# WebGL 기초

WebGL은 종종 3D API로 치부됩니다. 사람들은 "WebGL을 사용해서 멋진 3D를 만들어야지"라고 생각하는데요. 사실 WebGL은 래스터화 엔진에 불과합니다. 여러분이 제공한 코드를 기반으로 점, 선, 삼각형 등을 그리죠. WebGL로 원하는 작업을 수행하려면 점, 선, 삼각형을 사용하는 코드를 작성해야 합니다.

WebGL은 컴퓨터에 있는 GPU에서 실행됩니다. 따라서 GPU에서 실행되는 코드를 제공해야 하는데요. 해당 코드는 함수 쌍 형태로 제공해야 합니다. 이 두 함수는 정점 셰이더와 프래그먼트 셰이더로 불리고 C/C++처럼 엄격한 Type을 가지는 GLSL로 작성되어 있습니다. 이 두 개를 합쳐서 *프로그램*이라고 부릅니다.

정점 셰이더의 역할은 정점 위치를 계산하는 겁니다. WebGL은 함수가 출력하는 위치를 기반으로 점, 선, 삼각형을 포함한 다양한 종류의 프리미티브를 래스터화할 수 있는데요. 래스터화할 때 프리미티브는 프래그먼트 셰이더라 불리는 두 번째 사용자 작성 함수를 호출합니다. 프래그먼트 셰이더의 역할은 현재 그려지는 프리미티브의 각 픽셀에 대한 색상을 계산하는 겁니다.

대부분의 WebGL API는 이러한 함수 쌍을 실행하기 위한 상태 설정에 관한 것입니다. 원하는 것을 그리기 위해서는 여러 상태를 설정하고 GPU에서 셰이더를 실행하는 `gl.drawArrays` 나 `gl.drawElements` 를 호출해서 함수 쌍을 실행해야 합니다.

이러한 함수가 접근하는 모든 데이터는 GPU에 제공되어야 하는데요. 셰이더가 데이터를 받을 수 있는 방법에는 4가지가 있습니다.

1. 속성 & 버퍼

   버퍼는 GPU에 업로드하는 2진 데이터 배열입니다. 일반적으로 버퍼는 위치, 법선, 텍스처 좌표, 정점 색상 등을 포함하지만 원하는 걸 자유롭게 넣어도 됩니다.

   속성은 버퍼에서 데이터를 가져오고 정점 셰이더에 제공하는 방법을 지정하는데 사용됩니다. 예를 들어 위치당 3개의 32비트 부동 소수점으로 위치를 버퍼에 넣을 수 있는데요. 어느 버퍼에서 위치를 가져올지, 어떤 타입의 데이터(3개의 32비트 부동 소수점)를 가져와야 하는지, 버퍼의 오프셋이 어느 위치에서 시작되는지, 한 위치에서 다음 위치로 갈 때 몇 바이트를 이동시킬 것인지 등을 특정 속성에 알려줘야 합니다.

   버퍼는 랜덤하게 접근할 수 없습니다. 대신에 정점 셰이더가 지정한 횟수만큼 실행되는데요. 실행될 때마다 지정된 버퍼에서 다음 값을 가져와 속성에 할당합니다.

2. 유니폼

   유니폼은 셰이더 프로그램을 실행하기 전 설정하는 사실상 전역 변수입니다.

3. 텍스처

   텍스처는 셰이더 프로그램에서 랜덤하게 접근할 수 있는 데이터 배열입니다. 텍스처에 넣는 대부분은 이미지 데이터지만 텍스처는 데이터일 뿐이며 색상 이외의 것도 쉽게 담을 수 있습니다.

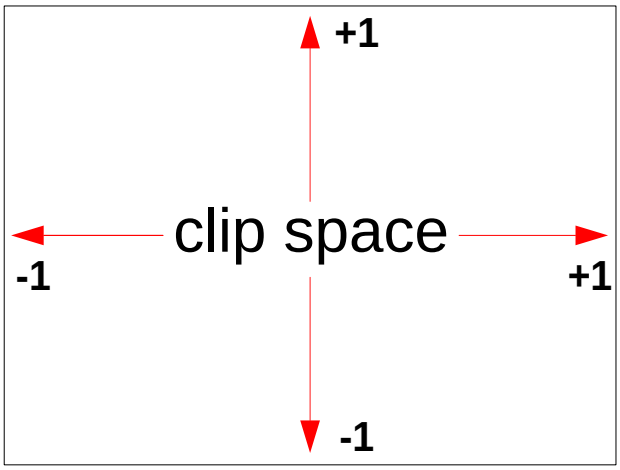
4. 베링

   베링은 정점 셰이더가 프래그먼트 셰이더에 데이터를 전달하는 방법입니다. 점, 선, 삼각형 등 렌더링되는 것에 따라 정점 셰이더에 의해 설정된 베링의 값은 프래그먼트 셰이더를 실행하는 동안 보간됩니다.

