mTransform))));

glUniformMatrix4fv(_ModelView, 1, GL_FALSE, glm::value_ptr(mModelView*mTransform));

Cube->Draw();

}
```

2. Zwróć uwagę na operacje `inverse` oraz `transform` zaimplementowane w GLM.
3. Skompiluj i uruchom program. Sprawdź, czy ściany rotowanej kostki cieniują się prawidłowo.
4. Sprawdź jak zachowa się system cieniowania obiektu z użyciem różnych kierunków padania oświetlenia



Rysunek 3: Przykład cieniowania bryły z użyciem modelu ambient+diffuse

1.3 Analityczne naliczanie wektorów normalnych

Prawidłowe definiowanie wersorów normalnych do powierzchni poddawanych oświetleniu w OpenGL jest niezbędnym warunkiem uzyskania zadowalających efektów cieniowania, które w znaczącym stopniu podnoszą realizm renderowanych scen. Obecnie omówimy ideę naliczania współrzędnych normalnej do powierzchni dowolnie zorientowanej w przestrzeni trójwymiarowej. Każda z takich powierzchni może być jednoznacznie zdefiniowana przez podanie trójki punktów $A = [a_x, a_y, a_z]^T B = [b_x, b_y, b_z]^T C = [c_x, c_y, c_z]^T$ spełniających równanie powierzchni

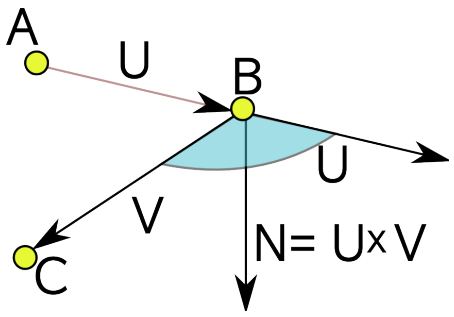
$$\begin{aligned} k_{11}a_x + k_{12}a_y + k_{13}a_z &= 1 \\ k_{21}b_x + k_{22}b_y + k_{23}b_z &= 1 \\ k_{31}c_x + k_{32}c_y + k_{33}c_z &= 1 \end{aligned} \tag{6}$$

gdzie $k..$ są stałymi współczynnikami. Na podstawie tak opisanej trójki punktów możliwe jest znalezienie współrzędnych dwóch wektorów $\vec{U} = [u_x, u_y, u_z]^T$ oraz $\vec{V} = [v_x, v_y, v_z]^T$ zawartych w danej powierzchni:

$$\vec{U} = \begin{bmatrix} a_x - b_x \\ a_y - b_y \\ a_z - b_z \end{bmatrix} \quad \vec{V} = \begin{bmatrix} b_x - c_x \\ b_y - c_y \\ b_z - c_z \end{bmatrix} \tag{7}$$

Ilustrację graficzną tak sformułowanego zagadnienia prezentuje rysunek 4. Naliczenie normalnej $\vec{N}$ do powierzchni zawierającej wektory $\vec{U}$ oraz $\vec{V}$ sprowadza się wówczas do wyznaczenia iloczynu wektorowego

$$\vec{N} = \vec{U} \times \vec{V} \tag{8}$$



Rysunek 4: Układ wektorów pozwalający naliczyć normalną do powierzchni opisanej punktami A,B,C

Współczynniki wektora normalnego można obliczyć na podstawie wyznacznika macierzy

$$\begin{vmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ n_x & n_y & n_z \end{vmatrix} = n_x(u_y v_z - u_z v_y) + n_y(u_z v_x - u_x v_z) + n_z(u_x v_y - u_y v_x) \tag{9}$$

Stąd

$$\vec{N} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix} \tag{10}$$

Powyższe równania zaimplementujemy jako funkcję C++ przystosowaną do przyjmowania punktów A, B, C Wynik działania funkcji jest przechowywany w tablicy opisującej normalną N

