

WebGL 2D 행렬

이 포스트는 WebGL 관련 시리즈에서 이어집니다. 첫 번째는 [기초](#)로 시작했고, 이전에는 [2D 지오메트리 스케일](#)에 관한 것이었습니다.

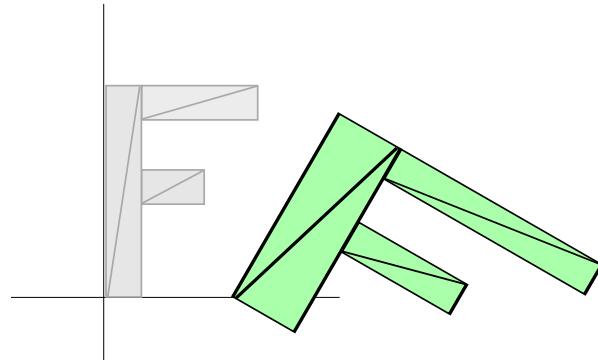
수학 vs 프로그래밍 vs WebGL

선형 대수나 행렬로 작업한 경험이 있다면 시작하기 전에 [이 글](#)을 먼저 읽어주세요.

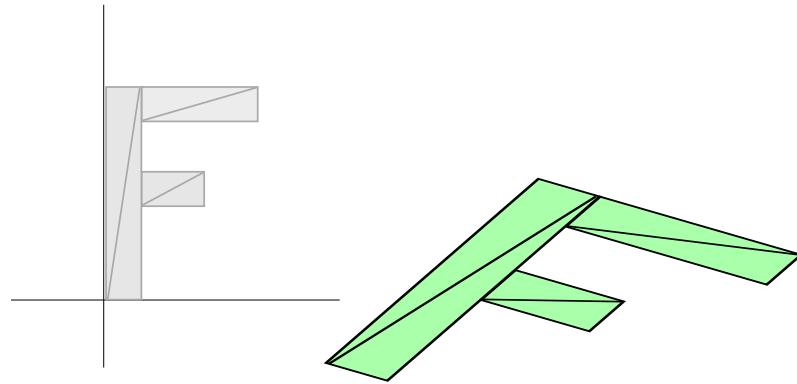
행렬에 대한 경험이 없다면 위의 링크를 건너뛰고 계속 읽으셔도 됩니다.

지난 3개의 포스트에서 우리는 [지오메트리 평행 이동](#), [지오메트리 회전](#), [지오메트리 스케일](#)에 대해 살펴봤습니다. 평행 이동, 회전, 스케일은 각각 '변환'의 한 종류로 간주되는데요. 각각의 변환은 세이더의 변경이 필요하고 3개의 변환은 각각의 순서에 따라 달라집니다. [이전 예제](#)에서는 크기를 조정하고, 회전한 다음, 평행 이동했는데요. 만약 다른 순서로 적용한다면 다른 결과를 얻게 될 겁니다.

예를 들어 $(2, 1)$ 의 스케일링, 30° 회전, $(100, 0)$ 의 평행 이동을 하면 이렇게 됩니다.



$(100, 0)$ 의 평행 이동, 30° 회전, $(2, 1)$ 의 스케일링을 하면 이렇게 됩니다.



결과는 완전히 다릅니다. 심지어 더 안 좋은 점은, 두 번째 예제가 필요하다면 우리가 원하는 새로운 순서로 평행 이동, 회전, 스케일을 적용한 다른 세이더를 작성해야 한다는 겁니다.

저보다 훨씬 더 똑똑한 사람들은 행렬 수학으로 동일한 모든 작업을 할 수 있다는 걸 밝혀냈는데요. 2D의 경우 3×3 행렬을 사용합니다. 3×3 행렬은 상자 9개가 있는 그리드와 같습니다.

1.0	2.0	3.0
4.0	5.0	6.0
7.0	8.0	9.0

계산하기 위해서 위치를 행렬의 열에 곱하고 결과를 합산합니다. 위치는 x 그리고 y, 2개의 값만을 가지지만, 계산하기 위해서는 3개의 값이 필요하므로 세 번째 값에 1을 사용할 겁니다.

이 경우 결과는 이렇게 됩니다.

$$\begin{array}{l} \text{newX} = x * \begin{matrix} 1.0 \\ 4.0 \\ 1 \end{matrix} + \quad \text{newY} = y * \begin{matrix} 2.0 \\ 5.0 \\ 1 \end{matrix} + \quad \text{extra} = 1 * \begin{matrix} 3.0 \\ 6.0 \\ 1 \end{matrix} + \\ \qquad \qquad \qquad \qquad \qquad \qquad \qquad \end{array}$$

아마 저걸 보고 "그래서 요점이 뭔데?"라고 생각하실 겁니다. 평행 이동을 있다고 가정해봅시다. 우리가 원하는 평행 이동 양을 tx와 ty라고 부를거구요. 이렇게 행렬을 만듭시다.

