1 Przekształcenia geometryczne

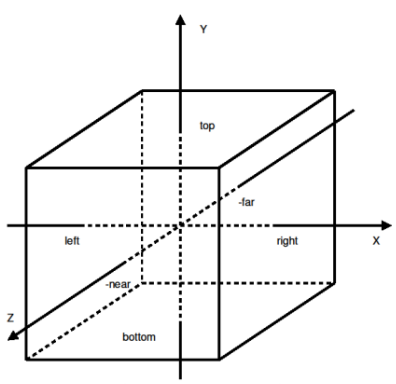
Celem niniejszych ćwiczeń jest omówienie macierzowych przekształceń pozwalających na transformowanie geometrii obiektu wyrażonej przy pomocy zbioru wierzchołków w położenie tego obiektu w wyimaginowanej przestrzeni trójwymiarowej z narzuconym odwzorowaniem widoku (projekcją) na płaszczyznę 2D. Ponadto omówiona zostanie macierz przekształcenia model-widok, która zapewnia możliwość modyfikacji położenia, obrotu, przeskalowania obiektu. Przykłady wykorzystane w ćwiczeniach zostały oparte o klasę glObject, która zawiera omawiane uprzednio techniki proceduralnego tworzenia tablic atrybutów wierzchołków.

1.1 Macierzowe przekształcenie projekcji

W poprzednio realizowanych ćwiczeniach zaprezentowana została idea shaderów jako niedużych programów, które pozwalają w sposób ujednolicony oddziaływać na wszystkie wierzchołki tworzące zbiór obiektów (prymitywów) do narysowania na scenie. Zwróciliśmy uwagę na zmienne jednorodne, które pozwalają na transferowanie wartości z aplikacji głównej do programu shadera. Uzyskanie złudzenia przestrzenności w generowanym widoku wymaga wprowadzenia macierzowego przekształcenia projekcji. Jest ono dane równaniem macierzowym

$$P' = M \cdot P \tag{1}$$

gdzie M jest macierzą rzutowania [1] a P punktem opisanym w przestrzeni. Przekształcenie to znajduje obraz punktu P' po zrzutowaniu na płaszczyznę ekranu. Wyróżniamy twa podstawowe typy rzutowania: ortogonalne oraz perspektywiczne. Rzutowanie ortogonalne (rys. 1) zakłada istnienie prostopadłościanu rozpiętego w kartezjańskim układzie odniesienia XYZ który jest ograniczony płaszczyznami prostopadłymi dla poszczególnych jego osi. Obiekty przestrzenne znajdujące się wewnątrz prostopadłościanu są rzutowane na płaszczyznę ekranu zawierającą się w płaszczyźnie ograniczonej współrzędną *-near*. Rzutowanie ortogonalne (prostokątne) pozwala



(a) Interpretacja graficzna

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

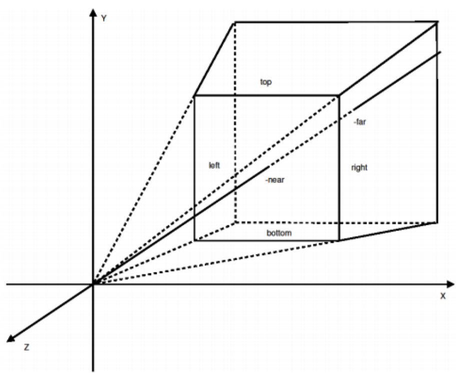
(b) Macierz projekcji ortogonalnej

Rysunek 1: Istota rzutowania ortogonalnego

na realizację prostych wizualizacji stosowanych w niektórych formach rysunku technicznego. Punkty wierzchołkowe znajdujące się w przestrzeni są odwzorowane na rzutni z użyciem linii prostopadłych do płaszczyzny rzutni i przechodzących przez te punkty, co bywa źródłem zniekształceń.

Drugim rodzajem projekcji, które zapewnia o wiele bardziej realistyczne efekty wizualne jest rzutowanie perspektywiczne. Na rysunku 2 zaprezentowano ideę tego rzutowania. Zakłada ona istnienie w przestrzeni graniastosłupa obciętego dwoma płaszczyznami (tzw. frusty [2]). Obiekty o wierzchołkach zawartych w przestrzeni frusty są rzutowane na płaszczyznę.

