Ćwiczenie 2. Modyfikacja koloru. Ruch obiektu – zmienne `uniform`.

Witold Alda

Spis treści

1 Co chcemy zrobić?

Celem tego ćwiczenia jest zwrócenie uwagi na następujęce elementy programowania w OpenGL:

1. Modyfikowanie *fragment shadera* (przykład z tutoriala Tut 02).
2. Przesyłanie do karty graficznej różnych atrybutów wierzchołków (np. położeń i kolorów) i poprawne ich odczytanie. Przykłady znajdują się w tutorialu Tut 02.
3. Przekazywanie do shaderów zmiennych typu uniform, których zadaniem jest sterowanie wyglądem i renderowanie sceny (np. poruszanie obiektów, zmiana ich koloru, ew. kształtu i rozmiarów). Poświęcone są temu przykłady z tutoriala Tut 03.

2 Krótka informacja o zmiennych wbudowanych.

W shaderach, używanych w przykładach, pojawiają się zmienne rozpoczynające się od przedrostka `gl_`, które nie są jawnie zdefiniowane. Są to zmienne wbudowane GLSL. W najbliższych przykładach spotkamy zmienne: `gl_Position` i `gl_FragCoord`.

`gl_Position` jest zdefiniowana jako `out vec4` i zakłada, że wartości, do niej wpisane, będą potraktowane jako czteroelementowe współrzędne wierzchołka (tzw. współrzędne jednorodne – nie używaliśmy jeszcze tego pojęcia).

Z kolei `gl_FragCoord` jest zmienną wbudowaną *fragment shadera*, zdefiniowaną jako `in vec4`. Zawiera ona współrzędne fragmentów (punktóe) okna wyrażone w jednostkach pikseli. W najbliższych przykładach wykorzystujemy tylko dwa pola: `gl_FragCoord.x` i `gl_FragCoord.y`.

W ostatnich wersjach OpenGL Shading Language zmiennych wbudowanych nie ma wiele (5 w vertex shaderze i 8 we fragment shaderze). Warto zwrócić uwagę, że we wcześniejszych wersjach GLSL było ich znacznie więcej. Więcej informacji na temat zmiennych wbudowanych jest w specyfikacji GLSL, np pod adresem `http://www.opengl.org/documentation/glsl/`.

3 Tutorial 02 – Modyfikowanie kolorów

W tutorialu `Tut 02` znajdują się dwa programy pokazujące jak można ustawiać i zmieniać rozkład kolorów w obrębie trójkąta. Przykłady nie wyczerpują oczywiście ani procenta możliwości, gdyż korzystając z shaderów mamy możliwość praktycznie dowolnego ustawienia wartości każdego piksela.

Pierwszy z przykładów, `FragPosition.cpp`, prawie nie różni się od programu z pierwszego tutoriala. W szczególności przekazuje do vertex shadera tylko położenia wierzchołków. Zmiany w kolorowaniu wnętrza trójkąta są wykonane w całości we fragment shaderze. Wyliczane są dla każdego piksela wyjściowego na podstawie współrzędnych fragmentów zawartych w zmiennej `gl_FragCoord` (dokładnie na podstawie składowej y jako średnia ważona:

```
#version 330

out vec4 outputColor;

void main()
{
    float lerpValue = gl_FragCoord.y / 500.0f;

    outputColor = mix(vec4(1.0f, 1.0f, 1.0f, 1.0f),
```

```
        vec4(0.2f, 0.2f, 0.2f, 1.0f), lerpValue);
}
```

Użyta jest tu funkcja GLSL `mix()`, która wylicza średnią ważoną dwóch pierwszych parametrów, traktując trzeci jako wagę.

