WebGL - 3D 직교 투영

이 포스트는 WebGL 관련 시리즈에서 이어집니다. 첫 번째는 기초로 시작했고, 이전에는 2D 행렬에 관한 것이었습니다. 아직 읽지 않으셨다면 해당 글들을 먼저 읽어주세요.

마지막 포스트에서 우리는 2D 행렬이 어떻게 동작하는지 살펴봤습니다. 평행 이동, 회전, 스케일링, 그리고 픽셀에서 클립 공간으로 투영하는 것까지 하나의 행렬로 처리할 수 있는 방법에 대해 얘기했었는데요. 거기서 한 걸음만 더 나아가면 3D를 할 수 있습니다.

2D 예제에서는 3x3 행렬을 곱한 2D 포인트(x, y)를 가졌었는데요. 3D를 수행하기 위해서는 3D 포인트(x, y, z)와 4x4 행렬이 필요합니다.

마지막 예제를 가져와서 3D로 바꿔봅시다. 다시 F를 사용하지만 이번엔 3D 'F'입니다.

먼저 해야할 일은 3D를 다루기 위해 정점 셰이더를 수정하는 겁니다. 다음은 기존 정점 셰이더입니다.

```
<script id="vertex-shader-2d" type="x-shader/x-vertex">
attribute vec2 a_position;

uniform mat3 u_matrix;

void main() {
  // 위치에 행렬 곱하기
  gl_Position = vec4((u_matrix * vec3(a_position, 1)).xy, 0, 1);
}
</script>
```

이게 새로운 셰이더입니다.

```
<script id="vertex-shader-3d" type="x-shader/x-vertex">
attribute vec4 a_position;

uniform mat4 u_matrix;

void main() {
  // 위치에 행렬 곱하기
  gl_Position = u_matrix * a_position;
}
</script>
```

한층 더 간단해졌습니다! 2D에서 `x`와 `y`를 제공한 뒤 `z`를 1로 설정한 것처럼, 3D에서는 `x`, `y`, `z`를 제공하고 `w`가 1이어야 하지만, `w`의 기본값이 1이라는 사실을 활용합시다.

다음으로 3D 데이터를 제공해야 합니다.

```
  ...

  // positionBuffer(ARRAY_BUFFER)에서 데이터 가져오는 방법을 속성에 지시
  var size = 3;          // 반복마다 3개의 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);

  ...

// 현재 ARRAY_BUFFER 버퍼 채우기
// 문자 'F'를 정의하는 값으로 버퍼 채우기
function setGeometry(gl) {
  gl.bufferData(
    gl.ARRAY_BUFFER,
    new Float32Array([
      // 왼쪽 열
       0,   0, 0,
      30,   0, 0,
       0, 150, 0,
       0, 150, 0,
      30,   0, 0,
      30, 150, 0,

      // 상단 가로 획
       30,  0, 0,
      100,  0, 0,
       30, 30, 0,
       30, 30, 0,
      100,  0, 0,
      100, 30, 0,

      // 중간 가로 획
      30, 60, 0,
      67, 60, 0,
      30, 90, 0,
      30, 90, 0,
      67, 60, 0,
      67, 90, 0
    ]),
    gl.STATIC_DRAW
  );
}
```

다음으로 모든 행렬 함수를 2D에서 3D로 바꿔야 합니다.

여기 `m3.translation`, `m3.rotation`, `m3.scaling` 의 2D 버전입니다.

```
var m3 = {
  translation: function translation(tx, ty) {
    return [
      1, 0, 0,
      0, 1, 0,
      tx, ty, 1
    ];
  },

  rotation: function rotation(angleInRadians) {
    var c = Math.cos(angleInRadians);
    var s = Math.sin(angleInRadians);
```

```
    return [
      c, -s, 0,
      s, c, 0,
      0, 0, 1
    ];
  },

  scaling: function scaling(sx, sy) {
    return [
      sx, 0, 0,
      0, sy, 0,
      0, 0, 1
    ];
  },
};
```

그리고 다음은 3D 버전입니다.

```
var m4 = {
  translation: function(tx, ty, tz) {
    return [
       1,  0,  0,  0,
       0,  1,  0,  0,
       0,  0,  1,  0,
       tx, ty, tz, 1,
    ];
  },

  xRotation: function(angleInRadians) {
    var c = Math.cos(angleInRadians);
    var s = Math.sin(angleInRadians);

    return [
      1, 0, 0, 0,
      0, c, s, 0,
      0, -s, c, 0,
      0, 0, 0, 1,
    ];
  },

  yRotation: function(angleInRadians) {
    var c = Math.cos(angleInRadians);
    var s = Math.sin(angleInRadians);

    return [
      c, 0, -s, 0,
      0, 1, 0, 0,
      s, 0, c, 0,
      0, 0, 0, 1,
    ];
  },

  zRotation: function(angleInRadians) {
    var c = Math.cos(angleInRadians);
    var s = Math.sin(angleInRadians);

    return [
       c, s, 0, 0,
      -s, c, 0, 0,
       0, 0, 1, 0,
       0, 0, 0, 1,
    ];
  },

  scaling: function(sx, sy, sz) {
    return [
      sx, 0,  0,  0,
```

```
      0, sy,  0,  0,
      0,  0, sz,  0,
      0,  0,  0,  1,
    ];
  },
};
```

이제 3가지의 회전 함수를 가진다는 점을 주목하세요. 2D에서는 Z축을 중심으로만 회전했기 때문에 하나만 필요했는데요. 3D를 수행하기 위해서는 X축과 Y축을 중심으로도 회전되어야 합니다. 보면 아시겠지만 모든 게 굉장히 비슷한데요. 이걸 이해하면 이전처럼 단순화할 수 있습니다.

