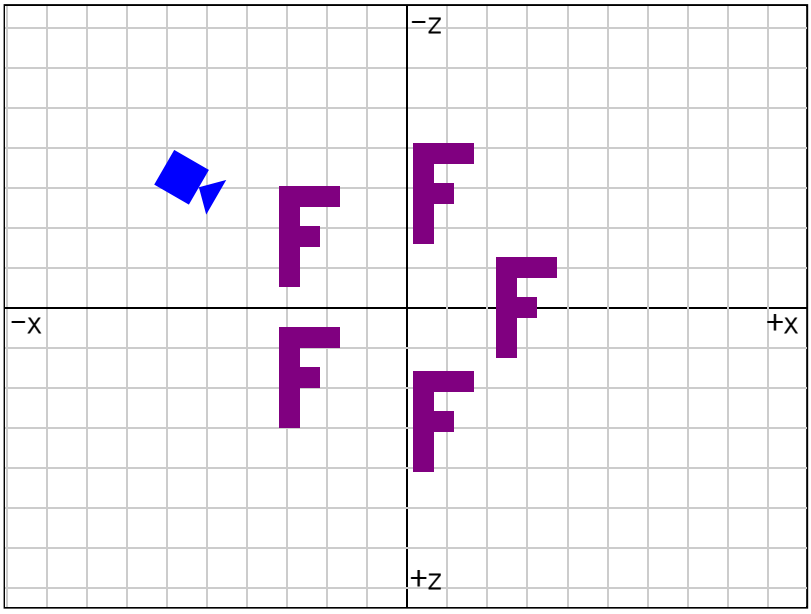
WebGL 3D - 카메라

이 포스트는 WebGL 관련 시리즈에서 이어집니다. 첫 번째는 기초로 시작했고, 이전에는 3D 원근 투영에 관한 것이었습니다. 아직 읽지 않으셨다면 해당 글들을 먼저 읽어주세요.

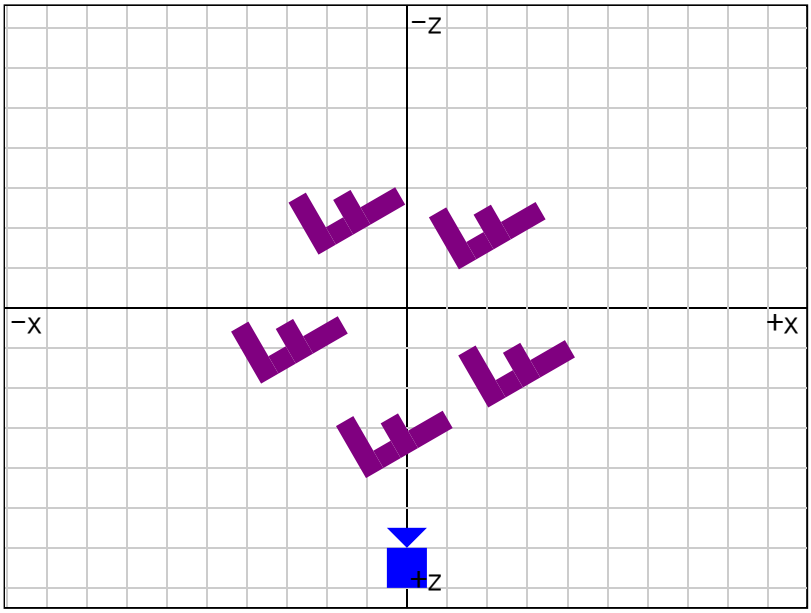
`m4.perspective` 함수는 F가 원점(0, 0, 0)에 있으리라 생각하고, 절두체의 객체는 원점 앞 `-zNear` 에서 `-zFar` 까지 있기 때문에, 지난 포스트에서 우리는 F를 절두체 앞으로 옮겨야 했습니다.

뷰 앞의 물체를 움직이는 것은 그다지 좋은 방법이 아닌 것 같습니다. 현실에서는 보통 건물의 사진을 찍기 위해 카메라를 움직이죠.



