

WebGL 셰이더와 GLSL

이 글은 [WebGL 기초](#)에서 이어집니다. 아직 [WebGL 작동 방식](#)을 읽지 않았다면 먼저 읽어보는 게 좋습니다.

셰이더와 GLSL에 대해 언급했지만 실제로 구체적인 세부 사항은 다루지 않았는데요. 예제로 충분하면 좋겠지만 혹시 모르니 좀 더 확실하게 해봅시다.

[작동 원리](#)에서 언급했듯이 WebGL은 뭔가를 그릴 때마다 2개의 셰이더를 필요로 하는데요. 바로 정점 셰이더와 프래그먼트 셰이더입니다. 각각의 셰이더는 함수인데요. 정점 셰이더와 프래그먼트 셰이더는 함께 셰이더 프로그램(또는 그냥 프로그램)으로 연결됩니다. 일반적인 WebGL 앱은 많은 셰이더 프로그램을 가집니다.

정점 셰이더

정점 셰이더의 역할은 클립 공간 좌표를 생성하는 겁니다. 항상 이런 형식을 취하는데요.

```
void main() {
    gl_Position = doMathToMakeClipspaceCoordinates
}
```

셰이더는 정점마다 한 번씩 호출됩니다. 호출될 때마다 특수 전역 변수, `gl_Position`을 어떤 클립 공간 좌표로 설정해야 합니다.

정점 셰이더는 데이터가 필요한데요. 3가지 방법으로 데이터를 가져올 수 있습니다.

1. [속성](#) (버퍼에서 가져온 데이터)
2. [유니폼](#) (단일 그리기 호출의 모든 정점에 대해 동일하게 유지되는 값)
3. [텍스처](#) (픽셀/텍셀 데이터)

속성

가장 일반적인 방법은 버퍼와 속성을 통하는 겁니다. [작동 원리](#)에서 버퍼와 속성을 다뤘었죠. 먼저 버퍼를 만들고,

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

이 버퍼에 데이터를 넣은 뒤,

```
gl.bindBuffer(gl.ARRAY_BUFFER, buf);
gl.bufferData(gl.ARRAY_BUFFER, someData, gl.STATIC_DRAW);
```

만든 셰이더 프로그램을 통해 초기화 시 속성의 위치를 찾고,

```
var positionLoc = gl.getAttribLocation(someShaderProgram, "a_position");
```

렌더링할 때 해당 버퍼에서 속성으로 데이터를 어떻게 가져올지 WebGL에 지시하는데,

```
// 이 속성의 버퍼에서 데이터 가져오기 활성화
gl.enableVertexAttribArray(positionLoc);

var numComponents = 3; // (x, y, z)
var type = gl.FLOAT; // 32비트 부동 소수점 값
var normalize = false; // 값 원본 그대로 유지
var offset = 0; // 버퍼의 처음부터 시작
var stride = 0; // 다음 정점으로 가기 위해 이동하는 바이트 수
// 0 = `type`과 `numComponents`에 맞는 스트라이드 사용

gl.vertexAttribPointer(positionLoc, numComponents, type, false, stride, offset);
```

WebGL 기초에서 우리는 셰이더에서 수식을 사용하지 않고 데이터를 직접 전달했습니다.

```
attribute vec4 a_position;

void main() {
    gl_Position = a_position;
}
```

클립 공간 정점을 버퍼에 넣으면 동작할 겁니다.

속성은 타입으로 float, vec2, vec3, vec4, mat2, mat3, mat4를 사용할 수 있습니다.

유니폼

셰이더 유니폼은 그리기 호출의 모든 정점에 대해 동일하게 유지되며 셰이더에 전달되는 값입니다. 간단한 예로 위의 정점 셰이더에 오프셋을 추가할 수 있습니다.

```
attribute vec4 a_position;
uniform vec4 u_offset;

void main() {
    gl_Position = a_position + u_offset;
}
```

이제 모든 정점을 일정량만큼 오프셋 할 수 있습니다. 먼저 초기화 시 유니폼의 위치를 찾아야 합니다.

```
var offsetLoc = gl.getUniformLocation(someProgram, "u_offset");
```

그런 다음 그리기 전에 유니폼을 설정합니다.

```
gl.uniform4fv(offsetLoc, [1, 0, 0, 0]); // 화면 오른쪽 절반으로 오프셋
```

참고로 유니폼은 개별 셔이더 프로그램에 속합니다. 만약 이름이 같은 유니폼을 가진 셔이더 프로그램이 여러 개 있다면, 두 유니폼 모두 고유한 위치와 값을 가지는데요. `gl.uniform???` 을 호출하면 현재 프로그램의 유니폼만 설정합니다. 현재 프로그램은 `gl.useProgram`에 전달한 마지막 프로그램입니다.