1.0	0.0	0.0
0.0	1.0	0.0
tx	ty	1.0

이제 확인해보면,

$$\begin{array}{l} \text{newX} = x * \begin{matrix} 1.0 \\ 0.0 \\ 1 \end{matrix} + \quad \text{newY} = y * \begin{matrix} 0.0 \\ 1.0 \\ 1 \end{matrix} + \quad \text{extra} = 1 * \begin{matrix} 0.0 \\ 0.0 \\ 1 \end{matrix} + \\ \qquad \qquad \qquad \qquad \qquad \qquad \qquad \end{array}$$

대수을 기억한다면 어느 곳이든 0으로 곱해서 지울 수 있다는 걸 아실겁니다. 1을 곱하면 아무런 효과가 없으므로 어떻게 되는지 간단하게 본다면,



$$\begin{aligned} \text{newX} &= x * \begin{pmatrix} 1.0 \\ 0.0 \end{pmatrix} + \\ &\quad y * \begin{pmatrix} 0.0 \\ 1.0 \end{pmatrix} + \\ &\quad 1 * \begin{pmatrix} tx \\ 1 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} \text{newY} &= x * \begin{pmatrix} 0.0 \\ 1.0 \end{pmatrix} + \\ &\quad y * \begin{pmatrix} 1.0 \\ 0.0 \end{pmatrix} + \\ &\quad 1 * \begin{pmatrix} ty \\ 1 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} \text{extra} &= x * \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} + \\ &\quad y * \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} + \\ &\quad 1 * \begin{pmatrix} 1.0 \\ 1 \end{pmatrix} \end{aligned}$$

혹은 더 간결하게,

$$\begin{aligned} \text{newX} &= x + tx; \\ \text{newY} &= y + ty; \end{aligned}$$

그리고 추가적으로 신경 쓸 것은 없습니다. 이건 [평행 이동 예제의 평행 이동 코드](#)와 놀라울 정도로 비슷합니다.

마찬가지로 회전을 해봅시다. 회전에 대한 글에서 강조한 것처럼 회전하고자 하는 각도의 사인과 코사인이 필요합니다.

$$\begin{aligned} s &= \text{Math.sin(angleToRotateInRadians)}; \\ c &= \text{Math.cos(angleToRotateInRadians)}; \end{aligned}$$

그리고 이렇게 행렬을 만들고,

$$\begin{pmatrix} c & -s & 0.0 \\ s & c & 0.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$$

행렬을 적용하면 이렇게 되며,

$$\begin{aligned} \text{newX} &= x * \begin{pmatrix} c \\ s \end{pmatrix} + \\ &\quad y * \begin{pmatrix} 0.0 \\ 1 \end{pmatrix} \\ &\quad 1 * \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} \end{aligned} \quad \begin{aligned} \text{newY} &= x * \begin{pmatrix} -s \\ c \end{pmatrix} + \\ &\quad y * \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &\quad 1 * \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} \end{aligned} \quad \begin{aligned} \text{extra} &= x * \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} + \\ &\quad y * \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} + \\ &\quad 1 * \begin{pmatrix} 1.0 \\ 1 \end{pmatrix} \end{aligned}$$

0과 1로 곱한 것들을 모두 검게 칠하고,

$$\begin{aligned} \text{newX} &= x * \begin{pmatrix} c \\ s \end{pmatrix} + \\ &\quad y * \begin{pmatrix} 0.0 \\ 1 \end{pmatrix} \\ &\quad 1 * \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} \end{aligned} \quad \begin{aligned} \text{newY} &= x * \begin{pmatrix} -s \\ c \end{pmatrix} + \\ &\quad y * \begin{pmatrix} 0.0 \\ 1 \end{pmatrix} \\ &\quad 1 * \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} \end{aligned} \quad \begin{aligned} \text{extra} &= x * \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} + \\ &\quad y * \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} + \\ &\quad 1 * \begin{pmatrix} 1.0 \\ 1 \end{pmatrix} \end{aligned}$$

단순화하면,

$$\begin{aligned} \text{newX} &= x * c + y * s; \\ \text{newY} &= x * -s + y * c; \end{aligned}$$

이게 바로 회전 샘플에 있는 수식입니다.

마지막으로 스케일입니다. 2개의 스케일 인수를 sx와 sy라고 부르겠습니다.

이런 행렬을 만들어,

sx	0.0	0.0
0.0	sy	0.0
0.0	0.0	1.0

행렬을 적용하면 이렇게 되며,

$$\begin{aligned} \text{newX} &= x * \boxed{\text{sx}} + \\ &y * \boxed{0.0} + \\ &1 * 0.0 \end{aligned}$$

$$\begin{aligned} \text{newY} &= x * \boxed{0.0} + \\ &y * \boxed{\text{sy}} + \\ &1 * 0.0 \end{aligned}$$

$$\begin{aligned} \text{extra} &= x * \boxed{0.0} + \\ &y * \boxed{0.0} + \\ &1 * 1.0 \end{aligned}$$

실제로는,

$$\begin{aligned} \text{newX} &= x * \boxed{\text{sx}} + \\ &y * \boxed{0.0} + \\ &1 * 0.0 \end{aligned}$$

$$\begin{aligned} \text{newY} &= x * \boxed{0.0} + \\ &y * \boxed{\text{sy}} + \\ &1 * 0.0 \end{aligned}$$

$$\begin{aligned} \text{extra} &= x * \boxed{0.0} + \\ &y * \boxed{0.0} + \\ &1 * 1.0 \end{aligned}$$

단순화하면,

$$\begin{aligned} \text{newX} &= x * \text{sx}; \\ \text{newY} &= y * \text{sy}; \end{aligned}$$

이건 스케일 샘플과 동일합니다.

아마 아직 "그래서 뭐요? 요점이 뭔데요?"라고 생각하고 계실 것 같습니다. 그냥 이미 하고 있던 동일한 것들을 수행하기 위한 여러 작업처럼 보입니다.

