Ćwiczenie 4. Przelot nad terenem

Witold Alda

Spis treści

1 Co chcemy zrobić?

Tytuł ćwiczenia jest skrótowy i umowny. Chodzi o zbudowanie nieco bardziej złożonej sceny, po której obserwator może się przemieszczać. Ćwiczenie jest oparte o przykład z tutoriala Tut 07. Z dwóch wersji przykładu, wybieramy ten z pliku `World Scene.cpp` - przynajmniej na początek. Program jest zdecydowanie bardziej rozbudowany od wcześniejszych – zawiera wiele ważniejszych i drobniejszych nowych elementów, o różnym charakterze. Będziemy chcieli zwrócić uwagę na najważniejsze z nich:

- Wprowadzenie pojęcia *przestrzeni świata* (*world space*) z nowym układem współrzędnych (świata). W tym układzie mogą się poruszać różne obiekty, z kolei sam świat jest obserwowany przez poruszającą się kamerą. Dla lepszego uporządkowania transformacji wprowadzamy dodatkową macierz, o czym niżej.
- możliwość nawigowania w przestrzeni świata. Użycie współrzędnych sferycznych do operowania kamerą.

- Obiekty o skomplikowanych kształtach, wczytane jako siatki, z przygotowanych wcześniej plików xml.
- Więcej niż jedna para shaderów.

2 Trzy macierze przekształceń

Zwróćmy uwagę, że w programie są użyte trzy macierze transformacji. Pierwsza z nich, `modelToWorldMatrix` służy do rozmieszczenia poszczególnych obiektów i ich elementów na scenie. Macierz ta zmienia się dla każdego obiektu. Dla ustalenia uwagi: drzewko wczytane z pliku xml ma współrzędne lokujące je na początku układu współrzędnych. `modelToWorldMatrix` przesuwa je na właściwe miejsce.

Drugą macierzą jest `worldToCameraMatrix`. Przedstawia ona współrzędne wszystkich obiektów (czyli współrzędne świata) względem położenia kamery. Macierz ta zmienia się, gdy wymuszamy ruch kamery w trakcie przelotu.

Trzecią macierzą jest `cameraToClipMatrix`, która wykonuje rzutowanie perspektywiczne. Jej wartości zmieniają się tylko gdy zmieniamy rozmiary i/lub proporcje okna na ekranie (u nas robi to funkcja `reshape()`).

3 Nowy mechanizm stosu i rysowanie obiektów

W przykładzie `World Scene` zastosowano inny mechanizm stosu niż w programie `Hierarchy` z poprzedniego tutoriala – choć bardzo zbliżony. Wykorzystano tu bibliotekę GL Util za pakietu "Unofficial OpenGL SDK", a dokładniej klasy `MatrixStack` i obiekt `glutil::PushStack`. W klasie `MatrixStack` mamy też zdefiniowane transformacje. Z kolei destruktor obiektu `PushStack` wykonuje operację Pop. Opis biblioteki można znaleźć w `http://glsdk.sourceforge.net/`

Proszę zwrócić też uwagę, że funkcjach rysowania obiektów, najpierw wykonujemy pewne transformacje (zwykle Scale, Translate), które zmieniają bieżącą macierz transformacji. Następnie tę bieżącą macierz transformacji wyłuskujemy za pomocą metody `Top()` i przesyłamy do shadera jako zmienną Uniform dla danego obiektu. Tak jest np. w funkcji rysowania podstawy "Partenonu":

```
//Draw base.
{
glutil::PushStack push(modelMatrix);

modelMatrix.Scale(glm::vec3(g_fParthenonWidth,
                  g_fParthenonBaseHeight, g_fParthenonLength));
modelMatrix.Translate(glm::vec3(0.0f, 0.5f, 0.0f));
```

```
glUseProgram(UniformColorTint.theProgram);
glUniformMatrix4fv(UniformColorTint.modelToWorldMatrixUnif, 1, GL_FALSE,
                   glm::value_ptr(modelMatrix.Top()));
glUniform4f(UniformColorTint.baseColorUnif, 0.9f, 0.9f, 0.9f, 0.9f);
g_pCubeTintMesh->Render();
glUseProgram(0);
}
```

Samo rysowanie dokonuje się za pomocą klasy `Framework::Mesh`. Z jej pomocą defniujemy obiekty, do których czytamy siatki z plików XML:

```
g_pConeMesh = new Framework::Mesh("UnitConeTint.xml");
g_pCylinderMesh = new Framework::Mesh("UnitCylinderTint.xml");
g_pCubeTintMesh = new Framework::Mesh("UnitCubeTint.xml");
g_pCubeColorMesh = new Framework::Mesh("UnitCubeColor.xml");
g_pPlaneMesh = new Framework::Mesh("UnitPlane.xml");
```

które rysujemy jak w wyżej przytoczonym fragmencie: `g_pCubeTintMesh->Render();`.

4 Operowanie kamerą

Pozostawiając analizę zachowania kamery we współrzędnych sferycznych na wykład, tutaj ograniczymy się do przedstawienia wygodnej i intuicyjnej w użyciu funkcji `CalcLookAtMatrix()`. Nawiązuje ona do funkcji ze starego OpenGLa: gluLookAt();

W obu przypadkach mamy 9 parametrów: pierwsze trzy oznaczją położenie kamery (we współrzędnych świata), droga trójka - punkt, na który patrzy kamera, trzecia trójka - kierunek pionu kamery.

Funkcja pozwala intuicyjny sposób realizować np. przelot nad terenem.

5 Więcej shaderów

W programie wykorzystany jest jeden vertex shader i trzy fragment shadery. W sumie tworzą one trzy pary shaderów.

Proszę zwrócić uwagę, jak użyta jest struktura `ProgramData`, która łączy parę shaderów ze zmiennymi `uniform` dla niej przeznaczonymi.

6 Co do zrobienia na zajęciach?

Na początek należy przeanalizować program. Następnie proszę wykonać zmianę (jedną do wyboru lub więcej) w zachowaniu się obiektów na scenie - w reakcji na sygnał z klawiatury.

1. Na początku scena pozbawiona jest drzew. Sygnał z klawiatury powoduje narysowanie kolejnego drzewa. Łatwo to zrobić, bo wszystkie drzewa i ich parametry są na sztywno umieszczone w tablicy – oryginalnie z modyfikatorem `const`. Czy równie łatwo jest usuwać drzewa? - można to też zrobić.

2. Na sygnał od klawiatury drzewa się przeskalowują - podrastają trochę. Wszystkie jednakowo, albo z wykorzystaniem generatora liczb pseudolosowych.

3. Realizujemy film animowany pt. "Makbet" i przygotowujemy scenę, w której las Birnam maszeruje w stronę zamku

Można również wykonać jakąs inną modyfikację w tym stylu.

Gdyby wystarczyło czasu:

4 Proszę wykonać automatyczny lot nad terenem z ustalonymi parametrami. Na sygnał z klawiatury zaczynamy krążyć np. po okręgu na niewielkiej wysokość