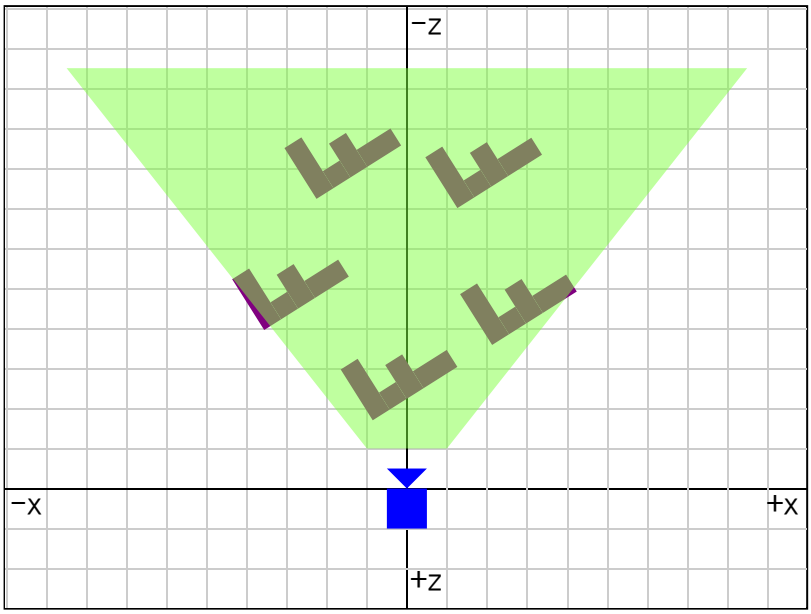
카메라를 객체로 이동

보통은 카메라 앞으로 건물을 움직이지 않습니다.



객체를 카메라로 이동

하지만 지난 포스트에서 우리는 -Z축의 원점 앞에 있도록 하는 투영법을 찾아냈습니다. 이를 위해 원점으로 카메라를 옮기고 나머지를 적절하게 이동하여 *카메라를 기준으로 동일한 위치*에 있도록 했습니다.



객체를 뷰로 이동

사실상 카메라 앞에 있는 월드를 움직여야 하는데요. 가장 쉬운 방법은 "역행렬"을 사용하는 겁니다. 일반적인 경우에 역행렬을 계산하는 수식은 복잡하지만 개념적으로는 쉽습니다. "역"은 어떤 값의 정반대로 사용하는 값입니다. 예를 들어 X에서 123으로 평행 이동하는 행렬의 역은 X에서 -123으로 평행 이동하는 행렬입니다. 5로 스케일링하는 행렬의 역은 1/5이나 0.2로 스케일링하는 행렬입니다. X축을 중심으로 30° 회전하는 행렬의 역은 X축을 중심으로 -30° 회전하는 행렬이 됩니다.

지금까지 'F'의 위치와 오리엔테이션에 영향을 주기 위한 평행 이동, 회전, 스케일을 사용했습니다. 모든 행렬을 곱한 후 'F'를 원점에서 원하는 위치, 크기, 오리엔테이션으로 움직이는 방법을 나타내는 단일 행렬이 생기는데요. 카메라에서도 똑같이 할 수 있습니다. 카메라를 원점

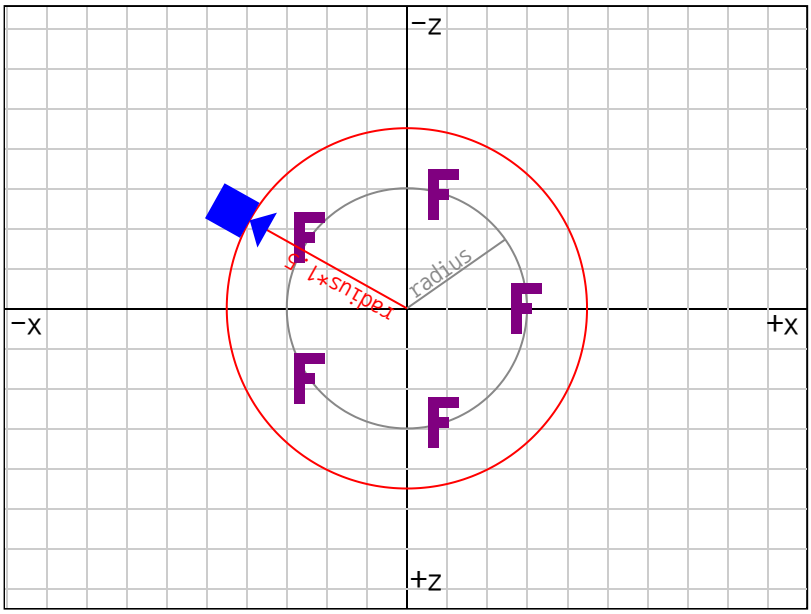
에서 원하는 위치로 이동하고 회전하는 방법을 알려주는 행렬이 있으면, 나머지를 정반대의 결과로 이동하고 회전하는 방법을 알려주는 행렬인 역행렬을 계산할 수 있으므로, 카메라는 (0, 0, 0)에 있도록 하고 그 앞에 있는 모든 것을 움직일 겁니다.