여기가 마법이 들어오는 곳입니다. 행렬을 함께 곱하고 모든 변환을 한 번에 적용하는거죠. 두 행렬을 받아서 곱하고 결과를 반환하는 함수 m3.multiply 가 있다고 가정해봅시다.

```
var m3 = {
  multiply: function(a, b) {
    var a00 = a[0] * 3 + 0;
    var a01 = a[0] * 3 + 1;
    var a02 = a[0] * 3 + 2;
    var a10 = a[1] * 3 + 0;
    var a11 = a[1] * 3 + 1;
    var a12 = a[1] * 3 + 2;
    var a20 = a[2] * 3 + 0;
    var a21 = a[2] * 3 + 1;
    var a22 = a[2] * 3 + 2;
  }
};
```

```

var b00 = b[0 * 3 + 0];
var b01 = b[0 * 3 + 1];
var b02 = b[0 * 3 + 2];
var b10 = b[1 * 3 + 0];
var b11 = b[1 * 3 + 1];
var b12 = b[1 * 3 + 2];
var b20 = b[2 * 3 + 0];
var b21 = b[2 * 3 + 1];
var b22 = b[2 * 3 + 2];

return [
  b00 * a00 + b01 * a10 + b02 * a20,
  b00 * a01 + b01 * a11 + b02 * a21,
  b00 * a02 + b01 * a12 + b02 * a22,
  b10 * a00 + b11 * a10 + b12 * a20,
  b10 * a01 + b11 * a11 + b12 * a21,
  b10 * a02 + b11 * a12 + b12 * a22,
  b20 * a00 + b21 * a10 + b22 * a20,
  b20 * a01 + b21 * a11 + b22 * a21,
  b20 * a02 + b21 * a12 + b22 * a22,
];
}
}
}

```

더 명확하게 하기 위해 평행 이동, 회전, 스케일을 위한 행렬을 만드는 함수를 만들어봅시다.

```

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

  rotation: function(angleInRadians) {
    var c = Math.cos(angleInRadians);
    var s = Math.sin(angleInRadians);
    return [
      c, -s, 0,
      s, c, 0,
      0, 0, 1,
    ];
  },

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

```

이제 셰이더를 바꿔봅시다. 기존 셰이더는 이렇게 되어 있었는데요.

```

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

uniform vec2 u_resolution;

```

```

uniform vec2 u_translation;
uniform vec2 u_rotation;
uniform vec2 u_scale;

void main() {
    // 위치에 스케일 적용
    vec2 scaledPosition = a_position * u_scale;

    // 위치에 회전 적용
    vec2 rotatedPosition = vec2(
        scaledPosition.x * u_rotation.y + scaledPosition.y * u_rotation.x,
        scaledPosition.y * u_rotation.y - scaledPosition.x * u_rotation.x);

    // 평행 이동 추가
    vec2 position = rotatedPosition + u_translation;
    ...
}

```

새로운 세이더는 훨씬 더 간단해질 겁니다.

```

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

uniform vec2 u_resolution;
uniform mat3 u_matrix;

void main() {
    // 위치에 행렬 곱하기
    vec2 position = (u_matrix * vec3(a_position, 1)).xy;
    ...
}

```

그리고 어떻게 사용하냐면,

```

// 장면 그리기
function drawScene() {

    ...

    // 행렬 계산
    var translationMatrix = m3.translation(translation[0], translation[1]);
    var rotationMatrix = m3.rotation(angleInRadians);
    var scaleMatrix = m3.scaling(scale[0], scale[1]);

    // 행렬 곱하기
    var matrix = m3.multiply(translationMatrix, rotationMatrix);
    matrix = m3.multiply(matrix, scaleMatrix);

    // 행렬 설정
    gl.uniformMatrix3fv(matrixLocation, false, matrix);

    // 사각형 그리기
    gl.drawArrays(gl.TRIANGLES, 0, 18);
}

```

여기 새로운 코드를 사용한 샘플입니다. 슬라이더는 동일하게 평행 이동, 회전, 스케일입니다. 하지만 세이더에서 사용되는 방식은 훨씬 더 간단해졌습니다.

