UNIWERSYTET
TECHNOLOGICZNO-HUMANISTYCZNY
im. Kazimierza Pułaskiego w Radomiu

WYDZIAŁ TRANSPORTU, ELEKTROTECHNIKI I INFORMATYKI

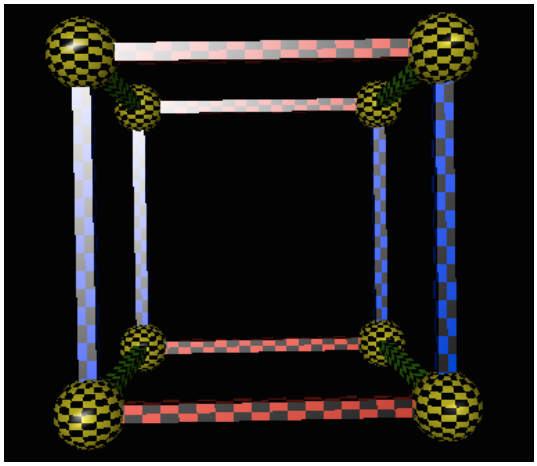
Kierunek: INFORMATYKA

Przedmiot: BIBLIOTEKA GRAFICZNA OPENGL

Autor: DR ARTUR HERMANOWICZ

Transformacje

Dysponując prostymi obiektami – bryłami (jak w naszym przypadku klasy reprezentujące np. kulę czy prostopadłościan) można pokusić się o zbudowanie nieco bardziej złożonych obiektów. Spróbujmy prześledzić to na przykładzie sześcianu zbudowanego „z zapałek" (Rys. 1). Bryła ta składa się z ośmiu sfer umieszczonych w wierzchołkach sześcianu oraz dwunastu prostopadłościanów stanowiących krawędzie.



Rys. 1. Sześcian zbudowany z podstawowych brył (p04.java)

Wcześniej jednak przyjrzyjmy się narzędziom, które umożliwiają manipulacje podstawowymi obiektami. Są to: translacja – przesunięcie, obrót i skalowanie.

Najprostsza jest translacja (metoda glTranslate). Przyjmuje ona trzy parametry oznaczające odpowiednio przesunięcia względem osi X, Y i Z. Zwykle większość tworzonych obiektów ma jest konstruowana względem punktu (0, 0, 0). Jeżeli zachodzi potrzeba umieszczenia obiektu w innym miejscu stosujemy translację. Przykładowo obiekt sfery (klasa Kula) tworzy obiekt, którego środek znajduje się w środku układu współrzędnych i jest wpisany w sześcian o boku 2r. Jeżeli zechcemy aby np. środek naszej kuli znajdował się na wysokości równej jej promieniowi możemy użyć glTranslate:

```
gl.glTranslatef(0.0f, r, 0.0f);
Kula.Draw(gl, r, n, n);
```

W ten sposób skonstruowana zostanie kula o promieniu r, n równoleżnikach i n południkach, a jej środek będzie znajdował się w punkcie o współrzędnych (0, r, 0). Litera f w nazwie metody glTranslatef oznacza, jak zwykle, że parametry są typu float. Jeżeli parametry będą typu double stosujemy glTranslated.

Obrót realizujemy przy pomocy metody glRotate. Przyjmuje ona cztery parametry. Pierwszym z nich jest kąt obrotu (w stopniach). Pozostałe trzy to współrzędne wersora osi obrotu, np.:

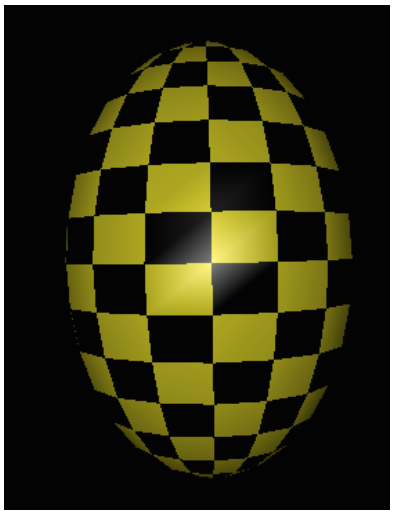
```
gl.glRotatef(180.0f, 1.0f, 0.0f, 0.0f);
Kula.Draw(gl, r, n, n);
```

spowoduje, że nasz obiekt (Kula) zostanie narysowany „do góry nogami" (obrót o 180^0 względem osi X). Oczywiście w przypadku tego obiektu nie będzie żadnej widocznej różnicy.

