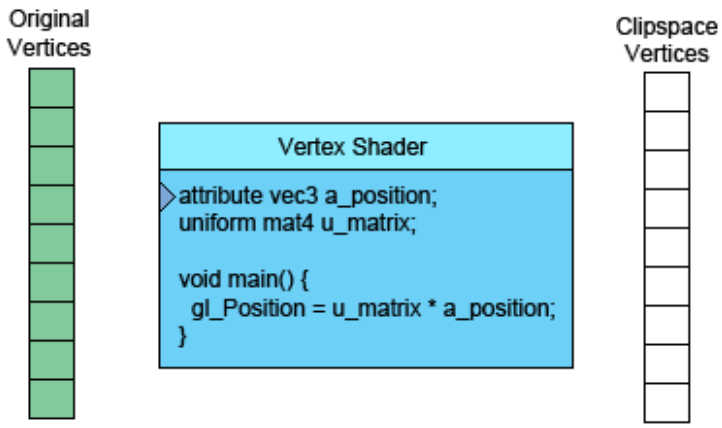
WebGL 작동 방식

이건 WebGL 기초에서 이어지는 글입니다. 이어서 하기 전에 WebGL과 GPU가 실제로 무엇을 하는지 기본적인 수준에서 얘기해봅시다. GPU에는 기본적으로 2가지 파트가 있는데요. 첫 번째 파트는 정점(또는 데이터 스트림)을 클립 공간의 정점으로 처리합니다. 두 번째 파트는 첫 번째 부분을 기반으로 픽셀을 그립니다.

```
var primitiveType = gl.TRIANGLES;
var offset = 0;
var count = 9;
gl.drawArrays(primitiveType, offset, count);
```

위와 같이 호출할 때 9는 "정점 9개 처리"를 의미하고 다음과 같이 9개의 정점이 처리되고 있습니다.



왼쪽은 여러분이 제공한 데이터입니다. 정점 셰이더는 GLSL로 작성하는 함수인데요. 이 함수는 각 정점마다 한 번씩 호출됩니다. 몇 가지 계산한 다음 현재 정점의 클립 공간 값으로 특수 변수 `gl_Position`을 설정하죠. GPU는 이 값을 가져와서 내부에 저장합니다.

`TRIANGLES`를 그린다고 가정하면, 첫 번째 부분에서 정점 3개를 생성할 때마다 GPU는 이걸 이용해 삼각형을 만듭니다. 어떤 픽셀이 삼각형의 점 3개에 해당하는지 확인한 다음, 삼각형을 래스터화(="픽셀로 그리기")하는데요. 각 픽셀마다 프래그먼트 셰이더를 호출해서 어떤 색상으로 만들지 묻습니다. 프래그먼트 셰이더는 특수 변수 `gl_FragColor`를 해당 픽셀에 원하는 색상으로 설정해야 합니다.

지금까지 예제에서 볼 수 있듯이 프래그먼트 셰이더는 픽셀마다 아주 적은 정보를 가지고 있는데요. 다행히 더 많은 정보를 전달할 수 있습니다. 정점 셰이더에서 프래그먼트 셰이더로 전달하려

는 각 값에 대해 “베링”을 정의하는 겁니다.

간단한 예시로 우리가 직접 계산한 클립 공간 좌표를 정점 셰이더에서 프래그먼트 셰이더로 전달해봅시다.

간단한 삼각형을 그려보려고 하는데요. 이전 예제에 이어서 사각형을 삼각형으로 바꿔봅시다.

```
// 삼각형을 정의한 값으로 버퍼 채우기
function setGeometry(gl) {
  gl.bufferData(
      gl.ARRAY_BUFFER,
      new Float32Array([
           0, -100,
         150,  125,
        -175,  100
      ]),
      gl.STATIC_DRAW
  );
}
```

그리고 3개의 정점만 그리면 됩니다.

```
// 장면 그리기
function drawScene() {
  ...
  // 지오메트리 그리기
  var primitiveType = gl.TRIANGLES;
  var offset = 0;
  var count = 3;
  gl.drawArrays(primitiveType, offset, count);
}
```