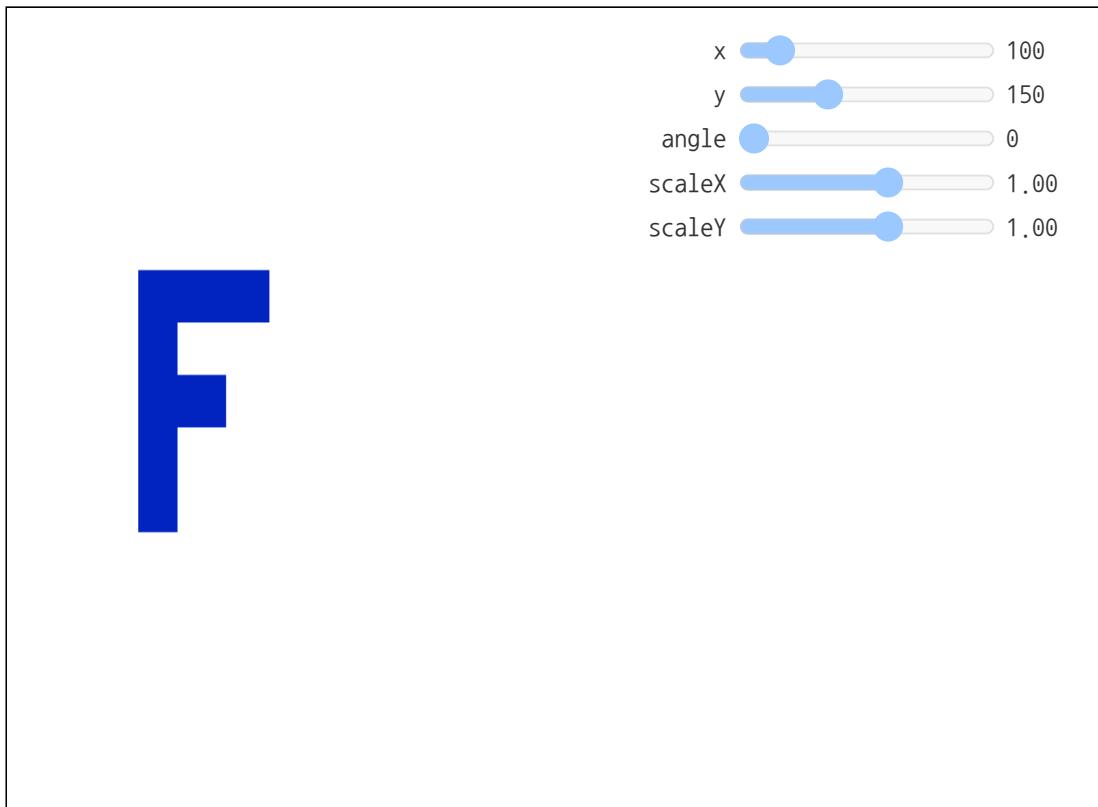


[새 창을 열려면 여기를 클릭](#)

그래도 물어보실 수 있습니다. 그래서 뭐요? 그다지 이점이 많아 보이지 않는데요. 하지만 이제 순서를 변경하려는 경우 새로운 세이더를 작성하지 않아도 됩니다. 그냥 수식만 바꿔주면 되죠.

```
...
// 행렬 곱하기
var matrix = m3.multiply(scaleMatrix, rotationMatrix);
matrix = m3.multiply(matrix, translationMatrix);
...
```

해당 버전입니다.



[새 창을 열려면 여기를 클릭](#)

이와 같은 행렬을 적용할 수 있다는 것은 신체의 팔, 태양 주변에 있는 행성의 위성, 나무의 가지같은 계층적 애니메이션에 특히 중요합니다. 계층적 애니메이션의 간단한 예제로 'F'를 5번 그리지만 매번 이전의 'F' 행렬로 시작해봅시다.

```
// 장면 그리기
function drawScene() {
    // 캔버스 지우기
    gl.clear(gl.COLOR_BUFFER_BIT);

    // 행렬 계산
    var translationMatrix = m3.translation(translation[0], translation[1]);
    var rotationMatrix = m3.rotation(angleInRadians);
    var scaleMatrix = m3.scaling(scale[0], scale[1]);

    // 행렬 시작
    var matrix = m3.identity();

    for (var i = 0; i < 5; ++i) {
        // 행렬 곱하기
        matrix = m3.multiply(matrix, translationMatrix);
        matrix = m3.multiply(matrix, rotationMatrix);
        matrix = m3.multiply(matrix, scaleMatrix);

        // 행렬 설정
        gl.uniformMatrix3fv(matrixLocation, false, matrix);

        // 지오메트리 그리기
        gl.drawArrays(gl.TRIANGLES, 0, 18);
    }
}
```

이를 위해 단위 행렬을 만드는 함수, `m3.identity` 를 도입했습니다. 단위 행렬은 사실상 1.0 을 나타내는 행렬이므로 단위 행렬을 곱해도 아무 일도 일어나지 않습니다.