Z 회전

$$newX = x * c + y * s;$$
$$newY = x * -s + y * c;$$

Y 회전

$$newX = x * c + z * s;$$
$$newZ = x * -s + z * c;$$

X 회전

$$newY = y * c + z * s;$$
$$newZ = y * -s + z * c;$$

위 수식들을 사용하면 이렇게 회전합니다.

x axis rotation y axis rotation z axis rotation

마찬가지로 단순화된 함수들을 만들어 봅시다.

```
  translate: function(m, tx, ty, tz) {
    return m4.multiply(m, m4.translation(tx, ty, tz));
  },

  xRotate: function(m, angleInRadians) {
    return m4.multiply(m, m4.xRotation(angleInRadians));
  },

  yRotate: function(m, angleInRadians) {
```

```
  return m4.multiply(m, m4.yRotation(angleInRadians));
},

zRotate: function(m, angleInRadians) {
  return m4.multiply(m, m4.zRotation(angleInRadians));
},

scale: function(m, sx, sy, sz) {
  return m4.multiply(m, m4.scaling(sx, sy, sz));
},
```

그리고 4x4 행렬 곱셈 함수가 필요합니다.

```
multiply: function(a, b) {
  var b00 = b[0 * 4 + 0];
  var b01 = b[0 * 4 + 1];
  var b02 = b[0 * 4 + 2];
  var b03 = b[0 * 4 + 3];
  var b10 = b[1 * 4 + 0];
  var b11 = b[1 * 4 + 1];
  var b12 = b[1 * 4 + 2];
  var b13 = b[1 * 4 + 3];
  var b20 = b[2 * 4 + 0];
  var b21 = b[2 * 4 + 1];
  var b22 = b[2 * 4 + 2];
  var b23 = b[2 * 4 + 3];
  var b30 = b[3 * 4 + 0];
  var b31 = b[3 * 4 + 1];
  var b32 = b[3 * 4 + 2];
  var b33 = b[3 * 4 + 3];
  var a00 = a[0 * 4 + 0];
  var a01 = a[0 * 4 + 1];
  var a02 = a[0 * 4 + 2];
  var a03 = a[0 * 4 + 3];
  var a10 = a[1 * 4 + 0];
  var a11 = a[1 * 4 + 1];
  var a12 = a[1 * 4 + 2];
  var a13 = a[1 * 4 + 3];
  var a20 = a[2 * 4 + 0];
  var a21 = a[2 * 4 + 1];
  var a22 = a[2 * 4 + 2];
  var a23 = a[2 * 4 + 3];
  var a30 = a[3 * 4 + 0];
  var a31 = a[3 * 4 + 1];
  var a32 = a[3 * 4 + 2];
  var a33 = a[3 * 4 + 3];

  return [
    b00 * a00 + b01 * a10 + b02 * a20 + b03 * a30,
    b00 * a01 + b01 * a11 + b02 * a21 + b03 * a31,
    b00 * a02 + b01 * a12 + b02 * a22 + b03 * a32,
    b00 * a03 + b01 * a13 + b02 * a23 + b03 * a33,
    b10 * a00 + b11 * a10 + b12 * a20 + b13 * a30,
    b10 * a01 + b11 * a11 + b12 * a21 + b13 * a31,
    b10 * a02 + b11 * a12 + b12 * a22 + b13 * a32,
    b10 * a03 + b11 * a13 + b12 * a23 + b13 * a33,
    b20 * a00 + b21 * a10 + b22 * a20 + b23 * a30,
    b20 * a01 + b21 * a11 + b22 * a21 + b23 * a31,
    b20 * a02 + b21 * a12 + b22 * a22 + b23 * a32,
    b20 * a03 + b21 * a13 + b22 * a23 + b23 * a33,
    b30 * a00 + b31 * a10 + b32 * a20 + b33 * a30,
    b30 * a01 + b31 * a11 + b32 * a21 + b33 * a31,
    b30 * a02 + b31 * a12 + b32 * a22 + b33 * a32,
    b30 * a03 + b31 * a13 + b32 * a23 + b33 * a33,
  ];
},
```

또한 투영 함수를 업데이트해야 합니다. 다음은 픽셀에서 클립 공간으로 변환하는 기존 코드입니다.

```
  projection: function (width, height) {
    // 참고: 이 행렬은 Y축을 뒤집기 때문에 0이 상단입니다.
    return [
      2 / width, 0, 0,
      0, -2 / height, 0,
      -1, 1, 1
    ];
  },
}
```

이를 3D로 확장해 보겠습니다.