다음은 프래그먼트 셰이더로 데이터를 전달하기 위해 정점 셰이더에 *베링*을 선언합니다.

```
varying vec4 v_color;
...
void main() {
  // 위치에 행렬 곱하기
  gl_Position = vec4((u_matrix * vec3(a_position, 1)).xy, 0, 1);

  // 클립 공간에서 색상 공간으로 변환
  // 클립 공간은 -1.0에서 +1.0까지
  // 색상 공간은 0.0에서 1.0까지
  v_color = gl_Position * 0.5 + 0.5;
}
```

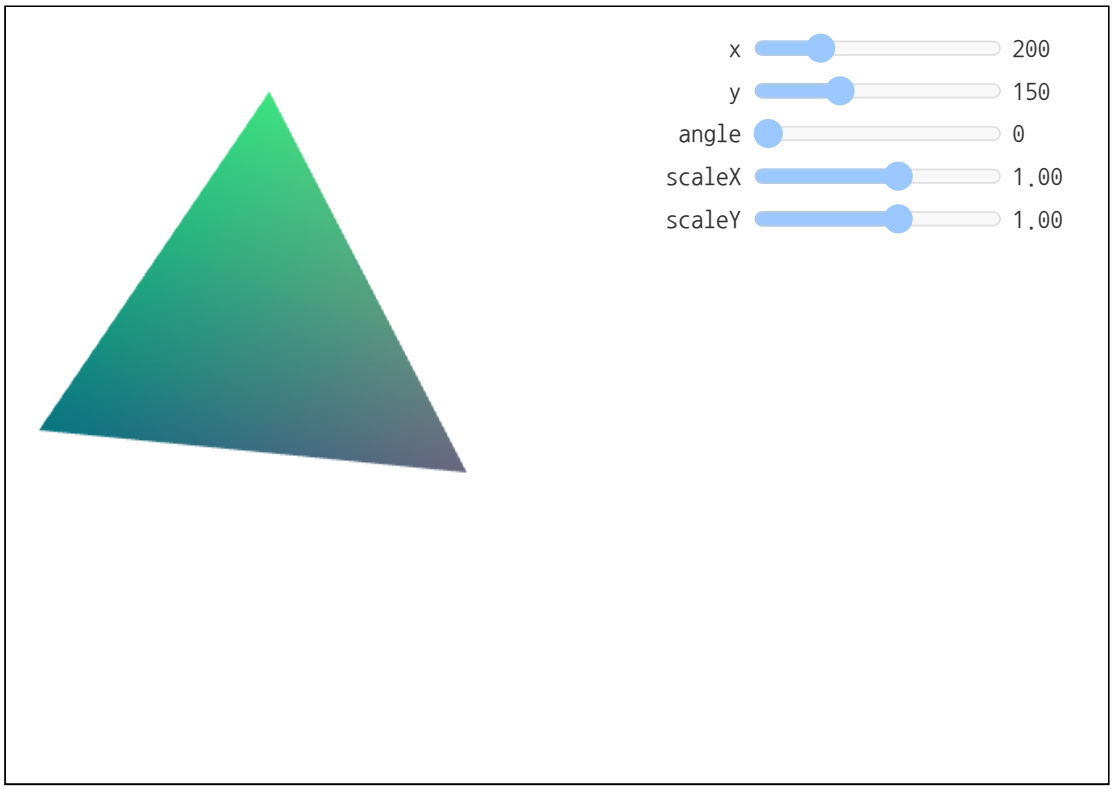
그런 다음 프래그먼트 셰이더에 동일한 *베링*을 선언합니다.

```
precision mediump float;

varying vec4 v_color;

void main() {
  gl_FragColor = v_color;
```

```
}
```

WebGL은 정점 셰이더의 베링을 이름과 타입이 같은 프래그먼트 셰이더의 베링으로 연결할 겁니다.

다음은 작동하는 버전입니다.



[새 창을 열려면 여기를 클릭]

삼각형을 이동시키고, 크기를 바꾸고, 회전시켜보세요. 참고로 색상은 클립 공간에서 계산되므로 삼각형과 함께 움직이지 않는데요. 색상은 배경에 상대적입니다.