위 다이어그램처럼 'F'가 선회하는 3D 장면을 만들어봅시다.

우선 5개를 그리고 있고 모두 동일한 투영 행렬을 사용하기 때문에 루프 바깥에서 계산할 겁니다.

```
// 투영 행렬 계산
var aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
var zNear = 1;
var zFar = 2000;
var projectionMatrix = m4.perspective(fieldOfViewRadians, aspect, zNear, zFar);
```

다음으로 카메라 행렬을 계산할 겁니다. 행렬은 월드에서 카메라의 위치와 오리엔테이션을 나타냅니다. 아래 코드는 카메라가 원점을 중심으로 radius * 1.5의 거리에서 원점을 바라보며 회전하는 행렬을 만듭니다.



카메라 움직임

```
var numFs = 5;
var radius = 200;

// 카메라에 대한 행렬 계산
var cameraMatrix = m4.yRotation(cameraAngleRadians);
cameraMatrix = m4.translate(cameraMatrix, 0, 0, radius * 1.5);
```

그런 다음 카메라 행렬에서 "뷰 행렬"를 계산합니다. "뷰 행렬"는 마치 카메라가 원점(0,0,0)에 있는 것처럼, 모든 걸 카메라와 정반대로 움직여 사실상 카메라를 기준으로 만드는 행렬입니다. 역행렬(제공된 행렬의 정반대를 수행하는 행렬)을 계산하는 `inverse` 함수를 사용하여 이를 수행할 수 있습니다. 이 경우 제공된 행렬은 원점을 기준으로 카메라를 어떤 위치와 오리

엔테이션으로 옮깁니다. 그 반대는 나머지를 움직여서 카메라가 원점에 있도록 하는 행렬입니다.

```
// 카메라 행렬로 뷰 행렬 만들기
var viewMatrix = m4.inverse(cameraMatrix);
```

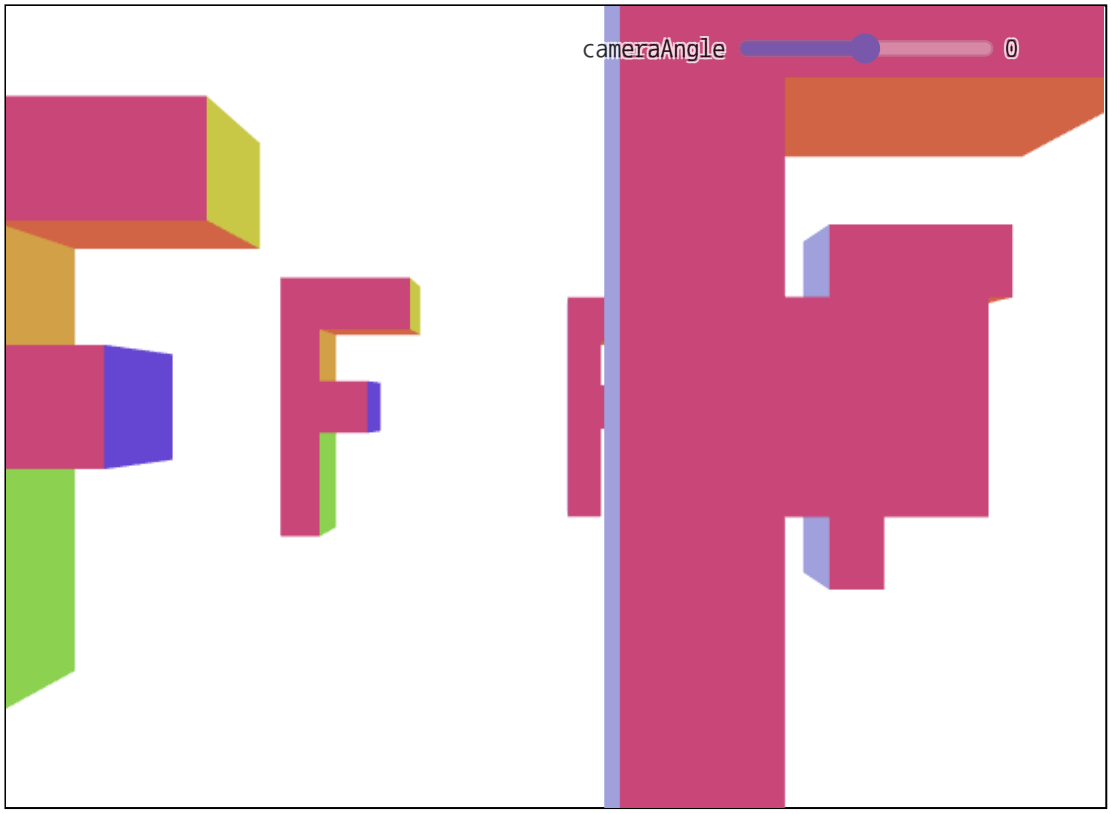
이제 뷰 행렬와 투영 행렬은 뷰 투영 행렬로 합칩니다.

```
// 뷰 투영 행렬 계산
var viewProjectionMatrix = m4.multiply(projectionMatrix, viewMatrix);
```

