Witold Alda

Grafika komputerowa. Laboratorium 8.

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/

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">
precision mediump float;     <!—set calculation precision on GPU -->
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 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.

Na koniec dołączany jest właściwy program *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
   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`) i pomalowania go jednolitym kolorem (`gl.ClearColor`).
2. Ładowanie shaderów, a przy tym ich kompilacja i linkowanie, wszystko za pomocą funkcji `initShaders();`
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`
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();`

Co możemy zrobić?

W pliku Lab08_Shader_wstęp_Angel korzystamy z czterech programów:

1. W CH0 - triangle.html
2. W CH2 – gasket2.html
3. W CH4 – cube.html
4. W CH6 – ShadedCube.html
5.