이제 생각해봅시다. 우리는 정점 3개만을 계산했습니다. 정점 셰이더는 3번만 호출되므로 3개의 색상만을 계산하지만 삼각형은 여러 색상입니다. 이게 *베링*이라고 불리는 이유죠.

WebGL은 각 정점을 계산한 3개의 값을 가져오고 삼각형을 래스터화할 때 계산된 정점들 사이를 보간하는데요. 각 픽셀마다 해당 픽셀에 대해 보간된 값으로 프래그먼트 셰이더를 호출합니다.

위 예제에서는 3개의 정점으로 시작합니다.

정점	
0	-100
150	125
-175	100

정점 셰이더는 `translation`, `rotation`, `scale`에 행렬을 적용하고 클립 공간으로 변환합니다. `translation`, `rotation`, `scale`의 기본값은 translation = 200, 150, rotation = 0,

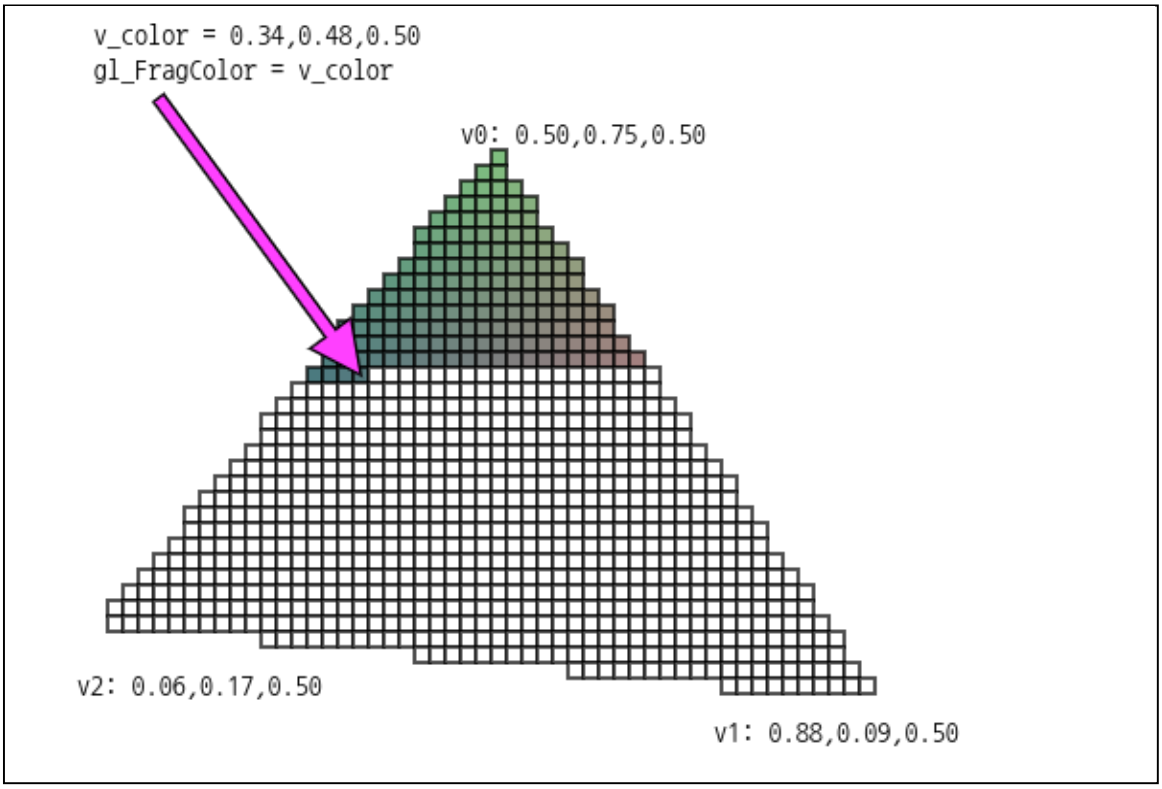
scale = 1,1이므로 실제로는 이동만 하는데요. 400x300인 백버퍼가 주어지면 정점 셰이더는 행렬을 적용한 뒤 다음과 같은 3개의 클립 공간 정점을 계산합니다.