[1] ang. projection matrix
[2] ang. frustum



(a) Interpretacja graficzna

$$\begin{pmatrix} \frac{\mathrm{ctg}\frac{\phi}{2}}{aspect} & 0 & 0 & 0 \\ 0 & \mathrm{ctg}\frac{\phi}{2} & 0 & 0 \\ 0 & 0 & \frac{far+near}{near-far} & \frac{2\cdot far\cdot near}{near-far} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

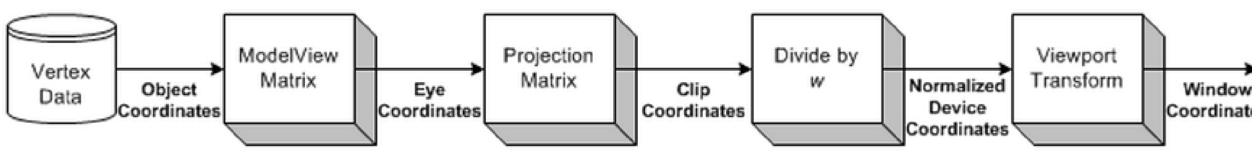
(b) Macierz projekcji perspektywicznej

Rysunek 2: Istota rzutowania ortogonalnego

W rzutowaniu perspektywicznym zakłada się niejednokrotnie, że obserwator jest umieszczony w punkcie (0,0,0) i spogląda na płaski obraz zrzutowany na płaszczyznę *-near*. W takiej interpretacji korzysta się z dwóch dodatkowych parametrów jakimi są kąt rozwarcia ostrosłupa widzenia ϕ oraz współczynnik aspektu [3] rozumiany jako stosunek szerokości do wysokości obszaru rzutowania.

1.2 Macierzowe przekształcenie widoku

Macierz projekcji odpowiada za odwzorowanie współrzędnych punktów w przestrzeni trójwymiarowej na płaszczyznę rzutni. Zanim to nastąpi, współrzędne wierzchołków są zwykle transformowane poprzez macierz model-widok [4], które zadaniem jest wykonanie pewnych elementarnych operacji geometrycznych, związanych ze zmianą położenia, skali lub rotacją obiektów. Macierz model-widok może być również wykorzystywana do modyfikacji położenia obserwatora na scenie oraz punktu, na który obserwator spogląda. Zaprezentowany na rysunku



Rysunek 3: Operacje macierzowe wykonywane na atrybutach wierzchołków

diagram przepływu sugeruje, ze operacje macierzowe winny być realizowane w ramach przekształcenia współrzędnych wierzchołków. W praktyce to zagadnienie sprowadza się do realizacji równania

$$[x', y', z', w']^T = MV \cdot MP \cdot [x, y, z, w]^T \tag{2}$$

Gdzie MV oraz MP są odpowiednio macierzami Model-Widok oraz Projekcji. Warto zwrócić uwagę, że współrzędne wierzchołków zawierają dodatkowy parametr w rozstrzyga on, czy dana trójka współrzędnych opisuje punkt w przestrzeni ($w = 1$) czy też kierunek ($w = 0$) jej obecność jest podyktowana między innymi koniecznością jednolitego przekształcania nie tylko punktów opisujących geometrię obiektu ale również wersorów, które

[3]ang. aspect ratio
[4]ang. modelview matrix

są wykorzystywane np. w modelowaniu oświetlenia. Obecnie zaimplementujemy to równanie w shaderze wierzchołków z wykorzystaniem zmiennych jednorodnych. Macierze będziemy tworzyli za pośrednictwem biblioteki glm [5], która implementuje większość znanych także z wcześniejszych wersji OpenGL operacji macierzowych.