$$X * 1 = X$$

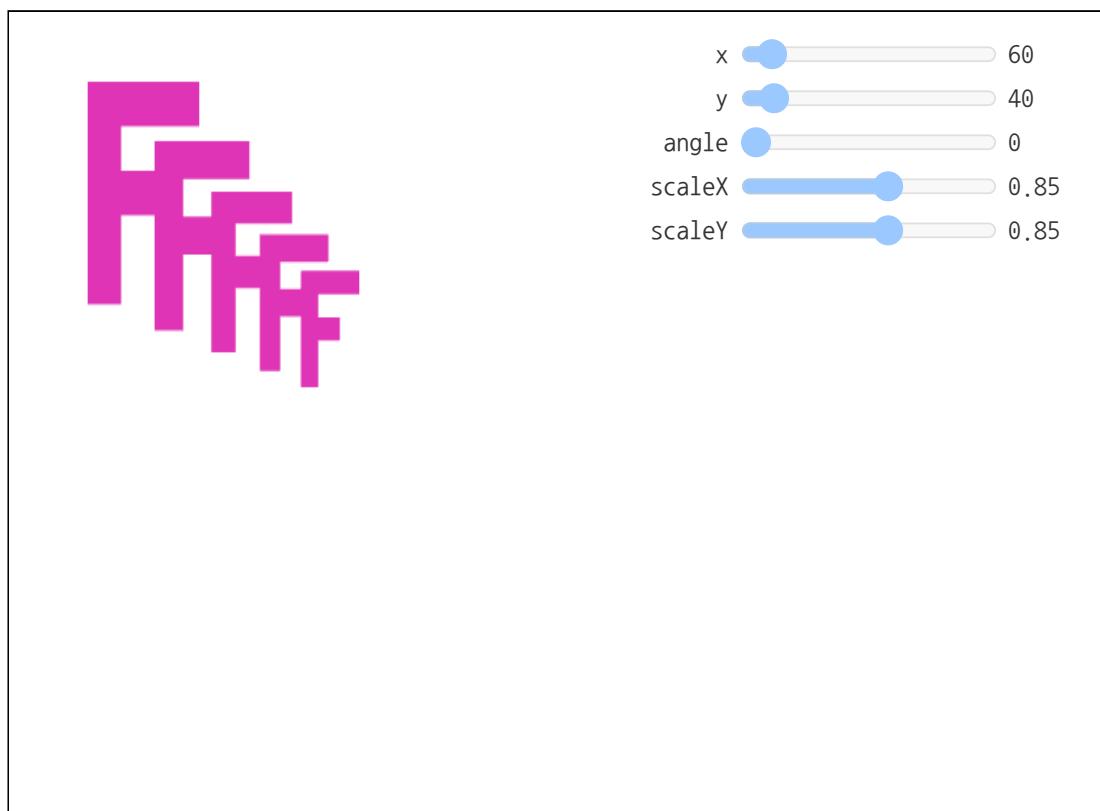
마찬가지로,

$$\text{matrix}X * \text{identity} = \text{matrix}X$$

다음은 단위 행렬을 만드는 코드입니다.

```
var m3 = {
  identity function() {
    return [
      1, 0, 0,
      0, 1, 0,
      0, 0, 1,
    ];
  },
  ...
}
```

다음은 5개의 F입니다.



[새 창을 열려면 여기를 클릭](#)

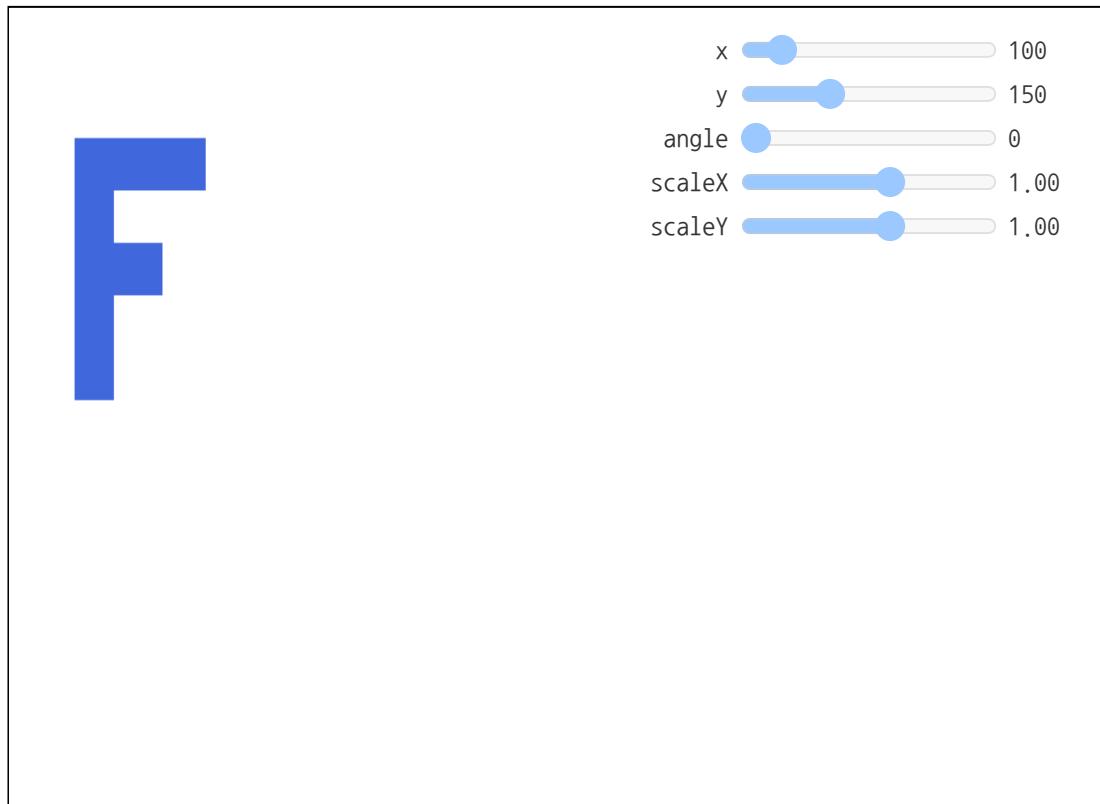
예제를 한 가지 더 봐봅시다. 지금까지 모든 샘플에서 'F'는 왼쪽 상단 모서리(예제를 제외하고는 위의 순서를 반대로 했음)를 기준으로 회전했는데요. 이건 우리가 사용하는 수식이 항상 원점을 기준으로 회전하고 'F'의 왼쪽 상단 모서리(0, 0)가 원점에 있기 때문입니다.

하지만 이제 행렬 수학을 할 수 있고 변환이 적용되는 순서를 선택할 수 있기 때문에 원점을 이동할 수 있습니다.

```
// 'F'의 원점을 중심으로 이동할 행렬 만들기
var moveOriginMatrix = m3.translation(-50, -75);
...

// 행렬 곱하기
var matrix = m3.multiply(translationMatrix, rotationMatrix);
matrix = m3.multiply(matrix, scaleMatrix);
matrix = m3.multiply(matrix, moveOriginMatrix);
```

여기 해당 샘플입니다. 참고로 F는 중심을 기준으로 회전하고 크기가 조정됩니다.



[새 창을 열려면 여기를 클릭](#)

이 기술을 사용하면 어떤 지점에서든 회전이나 크기를 조정할 수 있는데요. 이제 포토샵이나 플래시에서 어떻게 회전점을 이동시키는지 알게 되었습니다.