# WebGL Hello World

WebGL은 클립 공간의 좌표와 색상, 오직 2가지만을 다루는데요. 프로그래머로서 WebGL을 사용하는 여러분의 역할은 이 2가지를 작성하는 겁니다. 이를 위해 2개의 "셰이더"를 제공해야 합니다. 클립 공간 좌표를 제공하는 정점 셰이더, 색상을 제공하는 프래그먼트 셰이더입니다.

클립 공간 좌표는 캔버스 크기에 상관없이 항상 -1에서 +1까지입니다.




다음은 WebGL을 보여주는 가장 간단한 형태의 예제입니다.

정점 셰이더부터 시작해봅시다.

```
// 속성은 버퍼에서 데이터를 받습니다.
attribute vec4 a_position;

// 모든 셰이더는 main 함수를 가집니다.
void main() {

  // "gl_Position"은 정점 셰이더가 설정을 담당하는 특수 변수
  gl_Position = a_position;
}
```

GLSL 대신 자바스크립트로 작성된다면 이렇게 작성된다고 상상할 수 있습니다.

```
// *** 의사 코드!! ***

var positionBuffer = [
    0,   0, 0, 0,
    0, 0.5, 0, 0,
  0.7,   0, 0, 0,
];
var attributes = {};
var gl_Position;

drawArrays(..., offset, count) {
  var stride = 4;
  var size = 4;
  for (var i = 0; i < count; ++i) {
    // 다음 값 4개를 "positionBuffer"에서 속성 "a_position"으로 복사
    const start = offset + i * stride;
    attributes.a_position = positionBuffer.slice(start, start + size);
    runVertexShader();
    ...
    doSomethingWith_gl_Position();
  }
}
```

실제로는 `positionBuffer`가 2진 데이터(아래 참조)로 변환되어야 하기 때문에 그렇게 간단하지 않아서, 버퍼에서 데이터를 가져오기 위한 실제 계산은 약간 다를 수 있지만, 이걸로 정점 셰이더가 어떻게 실행되는지 아셨기를 바랍니다.

다음으로 프래그먼트 셰이더가 필요합니다.

```
// 프래그먼트 셰이더는 기본 정밀도를 가지고 있지 않으므로 하나를 선택해야 합니다.
// "mediump"은 좋은 기본값으로 "중간 정밀도"를 의미합니다.
precision mediump float;

void main() {
  // "gl_FragColor"는 프래그먼트 셰이더가 설정을 담당하는 특수 변수
  gl_FragColor = vec4(1, 0, 0.5, 1); // 자주색 반환
}
```

위에서 우리는 `gl_FragColor` 를 빨강 1, 초록 0, 파랑 0.5, 투명도 1인 `1, 0, 0.5, 1` 로 설정했는데요. WebGL에서 색상은 0에서 1까지입니다.

이제 두 셰이더 함수를 작성했으니 WebGL을 시작해봅시다.

먼저 HTML canvas 요소가 필요합니다.

```
<canvas id="c"></canvas>
```

그런 다음 자바스크립트에서 찾을 수 있습니다.

```
var canvas = document.querySelector("#c");
```

이제 WebGLRenderingContext를 만들 수 있습니다.

```
var gl = canvas.getContext("webgl");
if (!gl) {
  // webgl이 없어요!
  ...
}
```

이제 셰이더를 컴파일해서 GPU에 할당해야 하는데 먼저 문자열로 가져와야 합니다. 일반적으로 자바스크립트에서 문자열을 만드는 어떤 방법으로도 GLSL 문자열을 만들 수 있는데요. 문자열을 연결할 수도, AJAX를 이용해 다운로드할 수도, 여러 줄의 템플릿 문자열을 사용할 수도 있죠. 혹은 이 경우처럼 자바스크립트 타입이 아닌 스크립트 태그 안에 넣을 수도 있습니다.

```
<script id="vertex-shader-2d" type="notjs">

  // attribute는 buffer에서 데이터를 받음
  attribute vec4 a_position;

  // 모든 셰이더는 main 함수를 가짐
  void main() {

    // gl_Position은 정점 셰이더가 설정을 담당하는 특수 변수
    gl_Position = a_position;
  }

</script>

<script id="fragment-shader-2d" type="notjs">

  // 프래그먼트 셰이더는 기본 정밀도를 가지고 있지 않으므로 하나를 선택해야 합니다.
  // mediump은 좋은 기본값으로 "중간 정밀도"를 의미합니다.
  precision mediump float;

  void main() {
    // gl_FragColor는 프래그먼트 셰이더가 설정을 담당하는 특수 변수
    gl_FragColor = vec4(1, 0, 0.5, 1); // 자주색 반환
  }

</script>
```

사실 대부분의 3D 엔진은 다양한 종류의 템플릿, 문자열 연결 등을 사용하여 즉시 GLSL 셰이더를 생성하는데요. 이 사이트에 있는 예제들은 런타임에 GLSL을 생성해야 할 만큼 복잡하지 않습니다.