```
int CalcNormal(float A[], float B[], float C[], float *N)
{
```

```
 const int x = 0;
 const int y = 1;
 const int z = 2;

 float U[3];
 float V[3];
// oblicz wspolrzedne wektorow U oraz V
 U[x] = A[x] - B[x];
 U[y] = A[y] - B[y];
 U[z] = A[z] - B[z];

 V[x] = B[x] - C[x];
 V[y] = B[y] - C[y];
 V[z] = B[z] - C[z];

// wyznacz wspolrzedne normalnej
 N[x] = U[y]*V[z] - U[z]*V[y];
 N[y] = U[z]*V[x] - U[x]*V[z];
 N[z] = U[x]*V[y] - U[y]*V[x];
return 1;
}
```

1.4 Normalizacja

Wektory normalne o naliczonych współrzędnych, zgodnie z wymogami OpenGL winny mieć normę euklidesową równą 1.0. W ogólności, uzyskanie wektora normalnego w wyniku operacji mnożenia wektorowego wymaga wprowadzenia zabiegu normalizacji, celem ujednolicenia jego długości.

$$\hat{N} = \frac{1}{L} \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} \tag{11}$$

gdzie

$$L = \sqrt{n_x^2 + n_y^2 + n_z^2} \tag{12}$$

Proces naliczania współrzędnych normalnych do powierzchni uzupełniamy zatem o etap skalowania do postaci wersorowej.

```
int Normalize(float *N)
{
 const int x = 0;
 const int y = 1;
 const int z = 2;

// oblicz dlugosc wektora
 float L = (float)sqrt(N[x]*N[x] + N[y]*N[y] + N[z]*N[z]);