더 끝내주는 걸 해봅시다. 첫 번째 글인 [WebGL 기초](#)로 돌아가 보면, 세이더에 픽셀을 클립 공간으로 변환하는 코드가 있다는 걸 기억하실 겁니다.

```
...
// 사각형을 픽셀에서 0.0에서 1.0사이로 변환
vec2 zeroToOne = 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 * vec2(1, -1), 0, 1);
```

이 단계들을 차례대로 살펴보자면, 첫 단계, "픽셀을 0.0에서 1.0사이로 변환", 이건 실제 스케일링 작업입니다. 두 번째 역시 스케일링 작업입니다. 다음은 평행 이동이고 마지막으로 Y를 -1로 스케일링 하는데요. 실제로 우리가 세이더에 전달한 행렬로 모든 걸 수행할 수 있습니다. 2개의 스케일 행렬을 만들 수 있는데, 하나는 (1.0/해상도)로 스케일링하는 것이고, 다른 하나는 2.0으로 스케일링하는 것이며, 세 번째는 (-1.0, -1.0)으로 평행 이동하고, 네 번째는 Y를 -1로 스케일링한 뒤 모든 값을 함께 곱하지만, 대신 수식이 간단하기 때문에, 직접 주어진 해상도에 대한 '투영 행렬'을 생성하는 함수를 바로 만들어 봅시다.

```
var m3 = {
    projection: function(width, height) {
        // 참고: 이 행렬은 Y축을 뒤집어서 0이 위쪽에 있도록 합니다.
        return [
            2 / width, 0, 0,
            0, -2 / height, 0,
            -1, 1, 1
        ];
    },
    ...
}
```

이제 세이더를 더 단순하게 할 수 있습니다. 여기 완전히 새로운 정점 세이더입니다.

```
<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>
```

그리고 자바스크립트에서는 투영 행렬로 곱해야 합니다.

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

    // 행렬 계산
    var projectionMatrix = m3.projection(gl.canvas.clientWidth, gl.canvas.clientHeight);

    ...

    // 행렬 곱하기
    var matrix = m3.multiply(projectionMatrix, translationMatrix);
    matrix = m3.multiply(matrix, rotationMatrix);
    matrix = m3.multiply(matrix, scaleMatrix);
```

```
}
  ...
}
```

또한 해상도를 설정하는 코드를 삭제했는데요. 이 마지막 단계에서 우리는 행렬 수학의 마법 덕분에 6-7단계의 다소 복잡한 셰이더를 고작 1단계의 아주 간단한 셰이더로 바꿨습니다.



[새 창을 열려면 여기를 클릭](#)

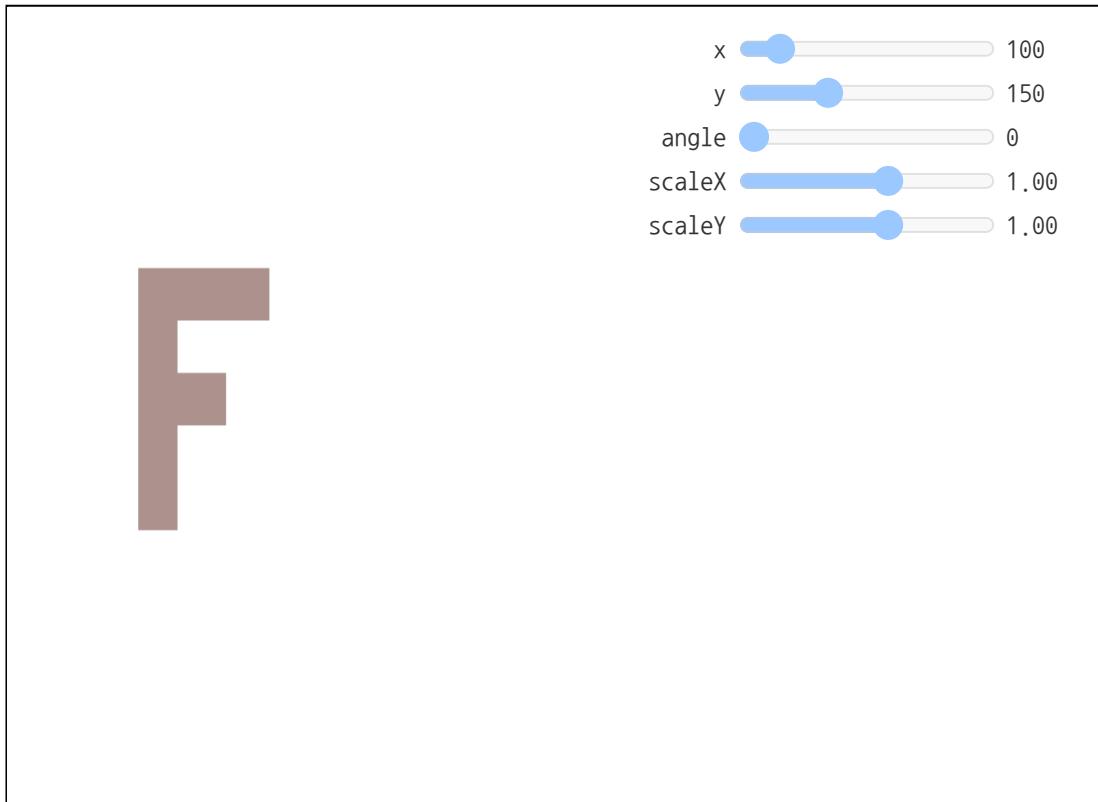
계속 진행하기 전에 조금 더 단순하게 해봅시다. 다양한 행렬을 생성하고 따로 곱하는 것이 일반적이지만 생성할 때마다 곱하는 것도 일반적인 방법입니다. 이처럼 효율적으로 함수를 정의 할 수 있고,