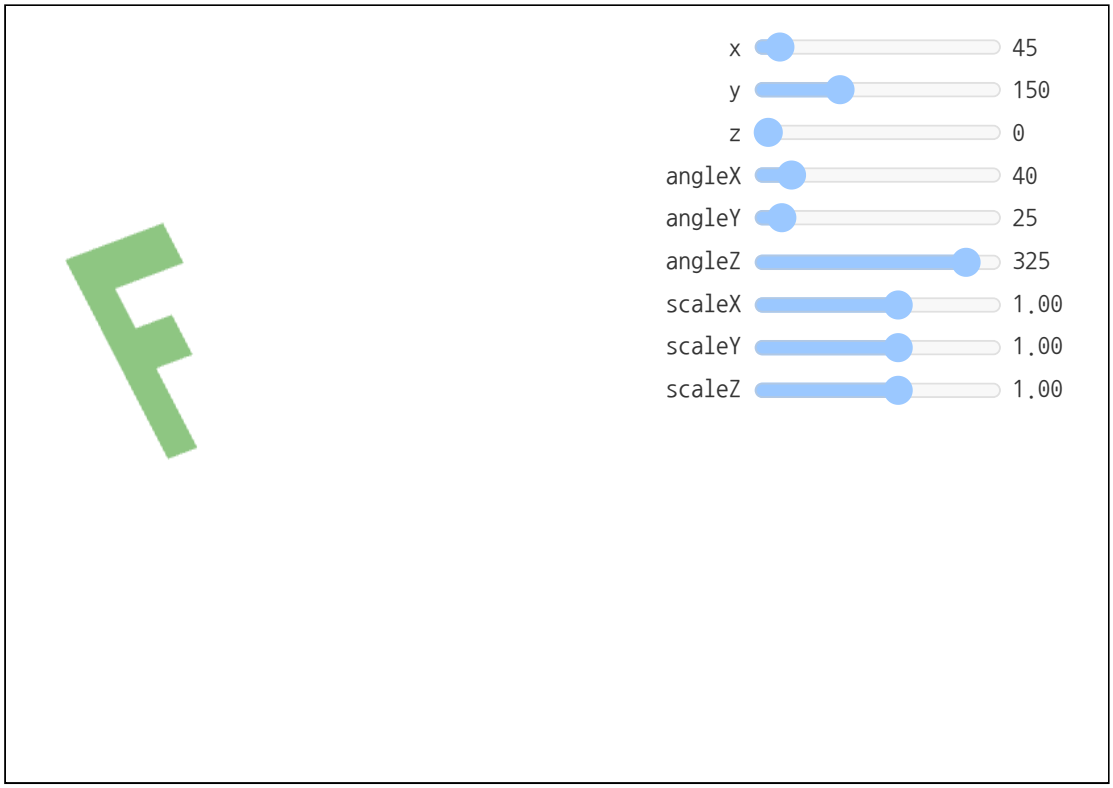
```
  projection: function(width, height, depth) {
    // 참고: 이 행렬은 Y축을 뒤집기 때문에 0이 상단입니다.
    return [
       2 / width, 0, 0, 0,
       0, -2 / height, 0, 0,
       0, 0, 2 / depth, 0,
      -1, 1, 0, 1,
    ];
  },
```

X와 Y를 픽셀 공간에서 클립 공간으로 변환해야 했던 것처럼 Z도 동일한 작업을 수행해야 합니다. 이 경우에는 Z축 픽셀 단위도 만들게 되는데요. depth 에 width 와 비슷한 값을 전달할 것이기 때문에, 너비는 0픽셀에서 width 로, 높이는 0픽셀에서 height 가 되지만, depth 는 -depth / 2 에서 +depth / 2 가 됩니다.

마지막으로 행렬을 계산하는 코드를 업데이트해야 합니다.

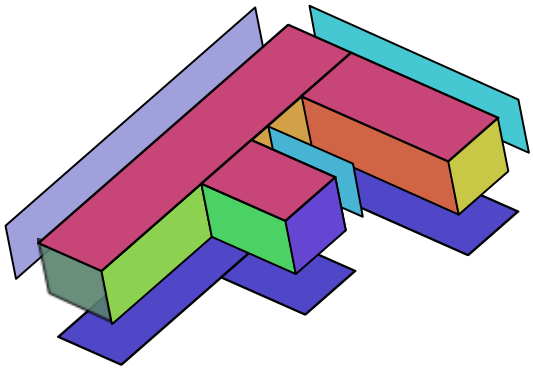
```
  // 행렬 계산
  var matrix = m4.projection(gl.canvas.clientWidth, gl.canvas.clientHeight, 400);
  matrix = m4.translate(matrix, translation[0], translation[1], translation[2]);
  matrix = m4.xRotate(matrix, rotation[0]);
  matrix = m4.yRotate(matrix, rotation[1]);
  matrix = m4.zRotate(matrix, rotation[2]);
  matrix = m4.scale(matrix, scale[0], scale[1], scale[2]);

  // 행렬 설정
  gl.uniformMatrix4fv(matrixLocation, false, matrix);
```

그리고 여기 샘플입니다.



[새 창을 열려면 여기를 클릭]

첫 번째 문제로 지오메트리가 3D로 보기 어려운 평평한 F입니다. 이걸 고치기 위해 지오메트리를 3D로 확장해봅시다. 현재 F는 삼각형 2개로 이루어진 사각형 3개로 만들어져 있습니다. 이걸 3D로 만들기 위해서는 총 16개의 사각형이 필요한데요. 사각형이 앞쪽에 3개, 뒤쪽에 3개, 왼쪽에 1개, 오른쪽에 4개, 위쪽에 2개, 아래쪽에 3개입니다.