다음은 셰이더를 만들고, GLSL을 업로드한 다음, 셰이더를 컴파일하는 함수가 필요합니다. 참고로 함수의 이름을 보면 어떤 역할인지 알 수 있기 때문에 주석을 작성하지 않았습니다.

```
function createShader(gl, type, source) {
  var shader = gl.createShader(type);
  gl.shaderSource(shader, source);
  gl.compileShader(shader);

  var success = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
  if (success) {
    return shader;
  }

  console.log(gl.getShaderInfoLog(shader));
  gl.deleteShader(shader);
}
```

이제 두 셰이더를 만드는 함수를 호출할 수 있습니다.

```
var vertexShaderSource = document.querySelector("#vertex-shader-2d").text;
var fragmentShaderSource = document.querySelector("#fragment-shader-2d").text;

var vertexShader = createShader(gl, gl.VERTEX_SHADER, vertexShaderSource);
var fragmentShader = createShader(gl, gl.FRAGMENT_SHADER, fragmentShaderSource);
```

다음으로 두 셰이더를 *프로그램*으로 *연결*해야 합니다.

```
function createProgram(gl, vertexShader, fragmentShader) {
  var program = gl.createProgram();
  gl.attachShader(program, vertexShader);
  gl.attachShader(program, fragmentShader);
  gl.linkProgram(program);

  var success = gl.getProgramParameter(program, gl.LINK_STATUS);
  if (success) {
    return program;
  }

  console.log(gl.getProgramInfoLog(program));
  gl.deleteProgram(program);
}
```

그리고 호출합니다.

```
var program = createProgram(gl, vertexShader, fragmentShader);
```

이제 GPU에 GLSL 프로그램을 만들었으니 데이터를 제공해줘야 하는데요. WebGL의 주요 API는 GLSL 프로그램에 데이터를 제공하기 위한 상태 설정에 관한 겁니다. 이 경우 GLSL 프로그램에 대한 유일한 입력은 속성인 `a_position` 입니다. 먼저 방금 생성된 프로그램의 속성 위치를 찾습니다.

```
var positionAttributeLocation = gl.getAttribLocation(program, "a_position");
```

속성 위치(그리고 유니폼 위치)를 찾는 것은 렌더링할 때가 아니라 초기화하는 동안 해야 합니다.

속성은 버퍼에서 데이터를 가져오기 때문에 버퍼를 생성해야 합니다.

```
var positionBuffer = gl.createBuffer();
```

WebGL은 전역 바인드 포인트에 있는 많은 WebGL 리소스를 조작하게 해줍니다. 바인드 포인트는 WebGL 안에 있는 내부 전역 변수라고 생각하면 됩니다. 리소스를 바인드 포인트에 바인딩하면 모든 함수가 바인드 포인트를 통해 리소스를 참조합니다. 그럼 `positionBuffer` 를 바인딩해봅시다.

```
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
```

이제 바인드 포인트를 통해 해당 버퍼를 참조해서 데이터를 넣을 수 있습니다.

```
// 2D 포인트 3개
var positions = [
    0, 0,
    0, 0.5,
  0.7, 0,
];
gl.bufferData(
  gl.ARRAY_BUFFER,
  new Float32Array(positions),
  gl.STATIC_DRAW
);
```

여기까지 많은 게 있었는데요. 먼저 자바스크립트 배열인 `positions` 가 있습니다. 자바스크립트와 다르게 WebGL은 강력한 타입을 가지는 데이터가 필요하므로, `new Float32Array(positions)` 는 새로운 32비트 부동 소수점 배열을 생성하고 `positions` 에서 값을 복사합니다. 그런 다음 `gl.bufferData` 는 데이터를 GPU의 `positionBuffer` 로 복사합니다. 위에서 `ARRAY_BUFFER` 바인드 포인트에 할당했기 때문에 위치 버퍼를 사용하고 있습니다.

마지막 매개변수 `gl.STATIC_DRAW` 는 데이터를 어떻게 사용할 것인지 WebGL에 알려줍니다. WebGL은 특정 항목들을 최적화하기 위해 해당 힌트를 사용할 수 있는데요. `gl.STATIC_DRAW` 는 이 데이터가 많이 바뀌지 않을 것 같다고 WebGL에 알립니다.

지금까지 작성한 것은 *초기화 코드*입니다. 이 코드는 페이지를 로드할 때 한 번 실행되는데요. 아래부터는 *렌더링 코드* 또는 render/draw를 원할 때마다 실행되는 코드입니다.

## 렌더링

그리기 전에 캔버스 크기를 디스플레이 크기에 맞게 조절해야 합니다. 이미지처럼 캔버스는 2가지 크기를 가지는데요. 실제로 그 안에 있는 픽셀 수와 이와 별개로 표시되는 크기입니다. CSS는 캔버스가 표시되는 크기를 결정합니다. 이게 다른 방법들보다 훨씬 더 유연하기 때문에 항상 **CSS로 원하는 캔버스 크기를 설정**해야 합니다.