```
var m3 = {
  ...
  translate: function(m, tx, ty) {
    return m3.multiply(m, m3.translation(tx, ty));
  },
  rotate: function(m, angleInRadians) {
    return m3.multiply(m, m3.rotation(angleInRadians));
  },
  scale: function(m, sx, sy) {
    return m3.multiply(m, m3.scaling(sx, sy));
  },
  ...
};
```

위의 행렬 코드 7줄을 이렇게 4줄로 바꿀 수 있으며,

```
// 행렬 계산
var matrix = m3.projection(gl.canvas.clientWidth, gl.canvas.clientHeight);
matrix = m3.translate(matrix, translation[0], translation[1]);
matrix = m3.rotate(matrix, angleInRadians);
matrix = m3.scale(matrix, scale[0], scale[1]);
```

그리고 해당 결과입니다.



[새 창을 열려면 여기를 클릭](#)

마지막으로 한 가지, 위에서 순서에 따른 문제를 보았습니다. 첫 번째 예제에서는,

```
translation * rotation * scale
```

그리고 두 번째에서는,

```
scale * rotation * translation
```

이 둘이 어떻게 다른지 봤습니다.

행렬을 보는 두 가지 방법이 있는데요. 주어진 표현식이 다음과 같을 때,

```
projectionMat * translationMat * rotationMat * scaleMat * position
```

많은 사람들이 자연스럽게 찾는 첫 방법은 오른쪽에서 시작하여 왼쪽으로 계산하는 겁니다.

먼저 scaledPosition을 얻기 위해 위치에 스케일 행렬을 곱하고,

```
scaledPosition = scaleMat * position
```

그런 다음 rotatedScaledPosition을 얻기 위해 scaledPosition에 회전 행렬을 곱하며,

```
rotatedScaledPosition = rotationMat * scaledPosition
```

다음으로 translatedRotatedScaledPosition를 얻기 위해 rotatedScaledPosition에 평행 이동 행렬을 곱한 뒤,

```
translatedRotatedScaledPosition = translationMat * rotatedScaledPosition
```

마지막으로 클립 공간의 위치를 얻기 위해 투영 행렬에 곱하는데,

```
clipspacePosition = projectionMatrix * translatedRotatedScaledPosition
```

두 번째 방법은 행렬을 왼쪽에서 오른쪽으로 읽는 겁니다. 이 경우 각각의 행렬은 캔버스에 표시되는 공간을 변경합니다. 캔버스는 각 방향에서 클립 공간(-1 ~ +1)을 표시하는 것으로 시작하는데요. 왼쪽에서 오른쪽으로 적용된 각 행렬은 캔버스에 표시되는 공간을 변경합니다.

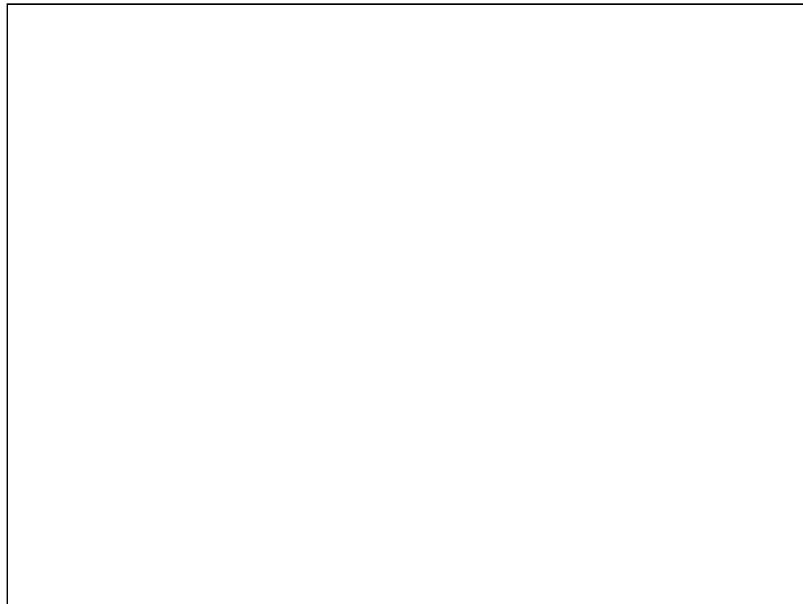
1단계: 행렬 없음 (혹은 단위 행렬)



클립 공간

흰색 영역은 캔버스입니다. 파란색은 캔버스 바깥입니다. 우리는 클립 공간에 있습니다. 전달된 위치는 클립 공간에 있어야 합니다.

2단계: `matrix = m3.projection(gl.canvas.clientWidth,
gl.canvas.clientHeight);`



클립 공간에서 픽셀 공간으로

이제 픽셀 공간에 있습니다. $X = 0$ 에서 400 , $Y = 0$ 에서 300 , 왼쪽 상단은 $0,0$ 입니다. 이 행렬을 사용하여 전달된 위치는 픽셀 공간에 있어야 합니다. 공간이 $+Y =$ 상단에서 $-Y =$ 하단으로 뒤집힐 때의 순간을 보실 수 있습니다.

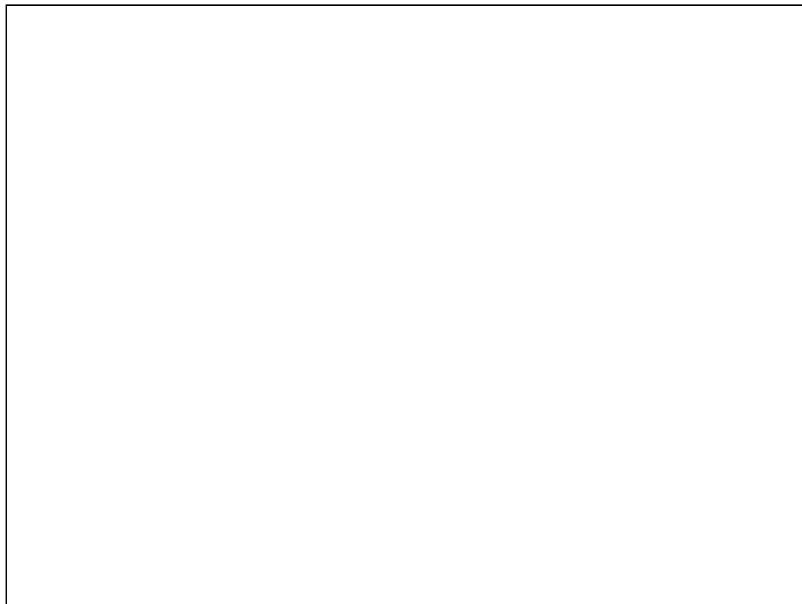
3단계: `matrix = m3.translate(matrix, tx, ty);`



원점을 tx, ty로 이동

원점은 이제 tx, ty (150, 100)로 이동됐습니다. 공간도 이동했습니다.

4단계: `matrix = m3.rotate(matrix, rotationInRadians);`



33도 회전

공간이 tx, ty를 중심으로 회전됐습니다.

5단계: `matrix = m3.scale(matrix, sx, sy);`



공간 크기 조정

이전에 tx, ty를 중심으로 회전된 공간이, x는 2로 y는 1.5로 스케일링 되었습니다.

세이더에서 `gl_Position = matrix * position;`을 실행하는데요. position 값은 마지막 공간에서 유효합니다.

이해하기 더 쉽다고 느껴지는 방법을 사용하시면 됩니다.

이 포스트가 행렬 수학을 이해하는데 도움이 되었기를 바랍니다. 2D를 계속 하고 싶다면 [Canvas 2D의 drawImage 함수 재현](#)과 [Canvas 2D의 행렬 스택 재현](#)을 봐주세요.

그게 아니라면 다음은 [3D](#)로 넘어갑니다. 3D에서 행렬 수학은 동일한 원리과 사용법을 따르는데요. 2D부터 시작해서 이해하기 쉽도록 만들었습니다.

또한 정말로 행렬 수학의 전문가가 되고 싶다면 [이 놀라운 영상](#)을 확인하세요.

clientWidth 와 clientHeight 가 뭔가요?

지금까지는 캔버스의 넓이를 참조할 때마다 `canvas.width`과 `canvas.height`를 사용했지만 위에서 `m3.projection`를 호출할 때는 `canvas.clientWidth`와 `canvas.clientHeight`를 사용했습니다. 왜 그랬을까요?

투영 행렬은 클립 공간(각 치수마다 -1 ~ +1)을 가져와서 다시 픽셀로 변환하는 방법과 관련이 있습니다. 하지만 브라우저에는 두 가지 타입의 픽셀이 있는데요. 하나는 캔버

스 자체의 픽셀 수입니다. 예를 들어 이렇게 정의된 캔버스가 있습니다.

