Grafika czasu rzeczywistego

Krzysztof Gdawiec

UNIWERSYTET ŚLĄSKI
INSTYTUT INFORMATYKI

WebGL 2 vs WebGL 1

WebGL 2 wprowadza wiele nowości, np. obiekty VAO, transformację sprzężenia zwrotnego, instancjonowanie, tekstury głębi, zapis do wielu buforów, zapytania o okluzję itp.

Zmienia również pewne kwestie związane z pisaniem shaderów. Jest to spowodowane zmianą wersji języka GLSL ES.

Aby korzystać z WebGL2 pierwsze co musimy zrobić to pobrać kontekst:

```
let gl = canvas.getContext("webgl2");
```

Wszystkie funkcje WebGL 1, z których do tej pory korzystaliśmy są dostępne w WebGL 2.

Po przejściu na WebGL 2 musimy zmodyfikować shadery.

Pierwszą rzeczą, którą musimy zmodyfikować jest wersja języka GLSL ES:

```
#version 300 es
```

W shaderze wierzchołków usunięto `attribute`. Zamiast tego atrybuty wierzchołka w shadrze wierzchołków oznaczane są przez **`in`**:

WebGL 1

```
attribute vec4 vPosition;
```

WebGL 2

```
in vec4 vPosition;
```

Przy przekazywaniu danych pomiędzy shaderami usunięto `varying`. Zamiast tego w shaderze wierzchołków używamy `out`, a w shaderze fragmentów `in`:

WebGL 1

shader wierzchołków

```
varying vec3 fColor;
```

shader fragmentów

```
varying vec3 fColor;
```

WebGL 2

shader wierzchołków

```
out vec3 fColor;
```

shader fragmentów

```
in vec3 fColor;
```

W WebGL 1 w shaderze fragmentów wynikowy kolor zapisywaliśmy w zmiennej `gl_FragColor`. W WebGL 2 zmienna ta została usunięta, a zamiast tego definiujemy własną zmienną wyjściową i to do niej przypisujemy wynikowy kolor.

WebGL 1 (shader fragmentów)

```
1  void main()
2  {
3    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
4  }
```

WebGL 2 (shader fragmentów)

```
1  out vec4 fragColor;
2
3  void main()
4  {
5    fragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
6  }
```

Przy odczytywaniu teksela w WebGL 1 używaliśmy funkcji, które w swojej nazwie miały rodzaj tekstury. W WebGL 2 mamy jedną funkcję `texture`, która rozpoznaje typ tekstury po typie samplera, który przekazujemy do niej.

WebGL 1

```
1  uniform sampler2D u_tex2D;
2  uniform samplerCube u_texCube;
3  ...
4  vec4 color1 = texture2D(u_tex2D, ...);
5  vec4 color2 = textureCube(u_texCube, ...);
```

WebGL 2

```
1  uniform sampler2D u_tex2D;
2  uniform samplerCube u_texCube;
3  ...
4  vec4 color1 = texture(u_tex2D, ...);
5  vec4 color2 = texture(u_texCube, ...);
```

W WebGL 1 musieliśmy pobrać lokalizację atrybutu, aby później ją wykorzystać w innych funkcjach wymagających tej lokalizacji, np. `enableVertexAttribArray`, `vertexAttribPointer`.

W WebGL 2 wprowadzono tzw. kwalifikatory formatu pozwalające ustawić określoną lokalizację atrybutu w shaderze wierzchołków:

```
1  layout(location = 0) in vec4 vPosition;
2  layout(location = 1) in vec3 vColor;
```

`vPosition` ma ustawioną pozycję `0`, a `vColor` pozycję `1`. Teraz wszędzie gdzie potrzebujemy podać lokalizację atrybutu podajemy ustawioną wartość.

Obiekty VAO

Obiekty tablic wierzchołków (ang. Vertex Array Object – VAO) został wprowadzony w WebGL 2. W pojedynczym VAO możemy umieszczać różne obiekty VBO, a następnie jednym poleceniem wyrysować.

Utworzenie unikatowego obiektu VAO:

```
gl.createVertexArray();
```

Funkcja zwraca obiekt VAO.

Sprawdzenie czy dany obiekt jest obiektem VAO:

```
gl.isVertexArray(vao);
```

`vao` obiekt, który chcemy sprawdzić. Funkcja zwraca `true` jeśli `vao` jest obiektem VAO, w przeciwnym razie zwraca `false`.

Dowiązanie obiektu VAO:

```
gl.bindVertexArray(vao);
```

`vao` obiekt VAO, który chcemy dowiązać.

Usunięcie obiektu VAO:

```
gl.deleteVertexArray(vao);
```

`vao` obiekty VAO do usunięcia.

WebGL 1

W funkcji inicjującej

```
1  let positions =
2    [
3      0.1, 0.0, 0.0, 1.0,
4      0.9, -0.5, 0.0, 1.0,
5      0.9, 0.5, 0.0, 1.0
6    ];
7
8  triangle.positionVBO = gl.createBuffer();
9  gl.bindBuffer( gl.ARRAY_BUFFER, triangle.positionVBO );
10 gl.bufferData( gl.ARRAY_BUFFER, new Float32Array( positions
       ), gl.STATIC_DRAW );
11
12 triangle.noVertices = 3;
```

W funkcji renderującej

```
1  gl.enableVertexAttribArray( vPosition );
2  gl.bindBuffer( gl.ARRAY_BUFFER, triangle.positionVBO );
3  gl.vertexAttribPointer( vPosition, 4, gl.FLOAT, false, 0, 0
       );
4
5  gl.drawArrays( gl.TRIANGLES, 0, triangle.noVertices );
```

WebGL 2

W funkcji inicjującej

```
1  let positions =
2    [
3      0.1, 0.0, 0.0, 1.0,
4      0.9, -0.5, 0.0, 1.0,
5      0.9, 0.5, 0.0, 1.0
6    ];
7
8  triangle.VAO = gl.createVertexArray();
9  gl.bindVertexArray( triangle.VAO );
10
11 triangle.positionVBO = gl.createBuffer();
12 gl.bindBuffer( gl.ARRAY_BUFFER, triangle.positionVBO );
13 gl.bufferData( gl.ARRAY_BUFFER, new Float32Array( positions
       ), gl.STATIC_DRAW );
14 gl.enableVertexAttribArray( 0 );
15 gl.vertexAttribPointer( 0, 4, gl.FLOAT, false, 0, 0 );
16
17 triangle.noVertices = 3;
```

W funkcji renderującej

```
1  gl.bindVertexArray( triangle.VAO );
2  gl.drawArrays( gl.TRIANGLES, 0, triangle.noVertices );
```