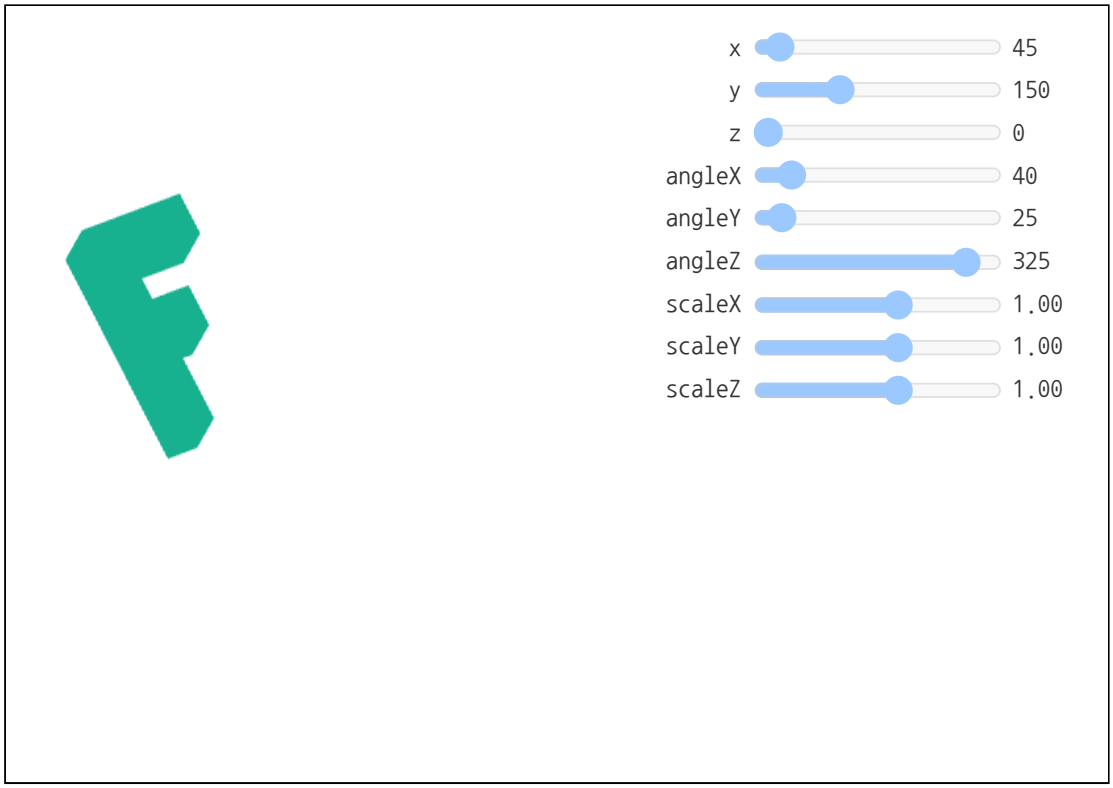


여기 나열하자니 꽤 많군요. 사각형마다 2개의 삼각형 그리고 삼각형마다 3개의 정점이 있는 사각형 16개는 96개의 정점을 가집니다. 이들 전부를 보고 싶다면 샘플의 소스 코드를 봐주세요.

더 많은 정점을 그려야 하므로 아래와 같이 수정합니다.

```
// 지오메트리 그리기
var primitiveType = gl.TRIANGLES;
var offset = 0;
var count = 16 * 6;
gl.drawArrays(primitiveType, offset, count);
```

그리고 여기 해당 버전입니다.



새 창을 열려면 여기를 클릭

슬라이더를 움직여봐도 이게 3D인지 확인하기 어려운데요. 각 사각형에 다른 색상을 칠해봅시다. 이를 위해 정점 셰이더에 또 다른 속성과 이걸 정점 셰이더에서 프래그먼트 셰이더로 전달하기 위한 베링을 추가할 겁니다.

다음은 새로운 정점 셰이더입니다.

```
<script id="vertex-shader-3d" type="x-shader/x-vertex">
attribute vec4 a_position;
attribute vec4 a_color;

uniform mat4 u_matrix;

varying vec4 v_color;

void main() {
  // 위치에 행렬 곱하기
  gl_Position = u_matrix * a_position;

  // 프래그먼트 셰이더로 색상 전달
  v_color = a_color;
}
</script>
```

프래그먼트 셰이더에서 해당 색상을 사용해야 합니다.

```
<script id="fragment-shader-3d" type="x-shader/x-fragment">
precision mediump float;

// 정점 셰이더에서 전달됩니다.
varying vec4 v_color;

void main() {
```

```
  gl_FragColor = v_color;
}
</script>
```