```
<canvas width="400" height="300"></canvas>
```

혹은 이렇게 정의됩니다.

```
var canvas = document.createElement("canvas");
canvas.width = 400;
canvas.height = 300;
```

둘 다 너비가 400픽셀이고 높이가 300픽셀인 이미지를 포함합니다. 하지만 이 크기는 실제로 브라우저가 캔버스를 표시하는 크기와는 별도인데요. CSS는 캔버스가 표시되는 크기를 정의합니다. 예를 들어 이렇게 캔버스를 만든다고 해봅시다.

```
<style>
  canvas {
    width: 100vw;
    height: 100vh;
  }
</style>
...
<canvas width="400" height="300"></canvas>
```

컨테이너의 크기에 상관없이 캔버스가 표시될 겁니다. 400x300은 아닌 것 같군요.

여기 캔버스의 CSS 표시 크기를 100%로 설정해서 캔버스가 페이지를 채우도록 늘어나는 두 예제가 있습니다. 첫 번째는 `canvas.width`와 `canvas.height`를 사용하는 겁니다. 새로운 창을 열고 창 크기를 조절해보세요. 'F'가 올바른 모양을 가지는지 확인해봅시다. 왜곡되고 있는데요.



[새 창을 열려면 여기를 클릭](#)

두 번째 예제에서는 `canvas.clientWidth`와 `canvas.clientHeight`를 사용합니다. `canvas.clientWidth`와 `canvas.clientHeight`는 캔버스가 실제 브라우저에서 표시되는 크기를 알려주기 때문에, 이 경우 캔버스는 여전히 400x300이지만 캔버스가 표시되는 크기에 따라 종횡비를 정의하고 있으므로 F는 항상 올바르게 보입니다.



[새 창을 열려면 여기를 클릭](#)

캔버스의 크기를 조절할 수 있는 대부분의 앱은 `canvas.width` 와 `canvas.height` 를 `canvas.clientWidth` 와 `canvas.clientHeight` 에 맞추려고 하는데, 브라우저에 표시되는 각 픽셀마다 캔버스에 하나의 픽셀이 있기를 원하기 때문입니다. 위에서 보셨듯이 그게 유일한 선택지는 아닙니다. 하지만 거의 모든 경우에 `canvas.clientHeight` 와 `canvas.clientWidth` 를 사용해서 투영 행렬의 종횡비를 계산하는 것이 기술적으로 더 정확합니다.