캔버스의 픽셀 수를 표시되는 크기와 일치하도록 만들기 위해 여기에서 읽을 수 있는 도우미 함수를 쓰고 있습니다.

여기 있는 대부분의 예제는 별도의 창에서 실행할 경우 캔버스 크기가 400x300픽셀이지만, 이 페이지에 있는 것처럼 iframe 내부라면 사용 가능한 공간을 채우기 위해 늘어나는데요. CSS로 크기를 결정한 다음 그것과 일치하도록 조정하여 이러한 경우를 모두 쉽게 처리할 수 있습니다.

```
webglUtils.resizeCanvasToDisplaySize(gl.canvas);
```

`gl_Position` 으로 설정할 클립 공간 값을 어떻게 화면 공간으로 변환하는지 WebGL에 알려줘야 하는데요. 이를 위해 `gl.viewport` 를 호출해서 현재 캔버스 크기를 전달해야 합니다.

```
gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
```

이는 -1 <-> +1 클립 공간을, x는 0 <-> `gl.canvas.width` 로, y는 0 <-> `gl.canvas.height` 로 매핑시켜줍니다.

캔버스를 지워봅시다. `0, 0, 0, 0` 은 각각 빨강, 초록, 파랑, 투명도이기 때문에 이는 캔버스를 투명하게 만듭니다.

```
// 캔버스 지우기
gl.clearColor(0, 0, 0, 0);
gl.clear(gl.COLOR_BUFFER_BIT);
```

실행할 셰이더 프로그램을 WebGL에 알려줍니다.

```
// 프로그램(셰이더 쌍) 사용 지시
gl.useProgram(program);
```

다음으로 위에서 설정한 버퍼에서 데이터를 가져와 셰이더의 속성에 제공하는 방법을 WebGL에 알려줘야 하는데요. 우선 속성을 활성화해야 합니다.

```
gl.enableVertexAttribArray(positionAttributeLocation);
```

그런 다음 데이터를 어떻게 꺼낼지 지정합니다.

```
// 위치 버퍼 할당
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);

// positionBuffer(ARRAY_BUFFER)의 데이터를 꺼내오는 방법을 속성에 지시
var size = 2;          // 반복마다 2개의 컴포넌트
var type = gl.FLOAT;   // 데이터는 32비트 부동 소수점
var normalize = false; // 데이터 정규화 안 함
var stride = 0;        // 0 = 다음 위치를 가져오기 위해 반복마다 size * sizeof(type) 만큼 앞으로 이동
var offset = 0;        // 버퍼의 처음부터 시작
gl.vertexAttribPointer(
  positionAttributeLocation,
  size,
  type,
  normalize,
  stride,
  offset
);
```

`gl.vertexAttribPointer` 의 숨겨진 부분은 현재 바인딩된 `ARRAY_BUFFER` 를 속성에 할당한다는 겁니다. 다시 말해 이 속성은 이제 `positionBuffer` 에 바인딩됩니다. 이건 `ARRAY_BUFFER` 바인드 포인트에 다른 걸 자유롭게 바인딩할 수 있다는 의미인데요. 다른 걸 바인딩해도 속성은 계속해서 `positionBuffer` 를 사용합니다.

참고로 GLSL 정점 셰이더 관점에서 속성 `a_position` 은 `vec4` 입니다.

```
attribute vec4 a_position;
```

`vec4` 는 4개의 부동 소수점 값입니다. 자바스크립트의 `a_position = {x: 0, y: 0, z: 0, w: 0}` 와 같이 생각할 수 있습니다. 위에서 `size = 2` 로 설정했는데요. 속성의 기본값은 `0, 0, 0, 1` 이기 때문에 이 속성은 버퍼에서 처음 2개의 값(x/y)을 가져옵니다. z와 w는 기본값으로 각각 0과 1이 될 겁니다.

드디어 GLSL 프로그램을 실행하도록 WebGL에 요청할 수 있습니다.

```
var primitiveType = gl.TRIANGLES;
var offset = 0;
var count = 3;
gl.drawArrays(primitiveType, offset, count);
```

count 가 3이기 때문에 정점 셰이더를 세 번 실행할 겁니다. 먼저 정점 셰이더 속성의 a_position.x 와 a_position.y 가 positionBuffer 의 첫 2개의 값으로 설정됩니다. 두 번째로 a_position.x 와 a_position.y 가 그다음 2개의 값으로 설정됩니다. 마지막에는 남아있는 2개의 값으로 설정됩니다.

primitiveType 을 gl.TRIANGLES 로 설정했기 때문에, 정점 셰이더가 3번 실행될 때마다, WebGL은 gl_Position 에 설정한 3개의 값을 기반으로 삼각형을 그리는데요. 캔버스 크기에 상관없이 이 값들은 -1에서 1사이의 클립 공간 좌표 안에 있습니다.