gl_Position에 작성된 값들	
0.000	0.660
0.750	-0.830
-0.875	-0.660

또한 이걸 색상 공간으로 변환하고 우리가 선언한 *varying* `v_color` 에 작성합니다.

v_color에 작성된 값들		
0.5000	0.830	0.5
0.8750	0.086	0.5
0.0625	0.170	0.5

v_color에 작성된 3개의 값들은 보간되어 각 픽셀에 대한 프래그먼트 셰이더로 전달됩니다.



v_color는 v0, v1, v2 사이에서 보간

또한 더 많은 데이터를 정점 셰이더에 전달하여 프래그먼트 셰이더에 전달할 수 있습니다. 예를 들어 2가지 색상을 가진 삼각형 2개로 이루어진 사각형을 그린다고 가정해봅시다. 이를 위해 정점 셰이더에 또 다른 속성을 추가하면, 더 많은 데이터를 전달할 수 있고, 그 데이터를 프래그먼트 셰이더에 직접 전달할 수 있습니다.

```
attribute vec2 a_position;
attribute vec4 a_color;
```

```
...
varying vec4 v_color;

void main() {
  ...
  // 속성에서 베링으로 색상 복사
  v_color = a_color;
}
```

이제 WebGL이 사용할 색상을 제공해줘야 합니다.

```
// 정점 데이터가 필요한 곳 탐색
var positionLocation = gl.getAttribLocation(program, "a_position");
var colorLocation = gl.getAttribLocation(program, "a_color");
...
// 색상을 위한 버퍼 생성
var colorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
// 색상 설정
setColors(gl);
...

// 사각형을 만드는 두 삼각형의 색상으로 버퍼 채우기
function setColors(gl) {
  // 2개의 색상을 랜덤하게 선택
  var r1 = Math.random();
  var b1 = Math.random();
  var g1 = Math.random();

  var r2 = Math.random();
  var b2 = Math.random();
  var g2 = Math.random();

  gl.bufferData(
    gl.ARRAY_BUFFER,
    new Float32Array([
      r1, b1, g1, 1,
      r1, b1, g1, 1,
      r1, b1, g1, 1,
      r2, b2, g2, 1,
      r2, b2, g2, 1,
      r2, b2, g2, 1
    ]),
    gl.STATIC_DRAW
  );
}
```

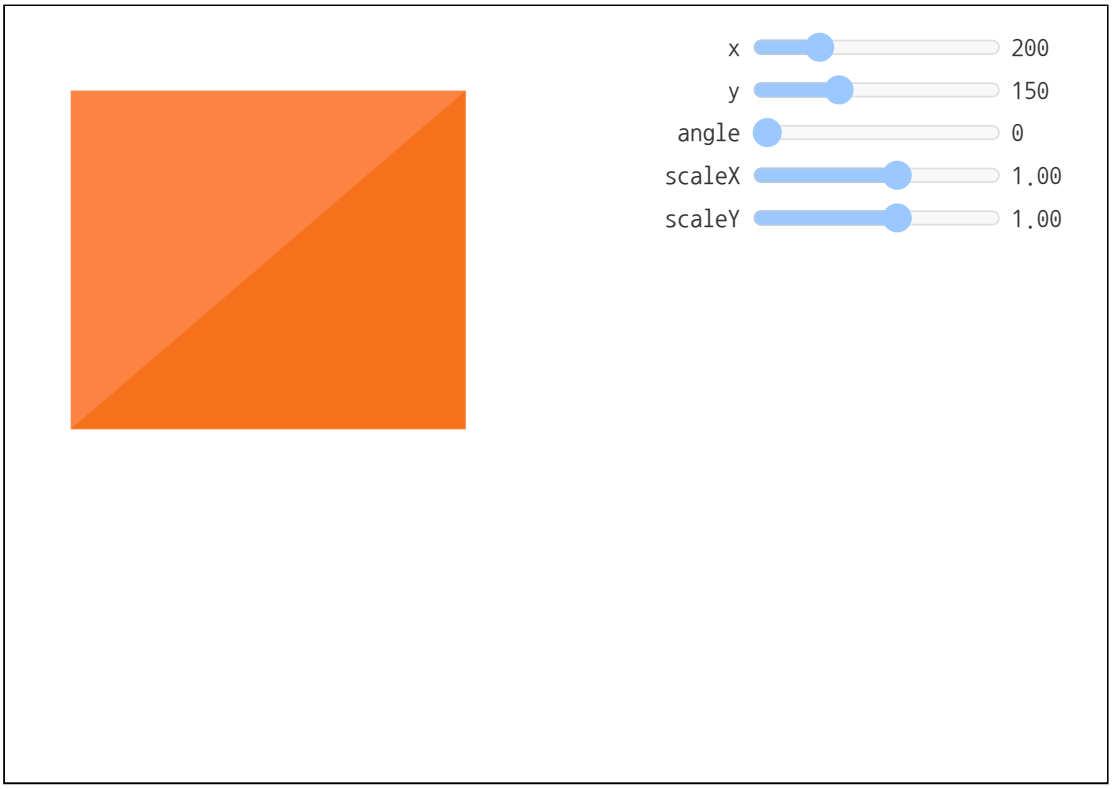
렌더링할 때 색상 속성을 설정합니다.

```
gl.enableVertexAttribArray(colorLocation);

// 색상 버퍼 할당
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);

// colorBuffer(ARRAY_BUFFER)의 데이터를 가져오는 방법을 색상 속성에 지시
var size = 4;          // 반복마다 4개의 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(colorLocation, size, type, normalize, stride, offset);
```