Ostatnią transformacją jest skalowanie (metoda glScale). Przyjmuje ona trzy parametry oznaczające współczynniki powiększenia względem każdej z osi. Chcąc np. uzyskać elipsoidę (Rys. 2) możemy użyć instrukcji:

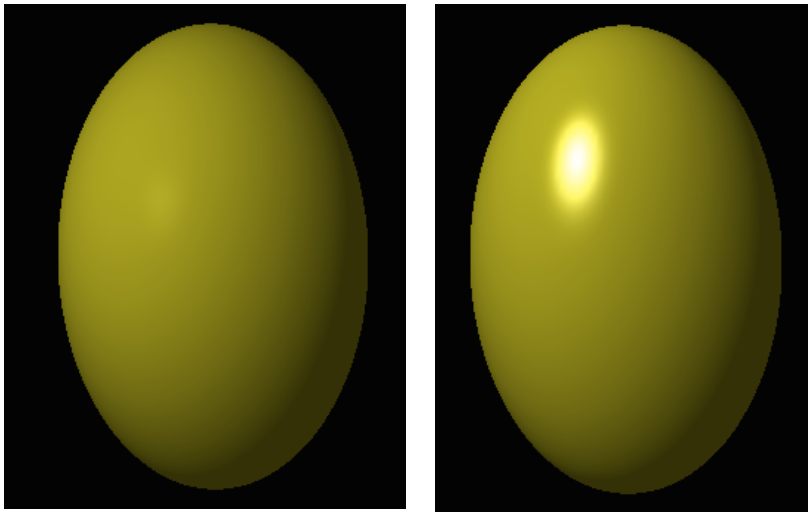
```
gl.glScalef(1.0f, 1.5f, 1.0f);
Kula.Draw(gl, r, n, n);
```

w wyniku czego średnica kuli względem osi Y zostanie powiększona o 50%. Parametr o wartości z przedziału (0, 1) oznacza pomniejszenie, np. 0.5 to dwukrotne zmniejszenie w stosunku do oryginału.



Rys. 2. Elipsoida uzyskana przez skalowanie (p05.java)

W przeciwieństwie do przesunięcia i obrotu, skalowanie modyfikuje coś jeszcze. Otóż skalowanie skaluje wszystko, wliczając w to wektory normalne. Należy o tym pamiętać, gdyż prowadzi do nieoczekiwanych skutków (Rys. 3). Wektor normalny wykorzystywany jest do obliczenia natężenia oświetlenia. Powinien być znormalizowany (mieć długość jeden). Po skalowaniu, uprzednio prawidłowy wektor już nie jest znormalizowany – konieczne jest włącznie automatycznej normalizacji wektorów normalnych (ostatnia instrukcja w metodzie Init: gl.glEnable(GL2.GL_NORMALIZE)).



Rys. 3. Błędne (z lewej) i prawidłowe (z prawej) oświetlenie skalowanego obiektu

Automatyczna normalizacja wektorów normalnych nie jest korzystna dla wydajności programu. Zauważmy, że wektorów normalnych jest tyle ile wierzchołków. Normalizacja polega na obliczeniu długości wektora i podzieleniu przez nią wszystkich składowych wektora, czyli co najmniej jeden pierwiastek i trzy dzielenia (i oczywiście trzy razy kwadrat). Obiekty należy konstruować tak, aby już miały znormalizowane wektory normalne, wtedy automatyczna normalizacja nie jest potrzebna. Kula jest tak skonstruowana (znormalizowane wektory normalne). Jednak wykorzystanie skalowania psuje tę zależność. Wtedy przydatne jest automatyczne normalizowanie jednak kosztem wydajności.

Eksperyment można przeprowadzić samodzielnie. Obiekty na rysunkach 2 i 3 to w praktyce ta sama kula (p04.java) i tak samo przeskalowana. Szachownicę można łatwo „zdjąć" – wystarczy objąć komentarzem lub usunąć wiersz:

```
if((i+j)%2==0) gl.glColor3f(0.0f,0.0f,0.0f);
```

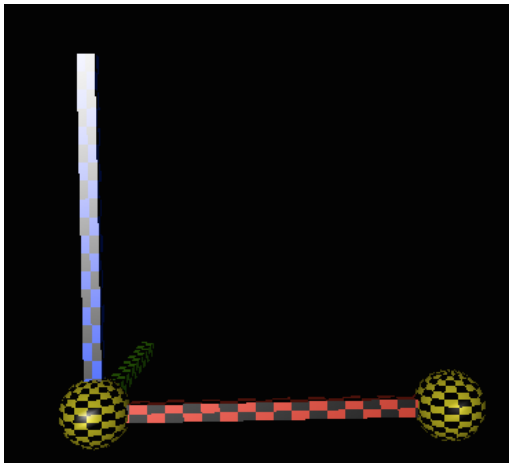
w klasie Kula (Kula.java) – to samo dotyczy pozostałych brył. Zwiększona została liczba przekrojów aby lepiej był widoczny refleks świetlny (Rys. 2 – 16 przekrojów, Rys. 3 – 200 przekrojów). Dodatkowo zostało przesunięte źródło światła GL_LIGHT0 z tego samego powodu.