1.2.1 Przebieg ćwiczenia

1. Przygotuj nowy projekt programistyczny i podłącz do niego pliki źródłowe zawarte w materiałach do lekcji.
2. Skompiluj i uruchom program. Zwróć uwagę, czy kompilator poprawnie odszukuje pliki z biblioteki glm. W razie potrzeby przekopiuj folder `glm` we właściwą lokalizację względem położenia pozostałych plików źródłowych.
3. Zapoznaj się z implementacją klasy `glObject` zarządzającej tworzeniem i utrzymywaniem obiektów buforów i obiektów
4. W metodzie `PrepareObjects` klasy scena przygotuj instancję obiektu `glObject`, która reprezentować będzie układ współrzędnych

```
        Axes = new glObject();
        Axes->BeginObject(GL_LINES);
        Axes->SetColor(1.0,0.0,0.0); // os X w kolorze czerwonym
        Axes->AddVertex(0.0,0.0,0.0);
        Axes->AddVertex(10.0,0.0,0.0);
        Axes->SetColor(0.0,1.0,0.0); // os Y w kolorze zielonym
        Axes->AddVertex(0.0,0.0,0.0);
        Axes->AddVertex(0.0,10.0,0.0);
        Axes->SetColor(0.0,0.0,1.0); // os Z w kolorze niebieskim
        Axes->AddVertex(0.0,0.0,0.0);
        Axes->AddVertex(0.0,0.0,10.0);
        Axes->EndObject();
```

   Powyższy kod dodaje do obiektu Axes jedną VAO (metoda `BeginObject`), która ma być rysowana przymitywami `GL_LINES`. Następnie do utworzonej tablicy atrybutów dokładane są kolejne wierzchołki opisujące geometrię prymitywów. Polecenie `EndObject()` kończy definiowanie VAO oraz wymusza przygotowanie i wypełnienie skojarzonych z nią VBO.

5. W kodzie rysowania sceny wydaj polecenie `Axes->Draw()`. Skompiluj i uruchom program. Na bieżącym etapie rozwoju powinien on wykreślić układ współrzędnych. Poszczególne osie układu oznaczane są kolorami R,G,B odpowiadającymi osiom X,Y,Z
6. Do aktualnej postaci aplikacji pora dodać macierze występujące w równaniu 2. W tym celu posłużymy się gotowymi obiektami macierzowymi z biblioteki glm, które są kompatybilne z definicjami obiektów macierzy w shaderach.
7. Zmodyfikuj kod shadera wierzchołków do następującej postaci:

```
// definicja wersji
#version 330
uniform mat4 projectionMatrix;
uniform mat4 modelViewMatrix;
```

[5] ang. Graphics Library Mathematics

```
// zmienna wejsciowa inPosition zbierana z VAO jako atrybut 0
layout (location = 0) in vec3 inPosition;
// zmienna wejsciowa inColor zbierania z VAO jako atrybut 1
layout (location = 1) in vec3 inColor;
// zmienna wyjsciowa do shadera fragmentow
out vec3 kolorek;
void main()
{
	// przypisz pozycje do zmiennej wbudowanej OpenGL
	gl_Position = projectionMatrix*modelViewMatrix*vec4(inPosition, 1.0);
	// przepisz kolor na wyjscie z shadera
	kolorek = inColor;
}
```

Zwróć uwagę na deklaracje macierzy Projekcji oraz Model-Widok zrealizowane w postaci zmiennych jednorodnych.

8. Wprowadź zmienne reprezentujące macierze Model-Widok oraz Projekcji jako składowe prywatne klasy scena. Mechanizm przekazywania zmiennych jednorodnych umożliwi wówczas na przekazywanie obu macierzy pomiędzy aplikacją a shaderem.

```
private:
glm::mat4 mProjection;
glm::mat4 mModelView;
```

9. W kodzie metody odpowiedzialnej za przeskalowanie widoku sceny wprowadź polecenie aktualizacji macierzy Projekcji

```
void Scene::Resize(int new_width, int new_height)
{
	// przypisz nowe gabaryty do pol klasy
	width = new_width;
	// uwzgledniaj obecnosc kontrolki wizualnej
	height = new_height-100;
	// rozszerz obszar renderowania do obszaru o wymiarach 'width' x 'height'
	glViewport(0, 100, width, height);

	mProjection = glm::perspective(45.0f,(GLfloat)width/(GLfloat)height,0.1f,100.0f);
}
```

Polecenie `glm::perspective` definiuje macierz przekształcenia perspektywicznego z uwzględnieniem kolejno: Φ,*aspect*, *near* oraz *far*.