삼각형 2개의 정점 6개를 계산하기 위해 count 를 조정합니다.

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

그리고 여기 결과물입니다.

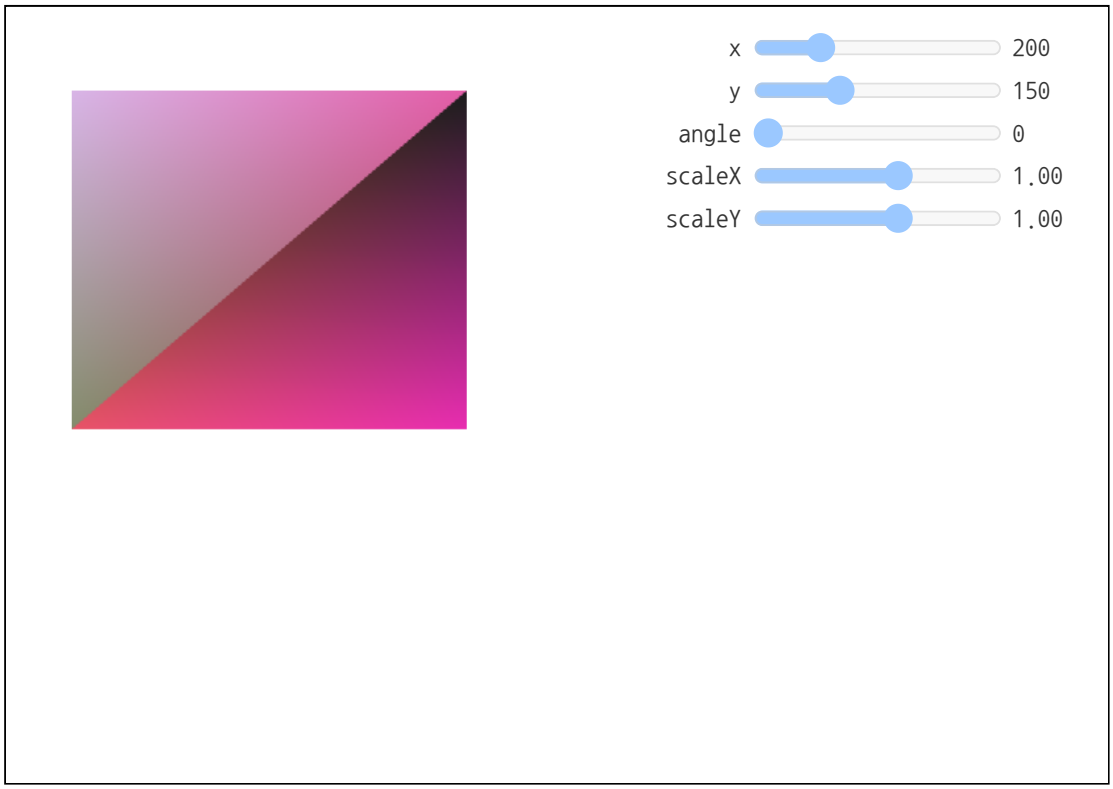


새 창을 열려면 여기를 클릭

2개의 단색 삼각형이라는 점에 주목해봅시다. 베링에 값이 전달되기 때문에 삼각형을 가로질러 변형되거나 보간되고 있는데요. 그럼에도 단색인 것은 각 삼각형의 정점 3개에 모두 같은 색상을 사용했기 때문입니다. 만약 각각의 색상을 다르게 만들면 보간된 걸 볼 수 있습니다.

```
// 사각형을 만드는 두 삼각형의 색상으로 버퍼 채우기
function setColors(gl) {
  // 모든 정점을 다른 색상으로 만들기
  gl.bufferData(
    gl.ARRAY_BUFFER,
    new Float32Array([
      Math.random(), Math.random(), Math.random(), 1,
      Math.random(), Math.random(), Math.random(), 1,
      Math.random(), Math.random(), Math.random(), 1,
      Math.random(), Math.random(), Math.random(), 1,
      Math.random(), Math.random(), Math.random(), 1,
      Math.random(), Math.random(), Math.random(), 1
    ]),
    gl.STATIC_DRAW
  );
}
```

이제 보간된 *베링*을 봅시다.



새 창을 열려면 여기를 클릭

2개 이상의 속성을 사용하고 데이터를 정점 셰이더에서 프래그먼트 셰이더로 전달하는 걸 보여주는데요. 이미지 처리 예제를 살펴보면 텍스처 좌표를 전달하기 위해 마찬가지로 추가적인 속성을 사용하는 것을 볼 수 있습니다.

버퍼와 속성 명령은 어떤 일을 하나요?