마지막으로 F의 원을 그립니다. 각 F에 대해 뷰 투영 행렬로 시작한 다음, 반경 단위로 회전하고 이동합니다.

```
for (var ii = 0; ii < numFs; ++ii) {
  var angle = (ii * Math.PI * 2) / numFs;
  var x = Math.cos(angle) * radius;
  var y = Math.sin(angle) * radius;

  // 뷰 투영 행렬로 시작하여 F에 대한 행렬 계산
  var matrix = m4.translate(viewProjectionMatrix, x, 0, y);

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

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

그리고 짜잔! 카메라가 F의 원을 돌고 있습니다. `cameraAngle` 을 드래그하여 카메라를 움직여보세요.



[새 창을 열려면 여기를 클릭]

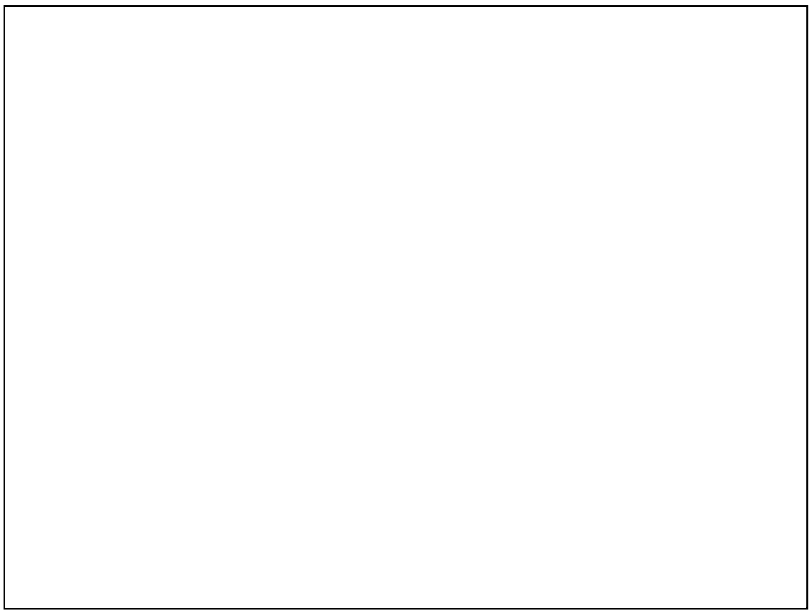
전부 괜찮아 보이지만 회전과 평행 이동을 사용하여 카메라를 원하는 곳으로 움직여 보고 싶은 방향을 향하게 하기가 쉽지 않은데요. 예를 들어 카메라가 특정 F를 가리키도록 하려면, 'F'의 원이 회전하는 동안 카메라가 해당 'F'를 가리키도록 회전하는 방법을 계산하기 위해 꽤 어려운 수식이 필요합니다.

다행히 더 쉬운 방법이 있는데요. 원하는 카메라 위치와 가리킬 대상을 결정한 다음, 그곳에 카메라를 배치하는 행렬을 계산할 수 있습니다. 행렬이 작동하는 방식에 기반한 이 작업은 놀라울 정도로 쉽습니다.

먼저 원하는 카메라 위치를 알아야 합니다. 이를 `cameraPosition` 이라 부를겁니다. 그런 다음 보고 싶거나 목표로 하고 싶은 것의 위치를 알아야 합니다. 이건 `target` 이라 부를겁니다. `cameraPosition` 에서 `target` 을 빼면 대상에 도달하기 위해 카메라가 이동해야 하는 방향을 가리키는 벡터가 생깁니다. 이걸 `zAxis` 라 부릅시다. 카메라가 -Z 방향을 향한다는 것을 알고 있기 때문에 다른 방법 `cameraPosition - target` 으로 뺄 수 있습니다. 그리고 결과를 정규화한 다음 행렬의 `z` 부분으로 직접 복사합니다.