 if (L<0.01) L=0.01;
```

```
// wyznacz wspolrzedne normalnej
 N[x] /= L;
 N[y] /= L;
 N[z] /= L;
return 1;
```

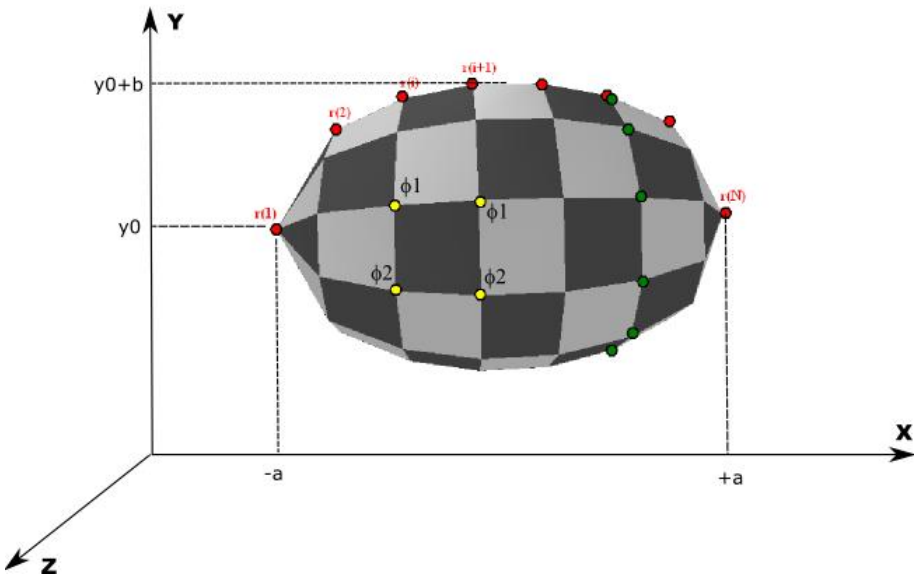
1.5 Budowa własnej bryły

Często zachodzi konieczność wprowadzenia własnego kodu rysującego złożony obiekt przestrzenny. W takich sytuacjach warto zapewnić analityczne naliczanie współrzędnych wierzchołków ograniczających jak również normalnych do powierzchni, co zagwarantuje realistyczne cieniowanie bryły. Obecnie omówimy mechanizm konstrukcji prostej bryły na przykładzie elipsoidy. Rozpatrzmy bryłę zaprezentowaną na rysunku 5. Obiekt składa się z czworokątów rozpiętych na siatce punktów spełniających równanie elipsy (13) względem osi OX, OY oraz równanie okręgu względem osi OY OZ. Wartości promieni okręgów są wyznaczane z równania (14) powstałego po przekształceniu wyjściowego równania elipsy względem zmiennej y. Odległości okręgów wzdłuż osi OX są stałe i mieszczą się w przedziale $< -a, a >$

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{13}$$

Wartość parametru a wpływa na rozpiętość bryły wzdłuż osi OX, natomiast parametr b określa rozpiętość wzdłuż osi OY. W sytuacji gdy $a = b$ elipsoida redukuje się do sfery.

$$r = \sqrt{a^2 - x^2}\frac{b}{a} \tag{14}$$

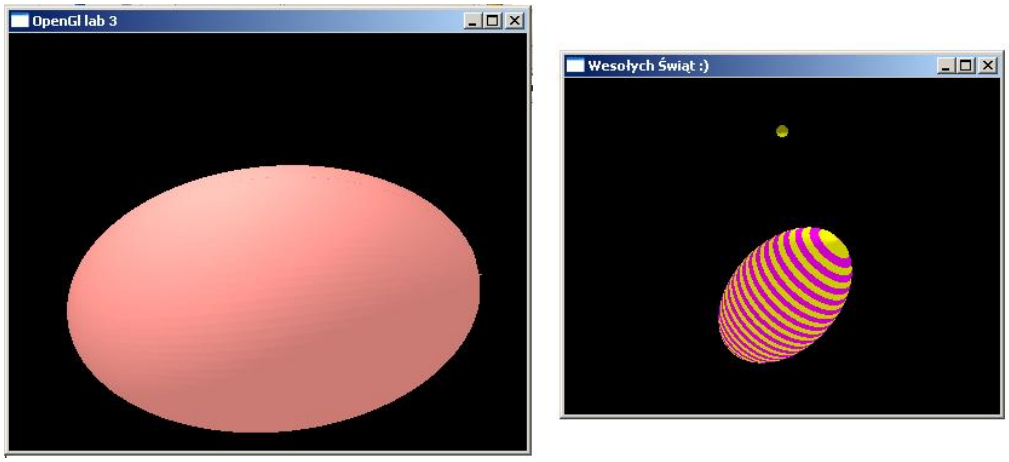


Rysunek 5: Geometria bryły elipsoidalnej

Dyskretna aproksymacja równań wyznaczających siatkę punktów kontrolnych wymaga wprowadzenia przyrostów, o które zmieniają się wartości zmiennej x w kolejnych segmentach jak również przyrostów wartości kąta phi używanego do kreślenia okręgów.

1.5.1 Przebieg ćwiczenia

1. Zapoznaj się z implementacją metody `MakeEgg` w klasie implementującej `glObject`
2. Dodaj do projektu instancję klasy `glObject`. Wywołaj na niej metodę `MakeEgg` w kodzie `PrepareObjects` Jako argumenty wywołania przyjmij np. $a = 1.3, b = 0.7, stacks = 50, slices = 10$
3. Wprowadź do projektu procedury obliczania wektorów normalnych
4. Zapewnij automatyczne obliczenie wartości normalnych dla wierzchołków `v1,v2,v3,v4` wykorzystywanych w metodzie `MakeEgg` do generowania powierzchni elipsoidy
5. Zapewnij wywołanie metody rysującej elipsoidę w kodzie rysowania sceny.
6. Sprawdź jak zmiana parametrów wywołania `MakeEgg` wpływa na efekt wizualny.
7. Zmodyfikuj algorytm rysujący elipsoidę, aby powstała "pisanka". Przykładowe realizacje zaprezentowano na rys. 6



Rysunek 6: Przykład realizacji metody generującej elipsoidę.