10. Rozbuduj kod rysowania sceny do następującej postaci

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

	int iModelViewLoc = glGetUniformLocation(program, "modelViewMatrix");
```

```
        int iProjectionLoc = glGetUniformLocation(program, "projectionMatrix");
        glUniformMatrix4fv(iProjectionLoc, 1, GL_FALSE, glm::value_ptr(mProjection));


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

        Axes->Draw();
```

Kod zapewnia przekazanie do shadera aktualnie wypracowanych postaci macierzy MP oraz MV. Procedura lookAt tworzy postać macierzy Model-Widok uwzględniającą przemieszczeni obserwatora. Trzy pierwsze argumenty polecenia określają położenie obserwatora, trzy kolejne definiują punkt, na który obserwator spogląda, a trzy ostatnie definiują wektor opisujący gdzie jest "góra"[6] obrazu względem zdefiniowanego układu współrzędnych.

11. Skompiluj i uruchom program. Na bieżącym etapie rozwoju shader wierzchołków realizuje macierzowe przekształcenia widoku z uwzględnieniem rzutowania perspektywicznego i ustawienia obserwatora.

1.3 Testowanie głębi

Procedura rzutowania działa bezkrytycznie na wszystkie wierzchołki zawarte w przestrzeni frusty. Gdyby ograniczyć się tylko i wyłącznie do operacji rzutowania powstałyby przekłamania związane z faktem iż w przestrzeni pewne obiekty mogą przesłaniać inne. W związku z tym, dla dalszego podniesienia realizmu w renderowanych scenach OpenGL został wyposażony w mechanizm automatycznej analizy przesłonięć, która przy realizacji rzutowania bierze pod uwagę głębokość umieszczenia obiektów Informacje tym są zapisywane w buforze głębi. Obecnie przećwiczymy wykorzystanie analizy przesłonięć w przypadku, gdy na scenie znajdą się dwa obiekty opisane odrębnymi VAO.

1.3.1 Przebieg ćwiczenia

1. Zdefiniuj w projekcie instancję dla obiektu `Cube` w sposób analogiczny, jak wprowadzono do niego obiekt `Axes`

2. W kodzie metody `PrepareObject` zapewnij wprowadzenie do obiektu wierzchołków opisujących dwie ściany sześcianu o boku 1, w którego środku znajduje się początek układu współrzędnych.

```
// sciany prostopadle do OX
        Cube->SetColor(0.5,0.0,0.0);
        Cube->BeginObject(GL_TRIANGLE_STRIP);
        Cube->AddVertex(0.5,0.5,0.5);
        Cube->AddVertex(0.5,-0.5,0.5);
        Cube->AddVertex(0.5,0.5,-0.5);
        Cube->AddVertex(0.5,-0.5,-0.5);
        Cube->EndObject();

        Cube->SetColor(0.3,0.0,0.0);
        Cube->BeginObject(GL_TRIANGLE_STRIP);
        Cube->AddVertex(-0.5,0.5,0.5);
```

[6] ang. up-vector

```
        Cube->AddVertex(-0.5,-0.5,0.5);
        Cube->AddVertex(-0.5,0.5,-0.5);
        Cube->AddVertex(-0.5,-0.5,-0.5);
        Cube->EndObject();
```

3. Zapewnij wyrysowanie obiektu `Cube` w kodzie rysowania sceny. Skompiluj i uruchom program, czy ściany obiektu pojawiają się na scenie zgodnie z oczekiwaniami? Porównaj położenie ścian z punktem przecięcia się osi układu współrzędnych

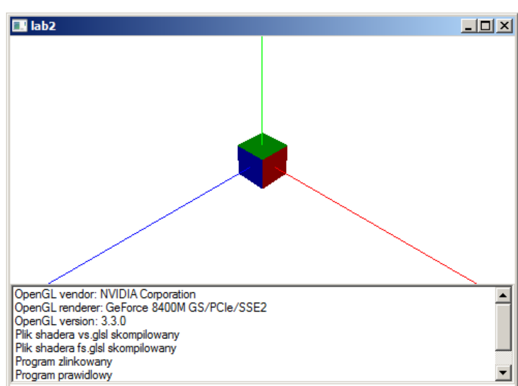
4. W metodzie inicjalizacji sceny wprowadź polecenie testowania głębi