Z kolei drugi program tutoriala Tut 02 – `VertexColors.cpp` odwołuje się do kolorów przypisanych wierzchołkom w programie OpenGL i przekazanym razem z nimi do karty graficznej. W przykładzie wierzchołki i kolory zdefiniowane są w jednej tablicy:

```
const float vertexData[] = {
     0.0f,    0.5f, 0.0f, 1.0f,
     0.5f, -0.366f, 0.0f, 1.0f,
    -0.5f, -0.366f, 0.0f, 1.0f,
     1.0f,    0.0f, 0.0f, 1.0f,
     0.0f,    1.0f, 0.0f, 1.0f,
     0.0f,    0.0f, 1.0f, 1.0f,
};
```

W celu przesłania wierzchołków i kolorów, należy zainicjować bufor dla danych:

```
void InitializeVertexBuffer()
{
    glGenBuffers(1, &vertexBufferObject);

    glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

a następnie przekazać dane do karty graficznej jako dwa bloki w funkcji `display()`:

```
void display()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glUseProgram(theProgram);

    glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
```

```
    glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, (void*)48);

    glDrawArrays(GL_TRIANGLES, 0, 3);

    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);
    glUseProgram(0);

    glutSwapBuffers();
    glutPostRedisplay();
}
```

Proszę zwrócić uwagę na dwukrotne wywoływanie funkcji `glEnableVertexAttribArray();`, `glVertexAttribPointer();` i `glDisableVertexAttribArray();` dla obu bloków danych. W funkcji `glVertexAttribPointer();` widzimy też przesunięcie w adresie początku drugiego bloku o 48 bajtów.

Alternatywnie można współrzędne i kolory umieścić w odrębnych tablicach:

```
const float vertexData[] = {
     0.0f,    0.5f, 0.0f, 1.0f,
     0.5f, -0.366f, 0.0f, 1.0f,
    -0.5f, -0.366f, 0.0f, 1.0f,
};
const float colorData[] = {
     1.0f,    0.0f, 0.0f, 1.0f,
     0.0f,    1.0f, 0.0f, 1.0f,
     0.0f,    0.0f, 1.0f, 1.0f,
};
```

Wtedy inicjalizacja dotyczy dwóch buforów:

```
void InitializeBuffers()
{
    GLuint vboObjects[2];
    glGenBuffers(2, vboObjects);

GLuint vertexBufferObject = vboObjects[0];
GLuint colorBufferObject = vboObjects[1];

glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
glBindBuffer(GL_ARRAY_BUFFER, colorBufferObject);
    glBufferData(GL_ARRAY_BUFFER, sizeof(colorData), colorData, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

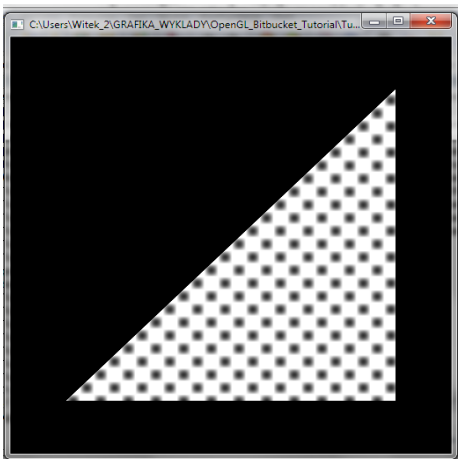
Funkcja `display()` może być w stylu:

```
void display()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glUseProgram(theProgram);

    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, colorBufferObject);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, 0);

    glDrawArrays(GL_TRIANGLES, 0, 3);

    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);
    glUseProgram(0);

    glutSwapBuffers();
    glutPostRedisplay();
}
```

W drugim zapisie nie musimy obliczać przesunięcia, ale chyba jest on mniej elegancki.

3.1 Do zrobienia na ćwiczeniach

Zmień program fragment shadera. Użyj obu składowych x i y zmiennej `gl_FragCoord` oraz funcji okresowej na zmiennej lerpValue, tak aby uzyskać obrazek w stylu, jak poniżej, lub inny - ale ciekawszy.