유니폼은 여러 타입을 가질 수 있는데요. 각 타입마다 설정을 위해 해당하는 함수를 호출해야 합니다.

```
gl.uniform1f (floatUniformLoc, v); // float
gl.uniform1fv(floatUniformLoc, [v]); // float 또는 float 배열
gl.uniform2f (vec2UniformLoc, v0, v1); // vec2
gl.uniform2fv(vec2UniformLoc, [v0, v1]); // vec2 또는 vec2 배열
gl.uniform3f (vec3UniformLoc, v0, v1, v2); // vec3
gl.uniform3fv(vec3UniformLoc, [v0, v1, v2]); // vec3 또는 vec3 배열
gl.uniform4f (vec4UniformLoc, v0, v1, v2, v4); // vec4
gl.uniform4fv(vec4UniformLoc, [v0, v1, v2, v4]); // vec4 또는 vec4 배열

gl.uniformMatrix2fv(mat2UniformLoc, false, [ 4x element array ]); // mat2 또는 mat2 배열
gl.uniformMatrix3fv(mat3UniformLoc, false, [ 9x element array ]); // mat3 또는 mat3 배열
gl.uniformMatrix4fv(mat4UniformLoc, false, [ 16x element array ]); // mat4 또는 mat4 배열

gl.uniform1i (intUniformLoc, v); // int
gl.uniform1iv(intUniformLoc, [v]); // int 또는 int 배열
gl.uniform2i (ivec2UniformLoc, v0, v1); // ivec2
gl.uniform2iv(ivec2UniformLoc, [v0, v1]); // ivec2 또는 ivec2 배열
gl.uniform3i (ivec3UniformLoc, v0, v1, v2); // ivec3
gl.uniform3iv(ivec3UniformLoc, [v0, v1, v2]); // ivec3 또는 ivec3 배열
gl.uniform4i (ivec4UniformLoc, v0, v1, v2, v4); // ivec4
gl.uniform4iv(ivec4UniformLoc, [v0, v1, v2, v4]); // ivec4 또는 ivec4 배열

gl.uniform1i (sampler2DUniformLoc, v); // sampler2D (texture)
gl.uniform1iv(sampler2DUniformLoc, [v]); // sampler2D 또는 sampler2D 배열

gl.uniform1i (samplerCubeUniformLoc, v); // samplerCube (texture)
gl.uniform1iv(samplerCubeUniformLoc, [v]); // samplerCube 또는 samplerCube 배열
```

`bool`, `bvec2`, `bvec3`, `bvec4` 타입도 있는데요. `gl.uniform?f?` 또는 `gl.uniform?i?` 함수를 사용합니다.

배열의 경우 배열의 모든 유니폼을 한번에 설정할 수 있습니다.

```
// 셰이더
uniform vec2 u_someVec2[3];

// 초기화 시 자바스크립트
var someVec2Loc = gl.getUniformLocation(someProgram, "u_someVec2");

// 렌더링할 때
gl.uniform2fv(someVec2Loc, [1, 2, 3, 4, 5, 6]); // u_someVec2의 전체 배열 설정
```

하지만 배열의 개별 요소를 설정하고 싶다면 각 요소의 위치를 개별적으로 찾아야 합니다.

```
// 초기화 시 자바스크립트
var someVec2Element0Loc = gl.getUniformLocation(someProgram, "u_someVec2[0]");
var someVec2Element1Loc = gl.getUniformLocation(someProgram, "u_someVec2[1]");
var someVec2Element2Loc = gl.getUniformLocation(someProgram, "u_someVec2[2]");

// 렌더링할 때
gl.uniform2fv(someVec2Element0Loc, [1, 2]); // 요소 0 설정
gl.uniform2fv(someVec2Element1Loc, [3, 4]); // 요소 1 설정
gl.uniform2fv(someVec2Element2Loc, [5, 6]); // 요소 2 설정
```

마찬가지로 구조체를 생성하면,

```
struct SomeStruct {
    bool active;
    vec2 someVec2;
};

uniform SomeStruct u_someThing;
```

각 필드를 개별적으로 찾아야 합니다.

```
var someThingActiveLoc = gl.getUniformLocation(someProgram, "u_someThing.active");
var someThingSomeVec2Loc = gl.getUniformLocation(someProgram, "u_someThing.someVec2");
```

정점 셰이더의 텍스처

[프래그먼트 셰이더의 텍스처를 봄주세요.](#)

프래그먼트 셰이더

프래그먼트 셰이더의 역할은 래스터화되는 현재 픽셀의 색상을 제공하는 것입니다. 항상 이런 형식을 취하는데요.

```
precision mediump float;
```

```
void main() {
    gl_FragColor = doMathToMakeAColor;
}
```

프래그먼트 셰이더는 각 픽셀마다 한 번씩 호출됩니다. 호출될 때마다 특수 전역 변수, `gl_FragColor`를 어떤 색상으로 설정해줘야 합니다.