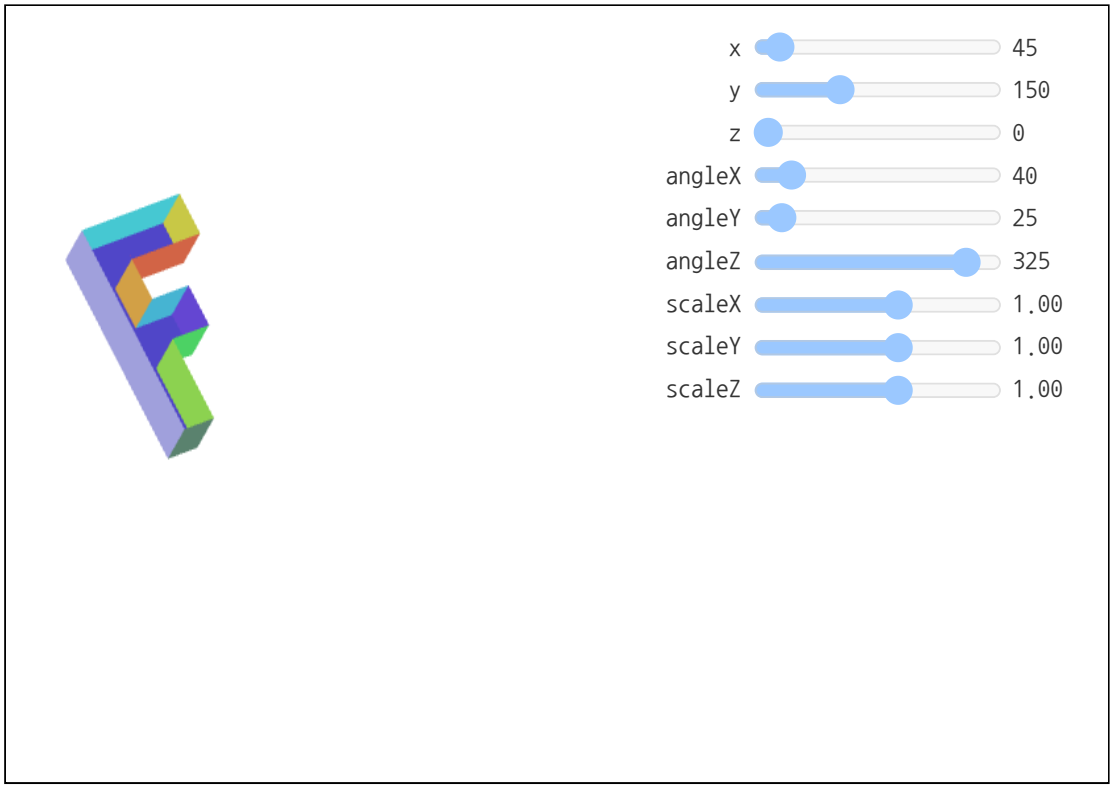
색상을 제공하려면 속성의 위치를 찾고, 색상을 지정하기 위해 또 다른 버퍼와 속성을 설정해야 합니다.

```
  ...
  var colorLocation = gl.getAttribLocation(program, "a_color");

  ...
  // 색상용 버퍼 생성
  var colorBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
  // 버퍼에 색상 넣기
  setColors(gl);


  ...

// 'F'의 색상으로 버퍼 채우기
function setColors(gl) {
  gl.bufferData(
    gl.ARRAY_BUFFER,
    new Uint8Array([
      // 왼쪽 열 앞쪽
      200, 70, 120,
      200, 70, 120,
      200, 70, 120,
      200, 70, 120,
      200, 70, 120,
      200, 70, 120,

      // 상단 획 앞쪽
      200, 70, 120,
      200, 70, 120,
      ...
      ...
    gl.STATIC_DRAW);
}
```

그런 다음 렌더링할 때 색상 버퍼에서 색상 가져오는 방법을 색상 속성에 알려줘야 합니다.

```
// 색상 속성 활성화
gl.enableVertexAttribArray(colorLocation);

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

// colorBuffer(ARRAY_BUFFER)에서 데이터 가져오는 방법을 속성에 지시
var size = 3;                 // 반복마다 3개의 컴포넌트
var type = gl.UNSIGNED_BYTE;  // 데이터는 부호없는 8비트 값
var normalize = true;         // 데이터 정규화 (0-255에서 0-1로 전환)
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);
```

이제 이런 결과를 얻게 됩니다.

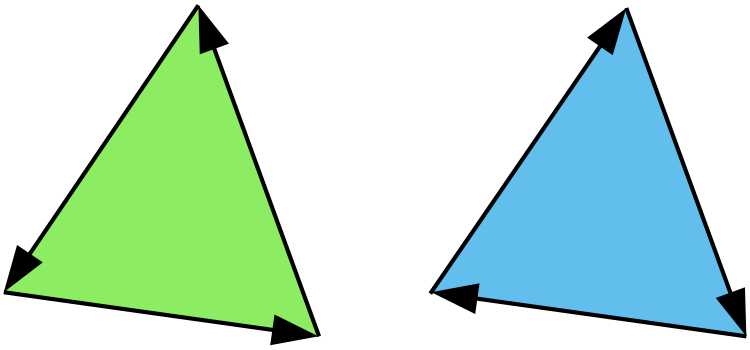


[새 창을 열려면 여기를 클릭](#)

엥 이게 뭐죠? 3D 'F'의 앞면, 뒷면, 옆면 등이 지오메트리 데이터에 나타난 순서대로 그려지고 있습니다.

붉은 부분 은 'F'의 **앞부분**이지만 데이터의 첫 부분이기 때문에 먼저 그려지고 그 뒤에 있는 다른 삼각형들은 이걸 덮어서 그려집니다. 예를 들어 보라색 부분 은 사실 'F'의 뒷부분이지만 데이터에서 두 번째로 나오기 때문에 두 번째로 그려집니다.

WebGL의 삼각형은 앞면과 뒷면의 개념을 가지고 있습니다. 기본적으로 삼각형 앞면은 반시계 방향으로 진행하는 정점을 가집니다. 삼각형 뒷면은 시계 방향으로 진행하는 정점을 가집니다.



