Witold Alda

Grafika komputerowa. Laboratorium 7. WebGL i shadery.

Najprostsze użycie shaderów.

W tym ćwiczeniu zapoznamy się z prostymi przykładami napisanymi wyłącznie w WebGL, bez wykorzystania biblioteki three.js, za to z wykorzystaniem shaderów kodowanych w GLSL. Przykłady są wzorowane na podręczniku i materiale kursu Ed Angela. Dodatkowe informacje: www.cs.unm.edu/~angel/
Przykłady są bardzo podstawowe i mogą być przeanalizowane błyskawicznie. Z tego też powodu jest ich więcej. Ważne jednak wydaje się prześledzenie działania shaderów na najniższym poziomie.

1. Pierwszy przykład – rysujemy trójkąt

Jest to chyba najbardziej elementarny przykład, jednak pokazujący strukturę typowych programów z shaderami.
Podstawowy plik triangle.html zawiera właściwie wyłącznie źródła shaderów i nazwy importowanych modułów/bibliotek pisanych w JS.

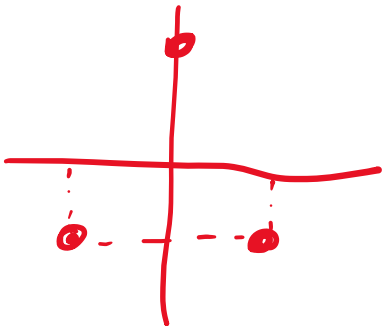
```
<!DOCTYPE html>
<html>
<head>
<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;
void main()
{
    gl_Position = vPosition;
}
</script>
<script id="fragment-shader" type="x-shader/x-fragment">
void main()
{
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
</script>
<script type="text/javascript" src="../webgl-utils.js"></script>
<script type="text/javascript" src="../initShaders.js"></script>
<script type="text/javascript" src="triangle.js"></script>
</head>
<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```

Można zwrócić uwagę, że szczątkowe shadery niewiele robią. Vertex shader akceptuje z głównego programu (w naszym przypadku triangle.js) zmienną `vPosition`, która jest czteroelementowym wektorem zawierającym podstawowy atrybut wierzchołka – współrzędne położenia. Na razie nie wiemy jak ta zmienna została przygotowana w trangle.js.
Shader w przykładzie zawiera tylko jedną funkcję `main();` , a w niej obowiązkowe przypisanie `vPosition` do systemowej zmiennej `gl_Position,` która powinna zawierać położenia gotowe do wyświetlenia na ekranie.
Fragment shader jest jeszcze prostszy, bo nie wymaga żadnych danych wejściowych, a na wyjściu ustawia w zmiennej systemowej `gl_FragColor` kolor czerwony.

Wypada może przypomnieć, że shadery są przygotowywane do obsługi pojedynczego wierzchołka lub piksela, w rzeczywistości jednak wiele instancji shaderów wykonywanych równolegle na procesorze graficznym przetwarza wiele wierzchołków i pikseli.

W dalszej części kodu html, importowane są moduły pomocnicze, które są używane w kursie i podręczniku Angela: **webgl-utils.js** zawiera zestaw funkcji użytkowych (od Google), które pozwalają na budowanie kontekstu WebGL. **initShaders.js** natomiast zawiera sekwencję wywołań funkcji WebGL do czytania, kompilowania i linkowania shaderów.

Organizacja przykładów w tym ćwiczeniu jest taka, że właściwy kod Javascript jest wyprowadzony do oddzielnego pliku, który jest dołączany po wszystkich bibliotekach. W pierwszym przykładzie jest to *triangle.js*.



```
var gl;
var points;
window.onload = function init()
{
    var canvas = document.getElementById( "gl-canvas" );
    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" );
 }
var vertices = new Float32Array([-1, -1, 0, 1, 1, -1]);
 //  Konfigurujemy WebGL
gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 1.0, 1.0, 1.0, 1.0 );
//  Ładujemy shadery za pomocą gotowej funkcji
   var program = initShaders( gl, "vertex-shader", "fragment-shader" );
      gl.useProgram( program );

   // Inicjujemy bufory i ładujemy dane do GPU
   var bufferId = gl.createBuffer();
   gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
   gl.bufferData( gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW );
// Kojarzymy zmienne shadera z danymi bufora i rysujemy
 var vP = gl.getAttribLocation( program, "vPosition" );
 gl.vertexAttribPointer( vP, 2, gl.FLOAT, false, 0, 0 );
 gl.enableVertexAttribArray( vP );
 render();
};
function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.TRIANGLES, 0, 3 );
```