프래그먼트 셰이더는 데이터가 필요한데요. 3가지 방법으로 데이터를 가져올 수 있습니다.

1. 유니폼 (단일 그리기 호출의 모든 정점에 대해 동일하게 유지되는 값)
2. 텍스처 (픽셀/텍셀 데이터)
3. 베링 (정점 셰이더에서 전달되고 보간된 데이터)

프래그먼트 셜이더의 유니폼

셰이더의 유니폼을 봐주세요.

프래그먼트 셜이더의 텍스처

셰이더의 텍스처에서 값을 가져오면 `sampler2D` 유니폼을 생성하고 값을 추출하기 위해 GLSL 함수 `texture2D`를 사용합니다.

```
precision mediump float;

uniform sampler2D u_texture;

void main() {
    vec2 texcoord = vec2(0.5, 0.5) // 텍스처 중앙에서 값 가져오기
    gl_FragColor = texture2D(u_texture, texcoord);
}
```

텍스처에서 나오는 데이터는 수많은 설정에 따라 달라집니다. 최소한의 텍스처 데이터를 생성하고 넣어 보겠습니다.

```
var tex = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, tex);
var level = 0;
var width = 2;
var height = 1;
var data = new Uint8Array([
    255, 0, 0, 255, // 빨강 픽셀
    0, 255, 0, 255, // 초록 픽셀
]);
gl.texImage2D(gl.TEXTURE_2D, level, gl.RGBA, width, height, 0, gl.RGBA, gl.UNSIGNED_BYTE, data);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
```

초기화 시 셜이더 프로그램의 유니폼 위치를 찾으며,

```
var someSamplerLoc = gl.getUniformLocation(someProgram, "u_texture");
```

렌더링할 때 텍스처 유닛에 텍스처를 할당하고,

```
var unit = 5; // 텍스처 유닛 선택
gl.activeTexture(gl.TEXTURE0 + unit);
gl.bindTexture(gl.TEXTURE_2D, tex);
```

텍스처를 바인딩한 유닛이 무엇인지 세이더에 알려줍니다.

```
gl.uniform1i(someSamplerLoc, unit);
```

베링

베링은 동작 원리에 다룬 정점 세이더에서 프래그먼트 세이더로 값을 전달하는 방법입니다.

베링을 사용하려면 정점 세이더와 프래그먼트 세이더 양쪽에 일치하는 베링을 선언해야 하는 데요. 각 정점마다 정점 세이더의 베링을 어떤 값으로 설정합니다. WebGL이 픽셀을 그릴 때 이 값들 사이를 보간하고 프래그먼트 세이더에서 대응하는 베링으로 전달할 겁니다.

정점 세이더:

```
attribute vec4 a_position;
uniform vec4 u_offset;
varying vec4 v_positionWithOffset;

void main() {
    gl_Position = a_position + u_offset;
    v_positionWithOffset = a_position + u_offset;
}
```

프래그먼트 세이더:

```
precision mediump float;
varying vec4 v_positionWithOffset;

void main() {
    // 클립 공간에서 (-1 <-> +1) color space로 (0 -> 1) 변환
    vec4 color = v_positionWithOffset * 0.5 + 0.5
    gl_FragColor = color;
}
```

위 예제는 사실 말도 안되는 예제입니다. 일반적으로는 클립 공간 값을 프래그먼트 셰이더에 직접 복사해서 색상으로 사용하지 않는데요. 그럼에도 불구하고 이 코드는 작동하며 색상을 만들어냅니다.

GLSL

GLSL은 Graphics Library Shader Language의 약자로 셰이더가 작성되는 언어입니다. 이 언어는 자바스크립트에서 흔하지 않은 특별한 준 고유 기능을 가지고 있는데요. 그래픽 래스터화에 일반적으로 필요한 수학적 계산을 수행하도록 설계되었습니다. 예를 들어 각각 2개의 값, 3개의 값, 4개의 값을 나타내는 `vec2`, `vec3`, `vec4` 같은 타입들이 내장되어 있는데요. 마찬가지로 2×2 , 3×3 , 4×4 행렬을 나타내는 `mat2`, `mat3`, `mat4`가 있습니다. `vec`에 스칼라를 곱하는 등의 작업을 수행할 수도 있죠.

```
vec4 a = vec4(1, 2, 3, 4);
vec4 b = a * 2.0;
// 현재 b는 vec4(2, 4, 6, 8);
```

마찬가지로 행렬 곱셈과 벡터 대 행렬 곱셈을 할 수 있습니다.

```
mat4 a = ???;
mat4 b = ???;
mat4 c = a * b;

vec4 v = ???;
vec4 y = c * v;
```

또한 `vec` 부분에 대한 다양한 선택자가 있습니다. `vec4`를 보면,

```
vec4 v;
```

- `v.x`는 `v.s`와 `v.r`과 `v[0]`과 같습니다.
- `v.y`는 `v.t`와 `v.g`와 `v[1]`과 같습니다.
- `v.z`는 `v.p`와 `v.b`와 `v[2]`와 같습니다.
- `v.w`는 `v.q`와 `v.a`와 `v[3]`과 같습니다.