```
// inicjuje proces renderowania OpenGL
void Scene::Init()
{
        //
    ...
        //
        PrepareObjects();

        glEnable(GL_DEPTH_TEST);
        glClearDepth(1.0);
}
```

W kodzie rysowania sceny uzupełnij polecenie `glClear` o flagę czyszczenia zawartości bufora głębi. Zapewni to prawidłową analizę przesłonięć przy każdorazowym wygenerowaniu nowego widoku sceny.

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

5. Uzupełnij definicję ścian w obiekcie `Cube` w ramach metody `PrepareObjects` tak, aby powstał kompletny sześcian. Niech każda ze ścian prostopadła do danej osi układu współrzędnych ma kolor tej osi pociemniony do 0.5 oraz 0.3 jak w przykładzie ze ścianami dla OX. Skompiluj i uruchom program. Widok wyrenderowanego sześcianu prezentuje rys. 4



Rysunek 4: Sześcian złożony z 6-u przystających czworokątów uzyskanych z prymitywu triangle strip.

1.4 Macierzowe przekształcenie geometryczne

Współrzędne punktu P, którego położenie w wybranym układzie współrzędnych W opisuje wektor $[P_x, P_y, P_z]^T$ można przekształcić do opisu w innym układzie współrzędnych U za pomocą równań macierzowo-wektorowych definiujących rotacje, translacje oraz skalę.

Translacją U^T punktu P o wektor V nazywamy przekształcenie opisane równaniem (3)

$$U^T = \begin{bmatrix} 1 & 0 & 0 & V_x \\ 0 & 1 & 0 & V_y \\ 0 & 0 & 1 & V_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \tag{3}$$

1.4.1 Ćwiczenia

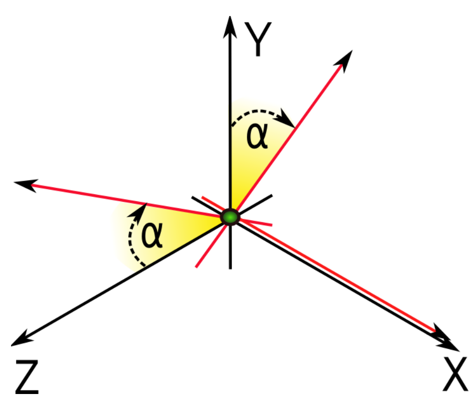
Znajdź obraz punktu P w układzie U jeśli punkt ten ma w układzie W współrzędne $(1.0, 2.0, 0.0)$ a przekształcenie geometryczne $U \rightarrow W$ jest translacją o wektor $\overrightarrow{V} = [2.0, 2.0, 0.0]^T$. Porównaj wynik z rysunkiem z punktu pierwszego niniejszego opracowania.

Skalą U^S punktu P o współczynniki S nazywamy przekształcenie opisane równaniem (4)

$$U^S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \tag{4}$$

Rotacją $U^{Rot(X,\alpha)}$ punktu P o kąt α względem osi OX nazywamy przekształcenie opisane równaniem (5).

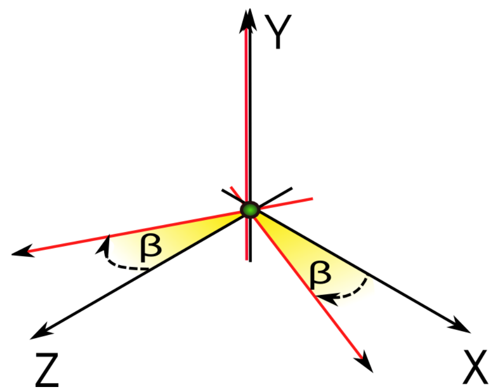
$$U^{Rot(X,\alpha)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha) & -sin(\alpha) & 0 \\ 0 & sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \tag{5}$$



Rysunek 5: Rotacja względem osi OX

Rotacją $U^{Rot(Y,\beta)}$ punktu P o kąt β względem osi OY nazywamy przekształcenie opisane równaniem (6).

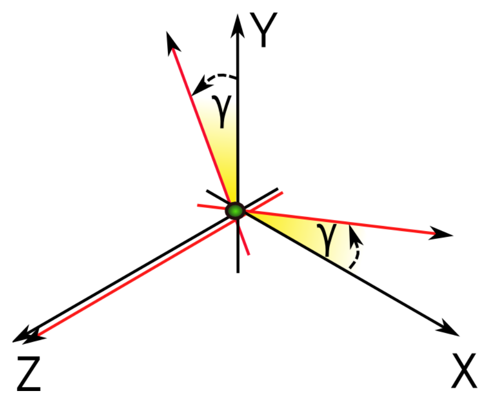
$$U^{Rot(Y,\beta)} = \begin{bmatrix} cos(\beta) & 0 & sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\beta) & 0 & cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \tag{6}$$



Rysunek 6: Rotacja względem osi OY

Rotacją $U^{Rot(Z,\gamma)}$ punktu P o kąt γ względem osi OZ nazywamy przekształcenie opisane równaniem (7).

$$U^{Rot(Z,\gamma)} = \begin{bmatrix} cos(\gamma) & -sin(\gamma) & 0 & 0 \\ sin(\gamma) & cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \tag{7}$$



Rysunek 7: Rotacja względem osi OZ

Powyższe transformacje można ze sobą składać przez wzajemne mnożenie macierzy przekształcenia geometrycznego. Wynikowa macierz przekształcenia geometrycznego jest niczym innym jak macierzą Model-Widok.

1.4.2 Przebieg ćwiczenia

1. Zapoznaj się z ze składnią funkcji `glm::translate` `glm::rotate` oraz `glm::scale` które definiują macierze odpowiednich przekształceń geometrycznych.

2. Zmodyfikuj kod rysujący scenę do następującej postaci:

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