정점 셰이더는 단순히 positionBuffer 값을 gl_Position 에 복사하기 때문에 삼각형은 클립 공간 좌표에 그려집니다.

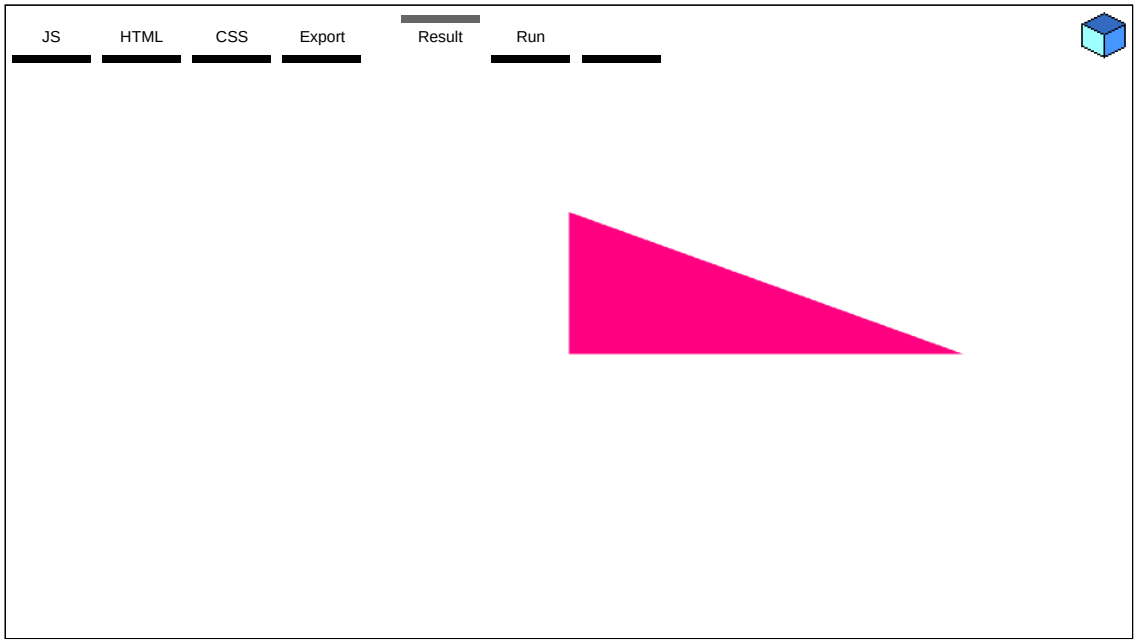
```
   0, 0,
   0, 0.5,
 0.7, 0,
```

캔버스 크기가 400x300이라면 이런 식으로 클립 공간을 화면 공간으로 변환합니다.

```
클립 공간            화면 공간
  0, 0       ->   200, 150
  0, 0.5     ->   200, 225
0.7, 0       ->   340, 150
```

WebGL은 이제 삼각형을 렌더링할 겁니다. 그리려는 모든 픽셀에 대해 WebGL은 프래그먼트 셰이더를 호출하는데요. 프래그먼트 셰이더는 gl_FragColor 를 1, 0, 0.5, 1 로 설정합니다. 캔버스는 채널당 8비트이기 때문에 이는 WebGL이 [255, 0, 127, 255] 를 캔버스에 작성한다는 걸 의미합니다.

다음은 라이브 버전입니다.




새 창을 열려면 여기를 클릭

위 경우에 정점 셰이더가 데이터를 직접 전달하는 것 외에는 아무것도 하지 않는데요. 위치 데이터가 이미 클립 공간에 있으므로 할 일이 없습니다. *WebGL은 래스터화 API에 불과하기 때문에 3D를 원한다면 3D를 클*

*립 공간으로 변환하는 셰이더를 작성해야 합니다.*

아마 삼각형이 중앙에서 시작하여 우측 상단으로 가는 이유가 궁금하실텐데요. x 의 클립 공간은 -1부터 +1 까지의 값을 가집니다. 즉 0이 중앙이고 양수 값이 오른쪽이라는 걸 의미합니다.

위쪽에 있는 이유는 클립 공간에서 -1은 아래쪽에 있고 +1은 위쪽에 있기 때문입니다. 즉 0이 중앙이고 양수가 중앙보다 위에 있다는 걸 의미합니다.

2D의 경우 클립 공간보다 픽셀로 작업하는 게 좋으니, 위치를 픽셀로 제공하고 클립 공간으로 변환할 수 있도록 셰이더를 바꿔봅시다. 여기 새로운 정점 셰이더입니다.

```
<script id="vertex-shader-2d" type="notjs">

-attribute vec4 a_position;
 attribute vec2 a_position;

 uniform vec2 u_resolution;

 void main() {
   // 위치를 픽셀에서 0.0과 1.0사이로 변환
   vec2 zeroToOne = a_position / u_resolution;

   // 0->1에서 0->2로 변환
   vec2 zeroToTwo = zeroToOne * 2.0;

   // 0->2에서 -1->+1로 변환 (클립 공간)
   vec2 clipSpace = zeroToTwo - 1.0;

   gl_Position = vec4(clipSpace, 0, 1);
 }

</script>
```

알아 두어야 할 변경 사항들이 몇 가지 있습니다. x 와 y 만 사용하기 때문에 a_position 을 vec2 로 수정했습니다. vec2 는 vec4 와 비슷하지만 x 와 y 만을 가집니다.