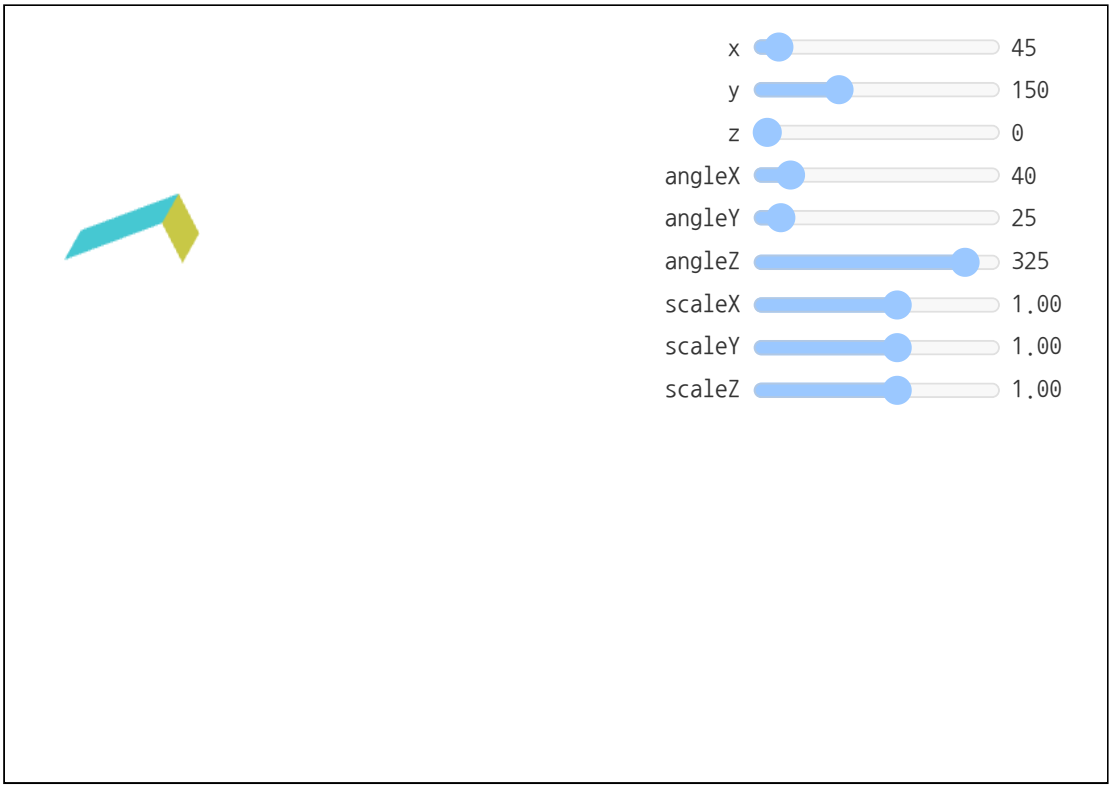
WebGL은 삼각형의 앞면 혹은 뒷면만 그릴 수도 있는데요. 이 코드로 해당 기능을 켤 수 있습니다.

```
gl.enable(gl.CULL_FACE);
```

이걸 `drawScene` 함수에 넣어줍시다. 해당 기능을 켜면 WebGL은 기본적으로 삼각형 뒷면을 "컬링"으로 설정하는데요. 이 경우 "컬링"은 "그리지 않음"을 의미하는 단어입니다.

참고로 WebGL에서 삼각형이 시계 혹은 반시계 방향으로 진행되는지는 클립 공간에 있는 해당 삼각형의 정점에 따라 달라집니다. 즉 WebGL은 정점 셰이더에서 정점에 수식을 적용한 후에 삼각형이 앞면인지 뒷면인지 파악합니다. 이건 X에서 -1로 스케일링되거나 180도 회전한 시계 방향 삼각형은 반시계 방향 삼각형이 된다는 걸 의미하는데요. CULL_FACE를 꺼놨기 때문에 시계 방향(앞면)과 반시계 방향(뒷면) 삼각형을 모두 볼 수 있었습니다. 이제 CULL_FACE를 켰기 때문에 스케일이나 회전 등의 이유로 앞면 삼각형이 뒤집히면 WebGL은 그리지 않을겁니다. 3D에서 무언가를 회전시킬 때 일반적으로 여러분을 향하는 삼각형이 앞면으로 간주되길 원하기 때문에 제법 괜찮은 기능입니다.

다음은 CULL_FACE를 켰을 때 얻게 되는 결과입니다.



[새 창을 열려면 여기를 클릭]