```
        int iModelViewLoc = glGetUniformLocation(program, "modelViewMatrix");
        int iProjectionLoc = glGetUniformLocation(program, "projectionMatrix");
        glUniformMatrix4fv(iProjectionLoc, 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(iModelViewLoc, 1, GL_FALSE, glm::value_ptr(mModelView));

        Axes->Draw();


        mModelView = glm::rotate(mModelView,
                                            rot_y,
                                            glm::vec3(0.0f, 1.0f, 0.0f));

        mModelView = glm::rotate(mModelView,
                                            rot_x,
                                            glm::vec3(1.0f, 0.0f, 0.0f));

        glUniformMatrix4fv(iModelViewLoc, 1, GL_FALSE, glm::value_ptr(mModelView));

        Cube->Draw();
}
```

3. Zwróć uwagę na obsługę zmiennych `rot_x` oraz `rot_y` w metodzie `KeyPressed` klasy implementującej scenę.

4. Skompiluj i uruchom program. Zaobserwuj jak przy pomocy klawiszy strzałkowych klawiatury można obracać sześcian na skutek zmiany macierzy Model-Widok

5. Przesuń kod rysowania osi pod kod rysowania sześcianu. Jaki teraz efekt można zaobserwować? Jaka jest cecha przetwarzania widoku sceny z użyciem macierzy ModelWidok?

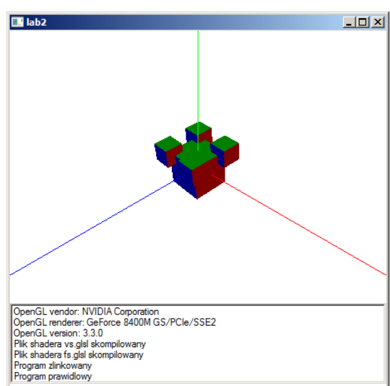
6. Rozbuduj kod rysowania sceny o polecenie translacji oraz skalowania.

```
        glm::mat4 mModelView = glm::lookAt( ... )
        glUniformMatrix4fv(iModelViewLoc, 1, GL_FALSE, glm::value_ptr(mModelView));

        Axes->Draw();

        mModelView = glm::rotate(mModelView,
                                            rot_y,
                                            glm::vec3(0.0f, 1.0f, 0.0f));

        mModelView = glm::rotate(mModelView,
                                            rot_x,
                                            glm::vec3(1.0f, 0.0f, 0.0f));
```

```
        glUniformMatrix4fv(iModelViewLoc, 1, GL_FALSE, glm::value_ptr(mModelView));

        Cube->Draw();

        mModelView = glm::translate(mModelView,
                                                   glm::vec3(0.5,0.5,0.5));

        mModelView = glm::scale(mModelView,
                                                   glm::vec3(0.5,0.5,0.5));

        glUniformMatrix4fv(iModelViewLoc, 1, GL_FALSE, glm::value_ptr(mModelView));