다음으로 u_resolution 이라는 uniform 을 추가했습니다. 이를 설정하려면 해당 위치를 찾아야 합니다.

```
var resolutionUniformLocation = gl.getUniformLocation(program, "u_resolution");
```

나머지는 주석을 보면 알 수 있습니다. u_resolution 을 캔버스의 해상도로 설정함으로써, 셰이더는 픽셀 좌표로 제공한 positionBuffer 에 넣은 위치를 가져와 클립 공간으로 변환합니다.

이제 위치 값을 클립 공간에서 픽셀로 바꿀 수 있습니다. 이번에는 각각 3개의 점으로 이루어진 삼각형 두 개로 만드는 사각형을 그려볼 겁니다.

```
var positions = [
  10, 20,
  80, 20,
  10, 30,
  10, 30,
  80, 20,
  80, 30,
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);
```

사용할 프로그램을 설정한 뒤 우리가 만든 유니폼의 값을 설정할 수 있습니다. gl.useProgram 은 위의 gl.bindBuffer 처럼 현재 프로그램을 설정하는데요. 이후 모든 gl.uniformXXX 함수는 현재 설정된 프로그램의 유니폼을 설정합니다.

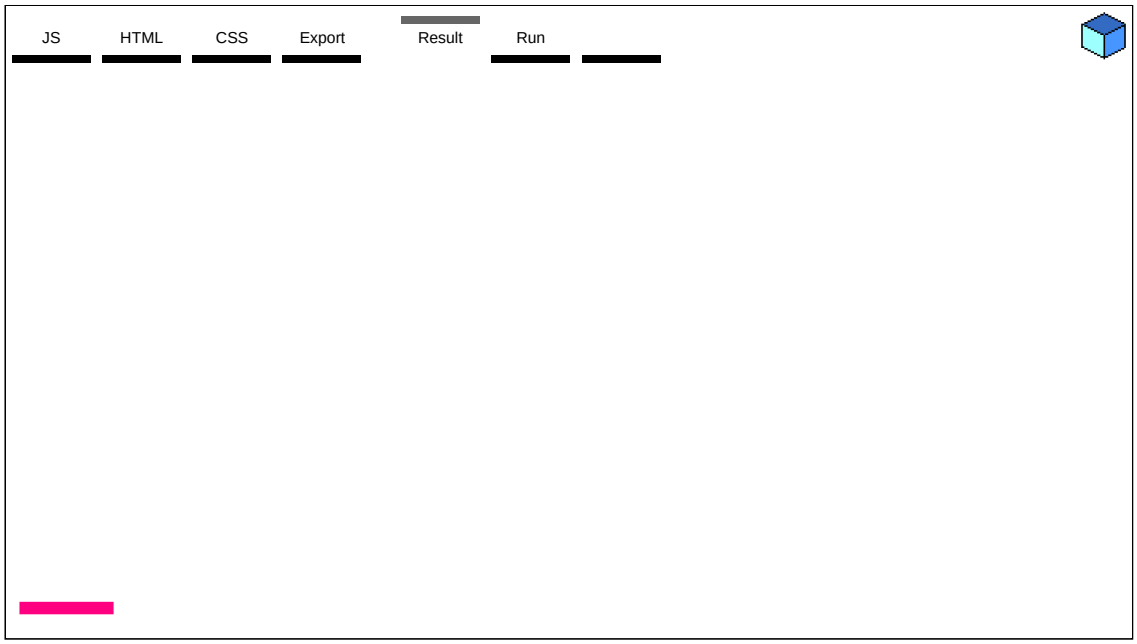
```
gl.useProgram(program);

...
```

```
// 해상도 설정
gl.uniform2f(resolutionUniformLocation, gl.canvas.width, gl.canvas.height);
```

그리고 2개의 삼각형을 그리기 위해서는 당연히 정점 셰이더를 6번 호출해야 하므로 `count`를 `6`으로 바꿔야 합니다.

```
// 그리기
var primitiveType = gl.TRIANGLES;
var offset = 0;
var count = 6;
gl.drawArrays(primitiveType, offset, count);
```

참고: 이 예제와 앞으로 나오는 모든 예제들은 셰이더를 컴파일하고 연결하는 함수가 포함된 `webgl-utils.js`를 사용합니다. 상용구 코드로 예제를 복잡하게 할 필요는 없을 것 같습니다.