Wróćmy do naszego sześcianu (Rys. 1). Korzystając z powyższych transformacji można już uzyskać taki efekt tworząc każdy element oddzielnie, przesuwając i obracając tak, aby trafił na właściwe miejsce. Przy bardziej złożonych konstrukcjach będzie to jednak praca żmudna i mało ciekawa. Zwróćmy uwagę, że w tworzonym obiekcie wiele elementów się powtarza, a to oznacza, że można je wykonać korzystając z pętli.

Jeżeli zajrzymy do metody Display (p04.java) to łatwo można odszukać konstrukcję, która powoduje utworzenie sześcianu:

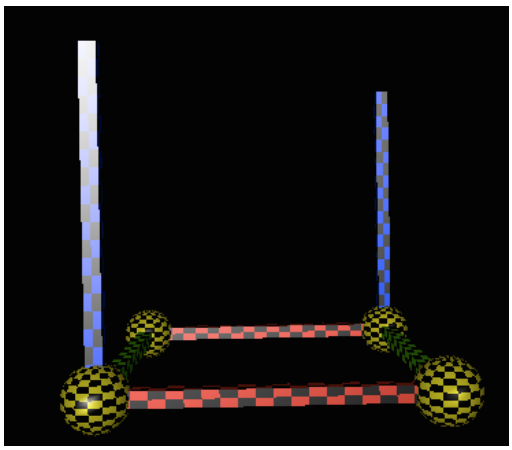
```
for (int j = 0; j < 2; j++) {
     for (int i = 0; i < 2; i++) {
          …
          gl.glRotatef(180.0f, 0.0f, 1.0f, 0.0f);
     }
     gl.glRotatef(180.0f, 0.0f, 0.0f, 1.0f);
}
```

Jest to pętla podwójna, która przy pomocy czterokrotnie powtarzającego się obiektu (Rys. 4), odpowiednio go powtarzając pozwala uzyskać całość obiektu (Rys. 1)



Rys. 4. Powtarzający się element w sześcianie

Zadaniem pętli wewnętrznej jest obrót tego obiektu (w powyższym kodzie – trójkropek) o 180^0 względem osi Y, co po dodaniu do istniejącego już fragmentu stworzy podstawę (Rys. 5).



Rys. 5. Podstawa sześcianu

Zadaniem pętli zewnętrznej jest obrót wykonanego dotychczas obiektu (Rys. 5) o 180^0 względem osi Z, dzięki czemu uzyskamy górę sześcianu i ostatecznie po dodaniu gotowy obiekt (Rys. 1).

Do rozwiązania pozostaje problem skonstruowania obiektu (Rys. 4), przy pomocy którego pętla podwójna tworzy całość. Aby powyższe obroty odniosły właściwy skutek całość bryły została skonstruowana tak, aby środek całego sześcianu znajdował się w punkcie (0, 0, 0). Pamiętać należy, że wszystkie transformacje mają charakter multiplikatywny („nakładają się na siebie"). Jeżeli wykonamy przesunięcie np. o 10 jednostek w górę, narysujemy kulę i chcemy narysować jeszcze jedną 10 jednostek nad poprzednią to wykonujemy przesunięcie o jeszcze 10 jednostek, a nie o 20.

Często składanie transformacji jest uciążliwe lub niepożądane. Pomocne są wtedy instrukcje glPushMatrix i glPopMatrix działające na stosie macierzy. W tym przykładzie są one wykorzystywane. Każdy z „atomów" obiektu (Rys. 4) jest ujęty w parę „nawiasów" składających się z wywołań tych metod.

Pierwsza rysowana jest kula w lewym dolnym rogu:

```
gl.glPushMatrix();
gl.glColor3f(1.0f, 1.0f, 0.0f);
gl.glTranslatef(-1.0f, -1.0f, 1.0f);
```

```
Kula.Draw(gl, r, n, m);
gl.glPopMatrix();
```

Dzięki użyciu glPushMatrix zapamiętujemy na stosie macierzy przekształceń bieżące ustawienia (czyli w praktyce, w tym przypadku stan domyślny). Metodą glColor nadajemy kolor wykorzystany dla tworzonego obiektu (tu: żółty). Metoda glTranslatef przenosi środek układu współrzędnych w lewy dolny róg tworzonego sześcianu. Teraz wystarczy wywołać metodę Draw kuli. Zamykająca kod instrukcja glPopMatrix powoduje, że zostaje odtworzony stan macierzy przekształceń sprzed użycia glTranslate. Dzięki temu punkt (0, 0, 0) znów znajduje się w środku ekranu i konstruowanego sześcianu.