        Cube->Draw();
```

Powyższy kod powoduje wprowadzenie translacji układu odniesienia o wektor $[0.5, 0.5, 0.5]$ oraz skali proporcjonalnie względem każdej osi o 0.5. Po wprowadzeniu tych przekształceń rysowanie kostki jest powielone. Skompiluj i uruchom program. Porównaj uzyskiwany efekt graficzny z opisem w kodzie

7. Zmodyfikuj kod tak, aby zapewnić narysowanie 4 mniejszych sześcianów w narożach dużego. Przykład realizacji zadania prezentuje rys. 8

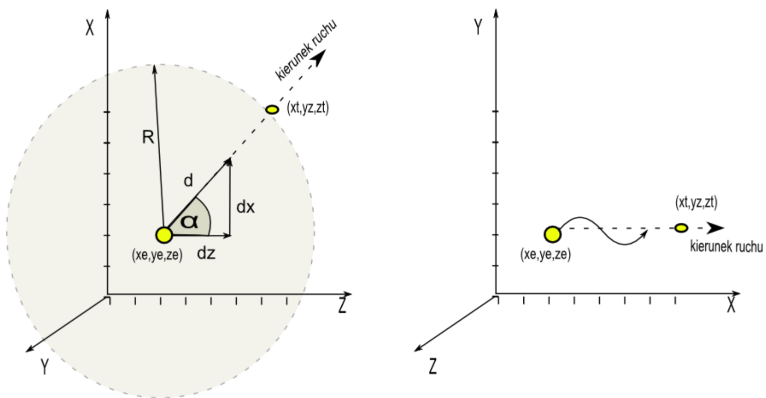


Rysunek 8: Sześcian główny i cztery mniejsze narysowane z użyciem przekształceń geometrycznych

1.5 Poruszanie się po scenie – „chodzenie sinusoidalne"

Obecnie rozważymy koncepcję przemieszczania obserwatora w przestrzeni sceny, które daje wrażenie spaceru. Rozpatrzmy obserwatora umieszczonego na płaszczyźnie XZ (rys. 9).

Położenie obserwatora na scenie wyznaczają współrzędne (xe, ye, ze), zaś jego orientację, kąt $alfa$. Punkt patrzenia obserwatora (cel) jest opisany względem położenia obserwatora za pomocą promienia R oraz kąta orientacji alfa. Wrażenie „rozglądania się" po scenie można uzyskać zatem zmieniając wartość kąta orientacji przy przyciskaniu klawiszy strzałek lewo-prawo. Zmiana położenia obserwatora w pojedynczym kroku na scenie zachodzi względem wektora przemieszczenia d. Aby wyznaczyć zmianę wartości współrzędnych xe, ze należy wektor przemieszczenia rozłożyć na składowe równoległe do osi OX i OZ korzystając z zależności trygonometrycznych względem kąta orientacji alfa. Naciśnięcie klawiszy strzałek góra-dół powinno zatem powodować zmianę współrzędnych xe oraz ze odpowiednio o dx oraz dz. Wrażenie kołysania horyzontu, związanego ze



Rysunek 9: Ilustracja zależności geometrycznych przy implementowaniu algorytmu chodzenia po scenie

stawianiem kroków na scenie można uzyskać poprzez modyfikację współrzędnych ye,yt modulując ich wartości przy pomocy sinusoidy kąta beta zwiększanego lub zmniejszanego w każdym kroku na scenie.

1.5.1 Ćwiczenia

1. Zapoznaj się z działaniem programu `chodzenie_sinus.exe`.
2. Stwórz podobną aplikację w oparciu o opisaną powyżej koncepcję algorytmu.
3. Do generowania podłoża możesz wykorzystać np. następujący kod

```
Plane->BeginObject(GL_LINES);
for (float p = -100.0; p <=100.0; p=p+5.0)
{
        Plane->AddVertex(-100.0f,0.0f,p);
        Plane->AddVertex(100.0f,0.0f,p);
        Plane->AddVertex(p,0.0f,-100.0);
        Plane->AddVertex(p,0.0f,100.0);
}
Plane->EndObject();
```