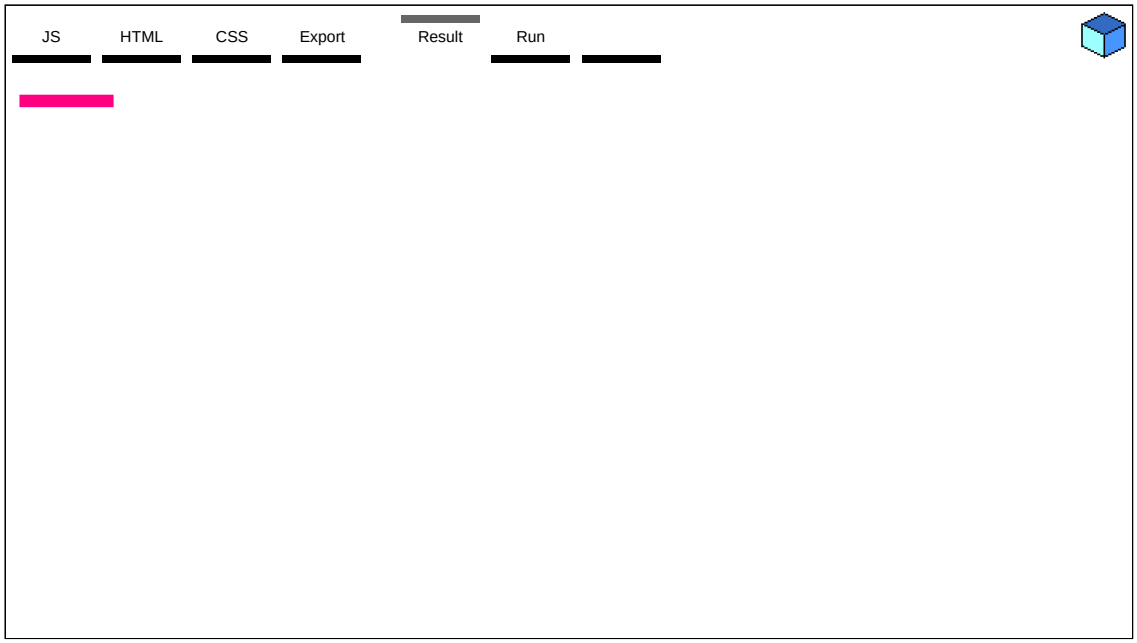



새 창을 열려면 여기를 클릭

사각형이 좌측 하단 근처에 있는 걸 확인할 수 있는데요. WebGL은 양수 Y를 위쪽으로, 음수 Y를 아래쪽으로 간주합니다. 클립 공간에서 좌측 하단 모서리는 -1,-1이 됩니다. 아직 어떤 부호도 바꾸지 않았기 때문에 현재 수식에서 0,0은 좌측 하단 모서리가 됩니다. 이를 2D 그래픽 API에서 더 전통적으로 사용되는 좌측 상단 모서리가 되도록 클립 공간 y좌표를 뒤집을 수 있습니다.

```
  gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
```

이제 우리가 예상한 위치에 사각형이 있습니다.




[새 창을 열려면 여기를 클릭]

함수에 사각형을 정의하는 코드를 만들어서 다른 크기의 사각형도 호출할 수 있도록 해봅시다. 이 작업을 하면서 색상 설정이 가능하도록 만들 겁니다.

먼저 프래그먼트 셰이더가 색상 유니폼 입력을 가져오도록 만듭니다.

```
<script id="fragment-shader-2d" type="notjs">
  precision mediump float;

  uniform vec4 u_color;

  void main() {
    gl_FragColor = u_color;
  }
</script>
```

그리고 다음은 임의의 위치와 임의의 색상으로 50개의 사각형을 그리는 새로운 코드입니다.

```
var colorUniformLocation = gl.getUniformLocation(program, "u_color");
...

// 임의의 색상으로 임의의 사각형 50개 그리기
for (var ii = 0; ii < 50; ++ii) {
  // 임의의 사각형 설정
  // ARRAY_BUFFER 바인드 포인트에 마지막으로 바인딩한 것이므로 `positionBuffer`에 작성됩니다.
  setRectangle(
    gl,
    randomInt(300),
    randomInt(300),
    randomInt(300),
    randomInt(300)
  );

  // 임의의 색상 설정
  gl.uniform4f(
    colorUniformLocation,
    Math.random(),
    Math.random(),
    Math.random(),
    1
  );

  // 사각형 그리기
  gl.drawArrays(gl.TRIANGLES, 0, 6);
}

// 0부터 -1사이 임의의 정수 반환
function randomInt(range) {
  return Math.floor(Math.random() * range);
}
```

```
// 사각형을 정의한 값들로 버퍼 채우기
function setRectangle(gl, x, y, width, height) {
  var x1 = x;
  var x2 = x + width;
  var y1 = y;
  var y2 = y + height;

  // 참고: gl.bufferData(gl.ARRAY_BUFFER, ...)는 `ARRAY_BUFFER` 바인드 포인트에 바인딩된 버퍼에 영향을 주지만 지금까지는 하나의 버퍼만 있
  // 두 개 이상이라면 원하는 버퍼를 `ARRAY_BUFFER`에 먼저 바인딩해야 합니다.
  gl.bufferData(
    gl.ARRAY_BUFFER,
    new Float32Array([
      x1, y1,
      x2, y1,
      x1, y2,
      x1, y2,
      x2, y1,
      x2, y2
    ]),
    gl.STATIC_DRAW
  );
}
```

그리고 여기 사각형들입니다.