버퍼는 정점과 각 정점의 다른 데이터를 GPU로 가져오는 방법입니다. `gl.createBuffer` 는 버퍼를 생성합니다. `gl.bindBuffer` 는 해당 버퍼를 작업할 버퍼로 설정하죠. `gl.bufferData` 는 데이터를 버퍼로 복사합니다. 이건 보통 초기화할 때 수행됩니다.

버퍼에 데이터가 있으면 어떻게 데이터를 가져와 정점 셰이더의 속성에 제공할지 WebGL에 알려줘야 합니다.

이를 위해, 먼저 속성을 할당한 위치를 WebGL에 물어봐야 하는데요. 예를 들어 위 코드에서 우리는 이렇게 위치를 찾았습니다.

```
// 정점 데이터가 어디로 가야하는지 탐색
var positionLocation = gl.getAttribLocation(program, "a_position");
var colorLocation = gl.getAttribLocation(program, "a_color");
```

이것도 보통 초기화할 때 수행됩니다.

속성의 위치를 알았으면 그리기 전에 3가지 명령어를 실행해야 합니다.

```
gl.enableVertexAttribArray(location);
```

이 명령어는 버퍼에서 데이터를 제공하길 원한다고 WebGL에 알려줍니다.

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

이 명령어는 ARRAY_BUFFER 바인드 포인트에 버퍼를 할당합니다. 이건 WebGL 내부에 있는 전역 변수입니다.

```
gl.vertexAttribPointer(
  location,
  numComponents,
  typeOfData,
  normalizeFlag,
  strideToNextPieceOfData,
  offsetIntoBuffer
);
```

그리고 이 명령어는 현재 ARRAY_BUFFER 바인드 포인트에 바인딩된 버퍼에서 데이터를 가져오기 위해, 정점마다 얼마나 많은 컴포넌트(1 - 4)가 있는지, 데이터 타입(`BYTE`, `FLOAT`, `INT`, `UNSIGNED_SHORT`, 등등...)은 무엇인지, 스트라이드는 한 데이터에서 다음 데이터를 가져오기 위해 몇 바이트를 건너뛰어야 하는지, 오프셋은 버퍼에서 데이터가 얼마나 멀리 있는지 등을 WebGL에 알려줍니다.

컴포넌트의 숫자는 항상 1에서 4까지의 범위를 가집니다.