Ten prosty przykład pozwala nam wyobrazić sobie potencjalne możliwości *fragment shadera* w ustalaniu wartości poszczególnych pikseli.

4 Tutorial 03 – poruszanie obiektami

Tutorial Tut 03 pokazuje cztery przykłady wprowadzenia ruchu obiektu, a przy okazji także innych jego modyfikacji. Dokładny opis przykładów umieszczony jest w trzecim rozdziale podręcznika. Poniżej dodano tylko krótkie uwagi.

W pierwszym programie `cpuPositionOffset.cpp` zmiana położenia dokonuje się w programie OpenGL. W pętli obliczane jest kolejne przesunięcie `fXOffset, fYOffset` (sterowane zmienną czasową GLUT_ELAPSED_TIME), a następnie zmodyfikowane położenia przesyłane są w funkcji `AdjustVertexData()` do karty graficznej. W funkcji `display()` istotne jest wywołanie w ostatnim wierszu `glutPostRedisplay()`, które wymusza w pętli zdarzeń odświeżenie ekranu nową zawartością sceny - bez tego wywołania nie zobaczymy animacji. Rozwiązanie powyższe, choć poprawne, jest jednak nieefektywne, gdyż wymaga przesyłania dla każdej klatki animacji kompletu nowych położeń wszystkich wierzchołków. Dla miliona wierzchołków stanowi to istotne obciążenie!

Lepszym rozwiązaniem jest przesłanie w każdej klatce animacji tylko jednej pary przesunięć, wspólnej w końcu dla wszystkich wierzchołków sceny, a następnie zmodyfikowanie położeń wierzchołków w *vertex shaderze* na podstawie przesłanych wcześniej położeń początkowych i przechowywanych w pamięci karty graficznej. Rozwiązanie to, użyte w programie `vertPositionOffset.cpp`, wymaga przesłania przesunięć jako zmiennych typu `uniform`.

Przygotowanie tej operacji wymaga dwóch kroków:

1. Wskazanie na początku (u nas w funkcji `InitializeProgram()`) powiązania programu OpenGL i *vertex shadera* poprzez zmienną `uniform`. Wywołanie:

   ```
   offsetLocation = glGetUniformLocation(theProgram, "offset");
   ```

   oznacza, że w *shaderze* identyfikowanym przez zmienną `theProgram` pojawi się zmienna o nazwie `offset` z kwalifikatorem `uniform`. Do niej wysyłamy informację o przesunięciu.

2. Wywołanie w funkcji `display()`

   ```
   glUniform2f(offsetLocation, fXOffset, fYOffset);
   ```

   przypisuje konkretne wartości `fXOffset, fYOffset` do wysłania jako `uniform`. Identyfikator `offsetLocation` jest łącznikiem z wywołaniem `glGetUniformLocation()`.

Trzecia wersja zawarta w programie `vertCalcOffset.cpp` przerzuca więcej obliczeń do shaderów. W programie OpenGL pobierana jest jedynie informacja od zegara i współczynnik określający szybkość pętli, a samo wyliczenie przesunięć dokonuje się już na GPU.

W ostatniej wersji, w pliku `fragChangeColor.cpp`, zmienne typu `uniform` służą dodatkowo do zmodyfikowania koloru obiektu w *fragment shaderze*.

4.1 Do zrobienia na ćwiczeniach

Posługując się ostatnią wersją przykładu (`fragChangeColor.cpp`), proszę wykonać następujące modyfikacje:

1. zmianę toru ruchu po okręgu na jakiś inny - elipsę, a jeszcze lepiej jakąś krzywą Lissajou.

2. zmianę ruchu na taki jak na stole bilardowym bez tarcia, z odbiciami od krawędzi. Można przy tym zmienić trójkąt na kwadrat.

3. bardziej nieregularną zmianę koloru - np. wprowadzenie rozbłysków co jakiś czas (może różnokolorowych rozbłysków?).

4. czy dałoby się wprowadzić rozbłyski przy odbiciach od krawędzi?