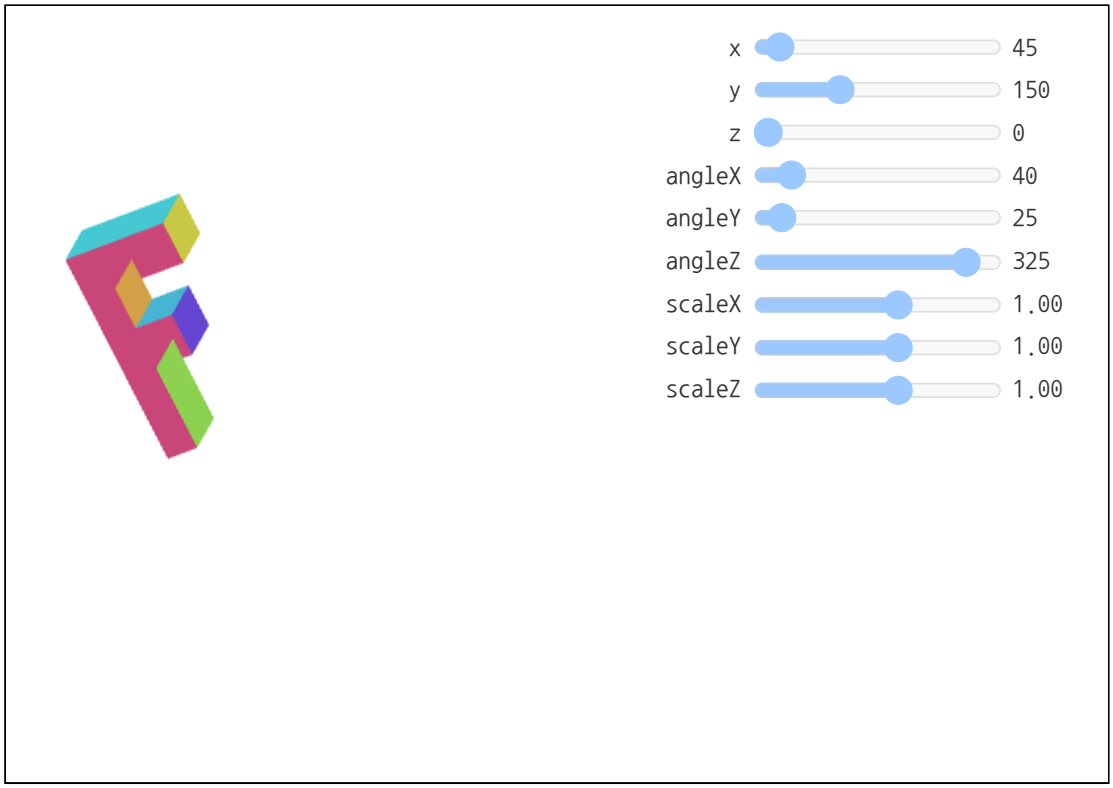
삼각형이 전부 어디로 간거죠? 알고보니 다수가 잘못된 방향을 향하고 있었습니다. 회전시켜서 다른 방향을 보면 나타나는 것을 볼 수 있는데요. 다행히 쉽게 고칠 수 있습니다. 뒤집혀 있는 부분을 찾아 정점 2개를 교환하면 됩니다. 예를 들어 뒤집힌 삼각형 하나가 있다면,

```
  1,   2,   3,
 40,  50,  60,
700, 800, 900,
```

앞을 향하도록 마지막 두 정점을 교환하면 됩니다.

```
  1,   2,   3,
700, 800, 900,
 40,  50,  60,
```

계속해서 뒤집힌 삼각형을 모두 고치면 이런 결과를 얻습니다.



새 창을 열려면 여기를 클릭

나아졌지만 아직 한 가지 문제가 더 남아있습니다. 모든 삼각형이 올바른 방향을 향하고 뒷면이 컬링된 경우에도, 뒤에 있어야 하는 삼각형이 앞에 있어야 하는 삼각형 위에 그려지는 부분이 있는데요.

DEPTH BUFFER를 입력해봅시다.

Z-버퍼라고도 불리는 깊이 버퍼는 *깊이* 픽셀로, 이미지를 만드는 데 사용되는 색상 픽셀마다 깊이 픽셀이 하나씩 존재합니다. WebGL은 각 색상 픽셀을 그리기 때문에 깊이 픽셀도 그릴 수 있는데요. 이건 Z축에 대해 정점 셰이더에서 반환한 값을 기반으로 합니다. X와 Y를 클립 공간으로 변환해야 했던 것처럼 Z도 클립 공간(-1에서 +1)에 있습니다. 해당 값은 깊이 공간 값(0에서 +1)으로 변환됩니다. WebGL은 색상 픽셀을 그리기 전에 대응하는 깊이 픽셀을 검사하는데요. 그릴 픽셀의 깊이 값이 대응하는 깊이 픽셀의 값보다 클 경우 WebGL은 새로운 색상 픽셀을 그리지 않습니다. 아니면 프래그먼트 셰이더의 색상으로 새로운 색상 픽셀을 모두 그린 다음 새로운 깊이 값으로 깊이 픽셀을 그립니다. 이는 다른 픽셀 뒤에 있는 픽셀은 그려지지 않는다는 걸 의미합니다.

컬링을 켰던 것처럼 이런 식으로 간단하게 해당 기능을 사용할 수 있습니다.