만약 데이터의 타입마다 1개의 버퍼를 쓴다면 스트라이드와 오프셋은 항상 0일 수 있습니다. 스트라이드가 0이면 "타입 크기에 맞는 스트라이드 사용"을 의미합니다. 오프셋이 0이면 "버퍼의 처음부터 시작"을 의미합니다. 0 이외의 다른 값으로 설정하는 건 더욱 복잡하고 성능 면에서 어느 정도 이점이 있긴 하지만, WebGL을 한계까지 몰아붙이기 위한 게 아니라면 복잡함을 감수할 만한 가치는 없을 것 같습니다.

버퍼와 속성이 무엇인지 정리되셨기를 바랍니다.

WebGL의 작동 방식을 이해하는 또 다른 방법으로 대화형 상태 다이어그램을 살펴보실 수 있습니다.

다음은 셰이더와 GLSL을 살펴보겠습니다.

vertexAttribPointer에서 normalizeFlag는 뭔가요?

정규화 플래그는 부동 소수점이 아닌 모든 타입을 위한 것입니다. `false`를 넘기면 해당 값의 타입으로 해석됩니다. BYTE는 -128에서 127까지, UNSIGNED_BYTE는 0부터 255까지, SHORT는 -32768부터 32767까지 등등...

정규화 플래그를 `true`로 설정하면 BYTE(-128 ~ 127) 값은 -1.0에서 +1.0사이가 되고, UNSIGNED_BYTE(0 ~ 255)는 0.0에서 +1.0사이가 됩니다. 정규화된 SHORT도 -1.0에서 +1.0사이가 되며 BYTE보다 더 높은 해상도를 가집니다.

정규화된 데이터의 가장 일반적인 용도는 색상입니다. 대부분의 경우 색상은 0.0에서 1.0 사이의 값을 가지는데요. 빨강, 초록, 파랑, 투명도 각각에 대해 전체 부동 소수점을 사용하며 각 정점의 색상마다 16바이트를 사용합니다. 복잡한 지오메트리가 있는 경우 많은 바이트가 추가될 수 있습니다. 대신에 0은 0.0이 되고 255는 1.0이 되는 UNSIGNED_BYTE로 색상을 변환할 수 있는데요. 그러면 각 정점의 색상마다 4바이트만 써서, 75%를 아낄 수 있습니다.

그렇게 하도록 코드를 수정해봅시다. 먼저 사용할 색상을 추출하는 방법을 WebGL에 지시합니다.

```
// colorBuffer(ARRAY_BUFFER)에서 데이터를 어떻게 가져올지 색상 속성에 지시
var size = 4;                 // 반복마다 4개의 컴포넌트
var type = gl.UNSIGNED_BYTE;  // 데이터는 8비트 부호없는 바이트
var normalize = true;         // 데이터 정규화
var stride = 0;               // 0 = 다음 위치를 가져오기 위해 반복마다 size * sizeof(type) 만큼 앞으로 이동
var offset = 0;               // 버퍼의 처음부터 시작
gl.vertexAttribPointer(colorLocation, size, type, normalize, stride, offset);
```

그리고 사용할 색상으로 버퍼를 채웁니다.

```
// 사각형을 만드는 두 삼각형의 색상으로 버퍼 채우기
function setColors(gl) {
  // 2개의 색상을 랜덤하게 선택
  var r1 = Math.random() * 256; // 0에서
  var b1 = Math.random() * 256; // 255.99999사이의
  var g1 = Math.random() * 256; // 값은
  var r2 = Math.random() * 256; // Uint8Array에
  var b2 = Math.random() * 256; // 저장될 때
  var g2 = Math.random() * 256; // 잘림

  gl.bufferData(
    gl.ARRAY_BUFFER,
    new Uint8Array([   // Uint8Array
      r1, b1, g1, 255,
      r1, b1, g1, 255,
      r1, b1, g1, 255,
      r2, b2, g2, 255,
      r2, b2, g2, 255,
      r2, b2, g2, 255
    ]),
    gl.STATIC_DRAW
  );
}
```

다음은 해당 샘플입니다.

x 200
y 150
angle 0
scaleX 1.00
scaleY 1.00

[새 창을 열려면 여기를 클릭](#)