새 창을 열려면 여기를 클릭

이 글을 보고 WebGL이 실제로는 아주 단순한 API라고 느끼셨기를 바랍니다. 네 단순하다는 말은 아마 틀릴지도 모르지만 하는 일은 단순합니다. 사용자가 제공한 두 함수인 정점 셰이더와 프래그먼트 셰이더를 실행시키고 점, 선, 삼각형을 그릴 뿐입니다. 3D를 작업하면서 더 복잡해질 수 있지만, 그 복잡함은 프로그래머에 의해 더 복잡한 셰이더의 형태로 추가됩니다. WebGL API 자체는 래스터화 엔진에 불과하며 개념적으로는 꽤 단순합니다.

속성과 유니폼 2개에 데이터를 제공하는 방법을 보여주는 예제를 다뤘는데요. 일반적으로는 여러 속성과 많은 유니폼을 가집니다. 이 글의 서두에서 *베링*과 *텍스처*도 언급했었는데 이것들은 다른 글에서 소개하겠습니다.

계속하기 전에 *대부분*의 어플리케이션은 `setRectangle`에서 했던 것처럼, 버퍼의 데이터를 업데이트하는 게 일반적이지 않다는 것을 말하고 싶습니다. 이 예제를 사용한 것은 픽셀 좌표를 입력으로 보여주고, GLSL에서 간단히 계산하는 걸 보여주기 때문에, 설명하기에 가장 쉬운 방법이라고 생각했기 때문입니다. 이게 틀리다는 것은 아니고, 올바른 방법인 경우도 많지만, WebGL에서 위치, 방향, 크기를 조정하는 좀 더 일반적인 방법을 확인하려면 계속 읽어주세요.

웹 개발이 처음이든 아니든 WebGL 개발 방법에 관한 몇 가지 팁을 보려면 설정 및 설치를 확인해주세요.

WebGL을 100% 처음 배우고 GLSL이나 셰이더 혹은 GPU가 무엇인지 모르겠다면 WebGL이 실제로 작동하는 원리 기초를 확인해주세요. WebGL 작동 방식을 이해하는 또 다른 방법으로 대화형 상태 다이어그램을 보실 수도 있습니다.

여기있는 대부분의 예제에서 사용된 상용구 코드도 간략하게 읽어보세요. 안타깝게도 거의 모든 예제가 한 가지만 그려서 해당 구조를 보여주지 않기 때문에, 일반적인 WebGL 앱이 어떻게 구조화되어 있는지에 알 수 있도록 여러 물체를 그리는 방법도 훑어봐야 합니다.

그게 아니라면 여기에서 2가지 방향으로 갈 수 있는데요. 이미지 처리에 관심이 있다면 2D 이미지 처리 방법을 보시면 됩니다. 평행 이동, 회전, 스케일 그리고 궁극적으로 3D에 대해 배우고 싶다면 여기부터 시작하시면 됩니다.

## type="notjs"는 어떤 의미인가요?

<script> 태그는 기본적으로 자바스크립트가 포함합니다. 타입을 넣지 않거나 type="javascript" 또는 type="text/javascript" 라고 넣으면 브라우저는 내용을 자바스크립트로 해석하는데요. 이외에 다른 type 을 넣으면 브라우저는 스크립트 태그의 내용을 무시합니다. 즉 type="notjs" 나 type="foobar" 는 브라우저에서 아무런 의미가 없습니다.

이는 셰이더를 수정하기 쉽게 만들어줍니다. 다른 대안으로는 다음과 같은 문자열 연결이 있습니다.

```
var shaderSource =
  "void main() {\n" +
  "  gl_FragColor = vec4(1,0,0,1);\n" +
  "}";
```

또는 ajax 요청으로 셰이더를 가져올 수 있지만 느리고 비동기적입니다.

한 가지 더 현대적인 대안은 템플릿 리터럴을 사용하는 겁니다.

```
var shaderSource = `
  void main() {
    gl_FragColor = vec4(1,0,0,1);
  }
`;
```

템플릿 리터럴은 WebGL을 지원하는 모든 브라우저에서 동작하는데요. 안타깝게도 정말 오래된 브라우저에서는 동작하지 않으니, 이런 브라우저들을 위한 폴백을 지원한다면, 템플릿 리터럴을 사용하지 않거나 트랜스파일러를 사용해야 합니다.