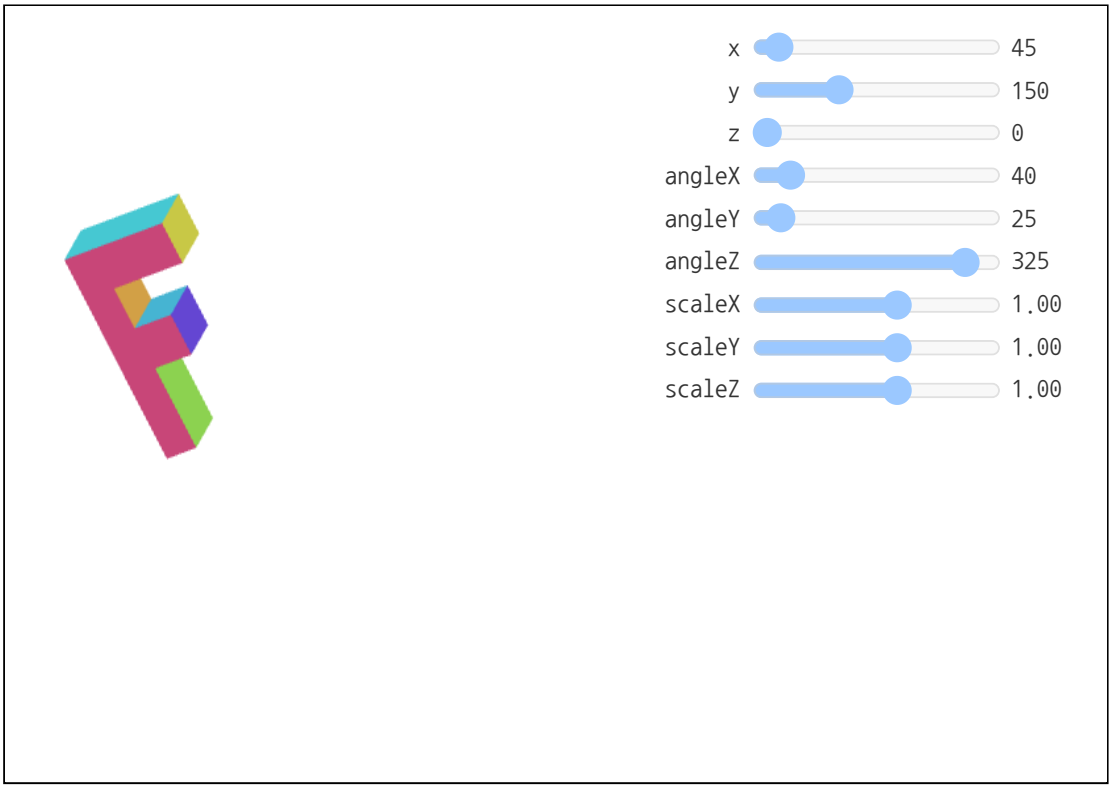
```
gl.enable(gl.DEPTH_TEST);
```

그리기를 시작하기 전에 깊이 버퍼를 1.0으로 초기화해야 합니다.

```
// 장면 그리기
function drawScene() {
  ...

  // 캔버스와 깊이 버퍼 초기화
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

```
...
```

그리고 이제 우리는 3D를 얻게 됩니다!



새 창을 열려면 여기를 클릭

한 가지 사소한 게 남았는데요. 대부분의 3D 수학 라이브러리에는 클립 공간에서 픽셀 공간으로 변환하는 `projection` 함수가 없습니다. 그보다는 보통 `ortho` 나 `orthographic` 이라 불리는 함수가 있습니다.

```
var m4 = {
  orthographic: function(left, right, bottom, top, near, far) {
    return [
      2 / (right - left), 0, 0, 0,
      0, 2 / (top - bottom), 0, 0,
      0, 0, 2 / (near - far), 0,

      (left + right) / (left - right),
      (bottom + top) / (bottom - top),
      (near + far) / (near - far),
      1,
    ];
  }
```

`width`, `height`, `depth` 등의 매개변수를 가지는 위의 단순한 `projection` 함수와 달리, 좀 더 일반적인 직교 투영 함수는 더 많은 유연성을 제공하는 `left`, `right`, `bottom`, `top`, `near`, `far` 등을 전달할 수 있는데요. 원래 쓰던 투영 함수와 동일하게 쓰기 위해서는 이렇게 호출할 수 있습니다.

```
var left = 0;
var right = gl.canvas.clientWidth;
var bottom = gl.canvas.clientHeight;
var top = 0;
var near = 400;
var far = -400;
var matrix = m4.orthographic(left, right, bottom, top, near, far);
```

다음 포스트에서는 원근감를 가지도록 만드는 방법에 대해 살펴보겠습니다.

속성이 vec4인데 gl.vertexAttribPointer의 크기는 왜 3인가요?

꼼꼼히 보시는 분들은 2개의 속성에 대해 다음과 같이 정의했다는 것을 눈치채셨을 겁니다.

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

둘 다 'vec4'지만 사용중인 버퍼에서 데이터 가져오는 방법을 이렇게 설정했는데요.

```
// positionBuffer(ARRAY_BUFFER)에서 데이터 가져오는 방법을 속성에 지시
var size = 3;          // 반복마다 3개의 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);

...
// colorBuffer(ARRAY_BUFFER)에서 데이터 가져오는 방법을 속성에 지시
var size = 3;                 // 반복마다 3개의 컴포넌트
var type = gl.UNSIGNED_BYTE;  // 데이터는 8비트 부호없는 바이트
var normalize = true;         // 데이터 정규화 (0-255에서 0-1로 전환)
var stride = 0;               // 0 = 다음 색상을 가져오기 위해 반복마다 size * sizeof(type) 만큼 앞으로 이동
var offset = 0;               // 버퍼의 처음부터 시작
gl.vertexAttribPointer(colorAttributeLocation, size, type, normalize, stride, offset);
```

해당 '3'은 정점 셰이더의 반복마다 각 속성의 버퍼에서 3개의 값만 가져오라는 뜻입니다. 이게 동작하는 이유는 정점 셰이더에서 입력하지 않는 값에 대해 WebGL이 기본값을 제공하기 때문인데요. 기본값은 0, 0, 0, 1로 x = 0, y = 0, z= 0, w = 1 입니다. 이게 기존 정점 셰이더에서 명시적으로 1을 입력해줘야 했던 이유입니다. x와 y는 전달했고, z는 1이 필요했지만, z의 기본값은 0이므로, 명시적으로 1을 입력해야 했던 거죠. 하지만 3D의 경우 'w'를 입력하지 않아도 수식이 작동하는데 필요한 값인 1을 기본값으로 가집니다.