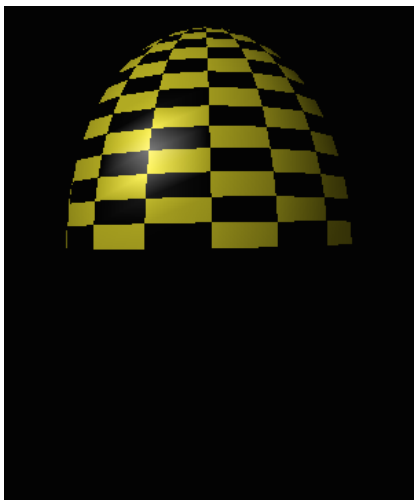
Oczywiście w tym prostym przykładzie wystarczyłoby użyć:

```
gl.glTranslatef(1.0f, 1.0f, -1.0f);
```

i uzyskalibyśmy ten sam efekt. W bardziej złożonych przypadkach wygodniejsze i prostsze jest skorzystanie z pary glPushMatrix i glPopMatrix.

Kolejne zestawy instrukcji w analogiczny sposób prowadzą do uzyskania pozostałych fragmentów konstruowanego obiektu.

Wróćmy jeszcze do przykładu ze skalowaniem kuli (Rys. 2 – p05.java). Powstała w wyniku skalowania elipsoida mogłaby być ciekawsza gdyby miała różne promienie krzywizny. Spróbujemy tego dokonać w możliwie najprostszy sposób. Połączymy dwie półkule o różnych współczynnikach skalowania. Górną część (Rys. 6 – p06a.java) zostawmy jak w poprzednim przykładzie (p05.java), dół zmienimy.



Rys. 6. Półkula

Korekta kodu dla górnej części jest nieznaczna – wystarczy zmienić klasę Kula na Polkula:

```
gl.glPushMatrix();
gl.glColor3f(1.0f, 1.0f, 0.0f);
gl.glScalef(1.0f, 1.5f, 1.0f);
Polkula.Draw(gl, r, n, n);
gl.glPopMatrix();
```

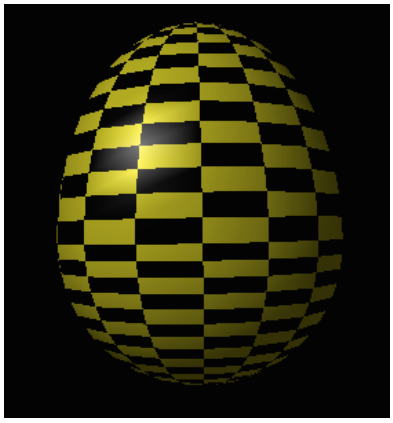
Ze względu na to, że będziemy łączyć różne elementy obiektu dodatkowo zostały dołożone instrukcje glPushMatrix i glPopMatrix. Dzięki temu skalowanie tej części nie będzie miało wpływu na kolejne obiekty.

Dolną część uzyskamy analogicznie klasą Polkula, ale tym razem bez skalowania. Dzięki temu górna część będzie nieco bardziej wydłużona, dolna – zaokrąglona. W efekcie

całość nieco bardziej będzie przypominać kształt jajka (Rys. 7). Dokładamy kilka linijek kodu do metody Display (p06b.java):

```
gl.glPushMatrix();
gl.glColor3f(1.0f, 1.0f, 0.0f);
gl.glRotatef(180.0f, 1.0f, 0.0f, 0.0f);
Polkula.Draw(gl, r, n, n);
gl.glPopMatrix();
```

Tym razem zamiast glScale została użyta transformacja glRotate. Bez obrotu druga półkula zostałaby również narysowana w górę od ośrodka układu współrzędnych przez co byłaby niewidoczna (całkowicie mieściłaby się wewnątrz pierwszej półkuli.