```
}
```

Możemy pominąć początek, który ma za zadanie sprawdzić czy nasz sprzęt obsługuje WebGL, i przejść do kolejnych czterech elementów.

1. Konfigurowanie WebGL. W tym przypadku ogranicza się do otworzenia okna grafiki (`gl.Viewport`), w tym wypadku o rozmiarach takich jak okno canvas (512x512) i pomalowania go jednolitym kolorem tła(`gl.ClearColor`).
2. Ładowanie shaderów, a przy tym ich kompilacja i linkowanie, wszystko za pomocą funkcji `initShaders()`; Bez wątpienia warto popatrzeć do środka kodu `initShaders()`; i zobaczyć kolejne czynności jakie się tam wykonują.
3. Inicjowanie bufora danych (na GPU) i przesłanie do niego atrybutów wierzchołków. W naszym przypadku są to tylko położenia. I tak:
   `var bufferId = gl.createBuffer();` tworzy bufor i nadaje mu identyfikator,
   `gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );` uaktywnia bufor o danym identyfikatorz w trybie tablicy danych
   `gl.bufferData( gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW );` przesyła do aktywnego bufora dane z tablicy `vertices.` Ostatni parametr oznacza przewidywany tryb użycia danych (statyczny, dynamiczny, strumień danych) i ma przede wszystkim znaczenia optymalizacyjne.
4. W następnym kroku należy skojarzyć wysłane do bufora dane z tym co powinien otrzymać shader. Używamy do tego:
   `var vP = gl.getAttribLocation( program, "vPosition" );` który mówi, że zmienna `vP` zapewni, że dane z bufora trafią do shadera `program,` do zmiennej `vPosition.`
   `gl.vertexAttribPointer( vP, 2, gl.FLOAT, false, 0, 0 );` określa w jaki sposób dane z bufora identyfikowanego przez `vP` są dystrybuowane i interpretowane. Kolejne argumenty oznaczają: 2 – liczba wartości w elemencie vPosition, `gl.FLOAT` – typ wartości, `false` – nie normalizujemy danych stałoprzecinkowych, tu akurat bez znaczenia, `0` – odstęp pomiędzy kolejnymi danymi (stride), `0` – offset (czy startujemy od początku bufora)
   Na koniec wywołujemy funkcję `render();`, która rysuje na ekranie zawartość bufora za pomocą funkcji `gl.DrawArray();`. W rysowaniu wezmą udział kolejno wszystkie elementy bufora.
   Rysowanie odbywa się w trybie `gl.drawArrays( gl.TRIANGLES, 0, 3 );`, co oznacza rysowanie pełnych trójkątów, drugi parametr wskazuje od którego elementu tablicy wybieramy wierzchołki, a trzeci ile wierzchołków rysujemy.

Do zrobienia

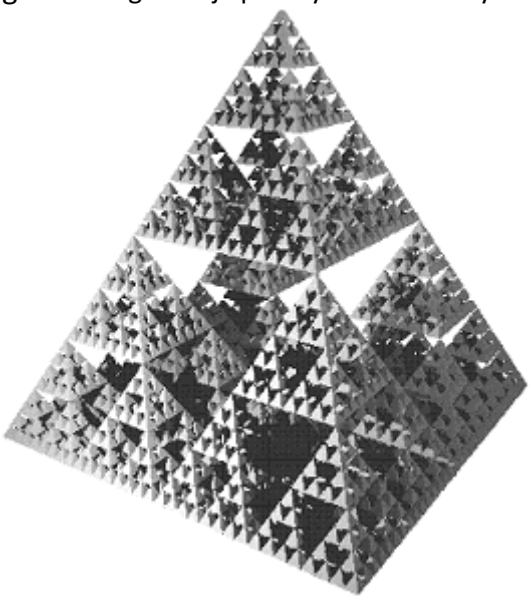
1. Proszę rozszerzyć tablicę vertices np. do 12 lub 18 elementów, tak, żeby można było narysować dwa lub trzy trójkąty w 2D. **(Proszę to potraktować jako pierwszą część zadania.)**
2. Proszę zmienić kolory rysowania trójkąta i tła.
3. Proszę zobaczyć jak działają inne tryby rysowania wierzchołków: `gl.POINTS, gl.LINES, gl.LINE_STRIP, gl.LINE_LOOP, gl.TRIANGLES, gl.TRIANGLE_STRIP, gl.TRIANGLE_FAN.`
4. Proszę wypróbować przykłady `triangle2.html i triangle3.html` – oba z sekcji `1-Triangle.` **Jako drugą część zadania proszę podesłać przykład ze zmodyfikowanym `fragment shaderem,` który wygeneruje ciekawszy wzór. Można (choć nie jest to obowiązkowe) skorzystać z przykładów ze strony thebookofshaders.com (omówię na zajęciach).**

Punkt 1. i 4. proszę potraktować jako jedno zadanie

Rzeczy poniżej są tylko do ewentualnego obejrzenia

2. Trójkąt Sierpińskiego zwany też uszczelką Sierpińskiego (Sierpinski Gasket)

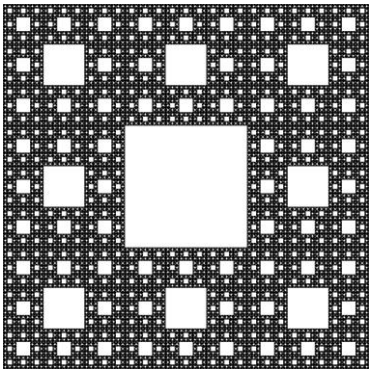
W drugiej grupie mamy kilka przykładów generowania rekurencyjnej figury fraktalnej, tzw. trójkąta Sierpińskiego.
`gasket1` generuje 5000 punktów dla zadanego podziału trójkąta. Shadery bez zmian. Można co najwyżej zmienić liczbę generowanych punktów.
`gasket2` rekurencyjnie generuje trójkat fraktalny. Można zmienić głębokość rekurencji.
`gasket3` generuje punkty w 3D i tworzy fraktalną bryłę, mniej więcej taką:

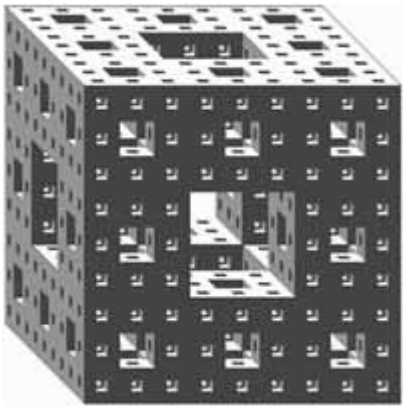


W vertex shaderze `gasket3` kolor punktu zależy od jego współrzędnych. Proszę zwiększyć liczbę generowanych punktów i zmienić zasadę zmiany koloru od położenia (np. w sposób nie liniowy wykorzystując sinus i cosinus).
`gasket4` generuje rekurencyjnie piramidki o ściankach w różnych kolorach.
`gasket5` wraca do uszczelki 2D, ale pozwala interaktywnie zmieniać głębokość rekurencji. Proszę zmienić tę głębokość.

W powyższych przykładach nie mamy oczywiście obiektu typu *camera*, ani też macierzy transformacji i rzutowania. Nie możemy więc zmieniać łatwo punktu patrzenia np. na uszczelkę 3D.

Dodatkowa rzecz możliwa do zrobienia: Proszę narysować zamiast uszczelki – dywan Sierpińskiego zbudowany w oparciu o kwadraty lub sześciany:





Do zrobienia

Uwagi w tekście powyżej.

3. Obracający się kwadrat + wstążka trójkątów (triangle strip)

W tej grupie mamy cztery programy. Trzy podobne obracające się kwadraty. Pierwszy bez interakcji, drugi z wyborem szybkości/kierunku przyciskami, trzeci z wyborem szybkości suwakiem.

Proszę zawrócić uwagę na mechanizm wprowadzenia ruchu.

1. Obroty realizują się w vertex shaderze i są sterowane zmienną uniform (parametrem) theta:

```
attribute vec4 vPosition;
uniform float theta;

void
main()
{
    float s = sin( theta );
    float c = cos( theta );

    gl_Position.x = -s * vPosition.y + c * vPosition.x;
    gl_Position.y =  s * vPosition.x + c * vPosition.y;
    gl_Position.z = 0.0;
    gl_Position.w = 1.0;
}
```

2. Wprowadzamy zmienną thetaloc, która użyta w instrukcji
   `thetaLoc = gl.getUniformLocation( program, "theta" );`
   jest identyfikatorem pamiętającym że do shadera `program` jest przekazywana zmienna typu uniform, która w samym shaderze będzie widziana pod nazwą theta.
3. Przekazywanie kolejnych wartości parametru do shadera odbywa się w pętli animacji w funkcji render():

```
function render() {

    gl.clear( gl.COLOR_BUFFER_BIT );

    theta += 0.1;
    gl.uniform1f( thetaLoc, theta );

    gl.drawArrays( gl.TRIANGLE_STRIP, 0, 4 );

    window.requestAnimFrame(render);
}
```

   gl.uniform1f przekazuje do shadera zgodnie z identyfikatorem thetaLoc jeden parametr. Do większej liczby parametrów mamy gl.uniform2f, gl.uniform3f, gl.uniform4f.

Do zrobienia

1. Proszę odpowiedzieć na pytanie jak rysowany jest kwadrat. Jak można go inaczej narysować?
2. Proszę dodać do obrotu kwadratu ruch posuwisty w lewo i w prawo (druga zmienna uniform), albo jeszcze lepiej ruch kwadratu po okręgu, elipsie lub bardziej złożonej krzywej Lissajou. Być może warto w tym celu kwadrat zmniejszyć.
3. Proszę dodatkowo zmienić kolor kwadratu w zależności od jego położenia. Np. z lewej czerwony, z prawej zielony.