`vec` 컴포넌트를 스위즐링 할 수 있는데 이는 컴포넌트를 교환하거나 반복할 수 있다는 걸 뜻합니다.

```
v.yyyy
```

이건 다음과 같고,

```
vec4(v.y, v.y, v.y, v.y)
```

마찬가지로,

```
v.bgra
```

이건 다음과 같으며,

```
vec4(v.b, v.g, v.r, v.a)
```

vec이나 mat을 만들 때 한 번에 여러 파트를 제공할 수도 있습니다. 예를 들어 이것은,

```
vec4(v.rgb, 1)
```

다음과 같고,

```
vec4(v.r, v.g, v.b, 1)
```

또한 이것은,

```
vec4(1)
```

다음과 같은데,

```
vec4(1, 1, 1, 1)
```

한 가지 주의해야할 점은 GLSL의 타입이 매우 엄격하다는 겁니다.

```
float f = 1; // ERROR: 1은 int입니다. float에는 int를 할당할 수 없습니다.
```

올바른 방법은 다음 중 하나입니다.

```
float f = 1.0;      // float 사용
float f = float(1) // integer를 float로 캐스팅
```

위 예제의 `vec4(v.rgb, 1)`는 `vec4`가 내부에서 `float(1)`처럼 캐스팅하기 때문에 1에 대해 문제가 발생하지 않습니다.

GLSL은 많은 내장 함수들을 가지고 있는데요. 대부분은 여러 컴포넌트에서 한 번에 작동합니다. 예를 들어,

```
T sin(T angle)
```

`T`는 `float`, `vec2`, `vec3`, `vec4`가 될 수 있음을 뜻합니다. 만약 `vec4`를 전달하면 각 컴포넌트의 사인 값인 `vec4`를 돌려받습니다. 다시 말해 `v`가 `vec4`라면,

```
vec4 s = sin(v);
```

다음과 같습니다.

```
vec4 s = vec4(sin(v.x), sin(v.y), sin(v.z), sin(v.w));
```

가끔은 한 매개변수가 `float`고 나머지는 `T`가 됩니다. 이는 모든 컴포넌트에 `float`가 적용된다 는 걸 뜻하는데요. 예를 들어 `v1`과 `v2`가 `vec4`이고 `f`는 `float`라면,

```
vec4 m = mix(v1, v2, f);
```

다음과 같습니다.

```
vec4 m = vec4(
    mix(v1.x, v2.x, f),
    mix(v1.y, v2.y, f),
    mix(v1.z, v2.z, f),
    mix(v1.w, v2.w, f)
);
```

[WebGL 레퍼런스 카드](#)의 마지막 페이지에서 모든 GLSL 함수 목록을 볼 수 있습니다. 만약 정말 재미없고 장황한 걸 좋아한다면 [GLSL 명세서](#)에 도전해볼 수도 있습니다.

총정리

이게 바로 모든 글들의 핵심입니다. WebGL은 다양한 셔이더를 생성하고, 데이터를 셔이더에 제공한 다음, `gl.drawArrays` 나 `gl.drawElements` 를 호출하여 WebGL이 정점을 처리하도록 각 정점에 대한 현재 정점 셔이더를 호출한 뒤, 각 픽셀에 대한 현재 프래그먼트 셔이더를 호출하여 픽셀을 렌더링하는 것에 관한 모든 것입니다.

실제로 셔이더를 생성하려면 여러 줄의 코드가 필요합니다. 이 코드들은 대부분의 WebGL 프로그램에서 똑같기 때문에 한 번 작성한 후에는 거의 생략할 수 있습니다. GLSL 셔이더를 컴파일하고 셔이더 프로그램에 연결하는 방법은 [여기](#)에서 다룹니다.

여기에서 막 시작했다면 두 가지 방향으로 갈 수 있는데요. 이미지 처리에 관심있다면 [2D 이미지 처리 방법](#)을 보시면 됩니다. 평행 이동, 회전, 스케일 그리고 최종적으로 3D를 공부하는데 관심있다면 [여기](#)부터 시작해주세요.