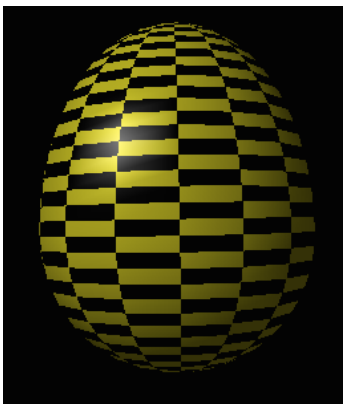


Rys. 7. Elipsoida o różnych promieniach krzywizny

Po złączeniu obu fragmentów bryły bardziej widoczne są dysproporcje między „kwadratami" szachownicy. Pierwszy widoczny efekt polega na tym, że paski w górnej części są wyższe niż w dolnej. Jest to oczywiście wynikiem skalowania. Ta część jest o połowę wyższa niż dolna. Przy tej samej liczbie przekrojów owocuje to ich większą wysokością. Łatwo dokonać korekty. Liczba przekrojów w górnej części powinna być o 50% większa niż w dolnej aby wysokości były takie same. Wystarczy zmodyfikować instrukcję (p06c.java):

```
Polkula.Draw(gl, r, n, n*3/2);
```

dla pierwszej półkuli. W efekcie (Rys. 8) uwidacznia się bardziej drugi problem z szachownicą: „kwadraty" mają znacznie większą szerokość w stosunku do wysokości.



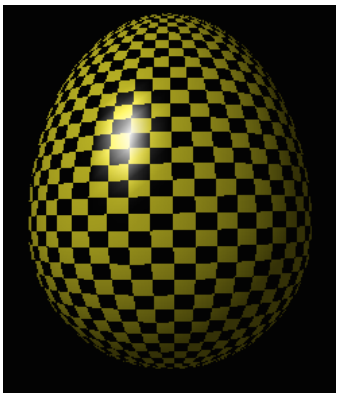
Rys. 8. Bryła po korekcie wysokości górnych przekrojów

Problem wynika z tego, że zarówno liczba „południków" jak i „równoleżników" jest taka sama. Niekorzystny efekt jest tym bardziej widoczny im bliżej „równika" go obserwujemy. Obwód „równika" wyrażony jest wzorem 2*π*r, wysokość okręgu to 2*r

(wprowadziliśmy już korektę wynikającą ze skalowania). Wynika z tego, że „południków” powinno być π razy więcej niż „równoleżników”. Zaokrąglimy to do 3 i zmieniamy odpowiednie liczby przekrojów w kodzie (p06d.java):

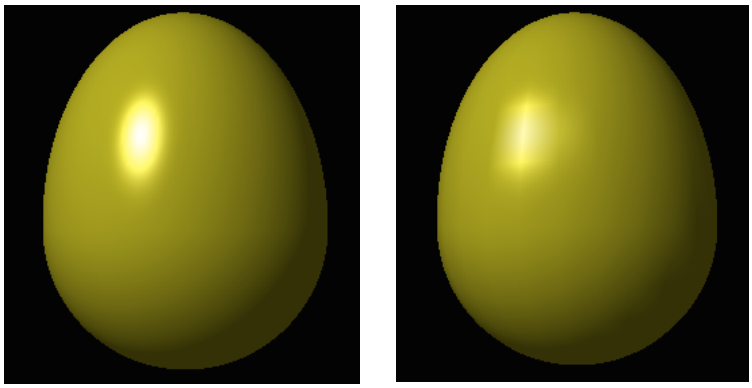
```
gl.glPushMatrix();
gl.glColor3f(1.0f, 1.0f, 0.0f);
gl.glScalef(1.0f, 1.5f, 1.0f);
Polkula.Draw(gl, r, n*3, n*3/2);
gl.glPopMatrix();

gl.glPushMatrix();
gl.glColor3f(1.0f, 1.0f, 0.0f);
gl.glRotatef(180.0f, 1.0f, 0.0f, 0.0f);
Polkula.Draw(gl, r, n*3, n);
gl.glPopMatrix();
```

W wyniku tej korekty pola szachownicy są bardziej kwadratowe (Rys. 9). Działanie to bynajmniej nie ma na celu tylko i wyłącznie uczynienia szachownicy bardziej doskonałą. Pamiętajmy, że oświetlenie jest liczone dla wierzchołków. Figury o podobnych odległościach między wierzchołkami będą lepiej oświetlone niż te o dużych różnicach proporcji.



Rys. 9. Bryła po korekcie szerokości przekrojów

Łatwiej to zaobserwować po usunięciu szachownicy – można wtedy skoncentrować się na samym oświetleniu (Rys. 10). Refleks świetny w wersji z prostokątnymi przekrojami wygląda nieco gorzej niż w przypadku, gdy przekroje mają podobne wymiary.



Rys. 10. Różnica w oświetleniu w zależności od zachowania proporcji między wymiarami przekrojów: proporcje zachowane (z lewej), różne wymiary (z prawej)