```
+----+----+----+----+
|    |    |    |    |
+----+----+----+----+
|    |    |    |    |
+----+----+----+----+
| Zx | Zy | Zz |    |
+----+----+----+----+
|    |    |    |    |
+----+----+----+----+
```

행렬의 이 부분은 Z축을 나타냅니다. 이 경우에는 카메라의 Z축입니다. 벡터 정규화는 벡터를 1.0을 나타내는 벡터로 만드는 걸 의미합니다. 2D 회전으로 돌아가보면 단위원이 2D 회전에 어떻게 도움이 되는지 말했었는데요. 3D에서는 단위구가 필요하고 정규화된 벡터는 단위구의 한 점을 나타냅니다.



Z축

그래도 정보가 충분하지 않습니다. 단일 벡터만으로도 단위구의 한 점을 제공하지만, 해당 점에서 사물이 향하는 오리엔테이션은 뭘까요? 이를 알기 위해서는 행렬의 다른 부분을 채워야 합니다. 특히 X축과 Y축 부분이요. 우리는 일반적으로 이 세 부분이 서로 수직임을 알고있습니다. 또한 "일반적으로" 카메라를 똑바로 향하지 않는다는 것도 알죠. 이를 고려해서 이 경우 (0,1,0)인 위쪽이 어느 쪽인지 안다면, 그것과 "벡터곱"을 사용하여 행렬에 대한 X축과 Y축을 계산할 수 있습니다.

수학적 용어로 벡터곱이 무엇을 의미하는지는 모르겠습니다. 제가 아는 건 2개의 단위 벡터가 있고 이들을 벡터곱으로 계산하면, 이 두 벡터에 수직인 벡터를 얻는다는 겁니다. 다시 말해 남동쪽을 가리키는 벡터와 위쪽을 가리키는 벡터를 벡터곱하면, 남동쪽과 위쪽에 수직인 두 벡터이기 때문에, 남서쪽이나 북동쪽을 가리키는 벡터를 얻게 됩니다. 계산하는 순서에 따라 반대의 결과를 얻게 될 겁니다.

하여튼 `zAxis` 와 `up` 의 벡터곱을 계산하면 카메라의 xAxis를 얻을 수 있습니다.

up x zAxis = xAxis

그리고 이제 `xAxis` 가 있으니 `zAxis` 와 `xAxis` 를 벡터곱해서 카메라의 `yAxis` 를 얻을 수 있습니다.

zAxis x xAxis = yAxis

이제 남은 일은 3개의 축을 행렬로 연결하는 겁니다. 이는 `cameraPosition` 에서 `target` 을 가리키는 무언가를 향하는 행렬을 제공하는데요. `position` 만 추가하면 됩니다.

```
+----+----+----+----+
| Xx | Xy | Xz |  0 |  <- x축
+----+----+----+----+
| Yx | Yy | Yz |  0 |  <- y축
+----+----+----+----+
| Zx | Zy | Zz |  0 |  <- z축
+----+----+----+----+
| Tx | Ty | Tz |  1 |  <- 카메라 위치
+----+----+----+----+
```

두 벡터의 벡터곱을 계산하는 코드

```
function cross(a, b) {
  return [
    a[1] * b[2] - a[2] * b[1],
    a[2] * b[0] - a[0] * b[2],
    a[0] * b[1] - a[1] * b[0],
  ];
}
```

두 벡터를 뺄셈하는 코드

```
function subtractVectors(a, b) {
  return [a[0] - b[0], a[1] - b[1], a[2] - b[2]];
}
```

벡터를 정규화하는 코드 (단위 벡터로 만듦)

```
function normalize(v) {
  var length = Math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
  // 0으로 나뉘지 않도록 하기
  if (length > 0.00001) {
    return [v[0] / length, v[1] / length, v[2] / length];
  } else {
    return [0, 0, 0];
  }
}
```

"lookAt" 행렬을 계산하는 코드

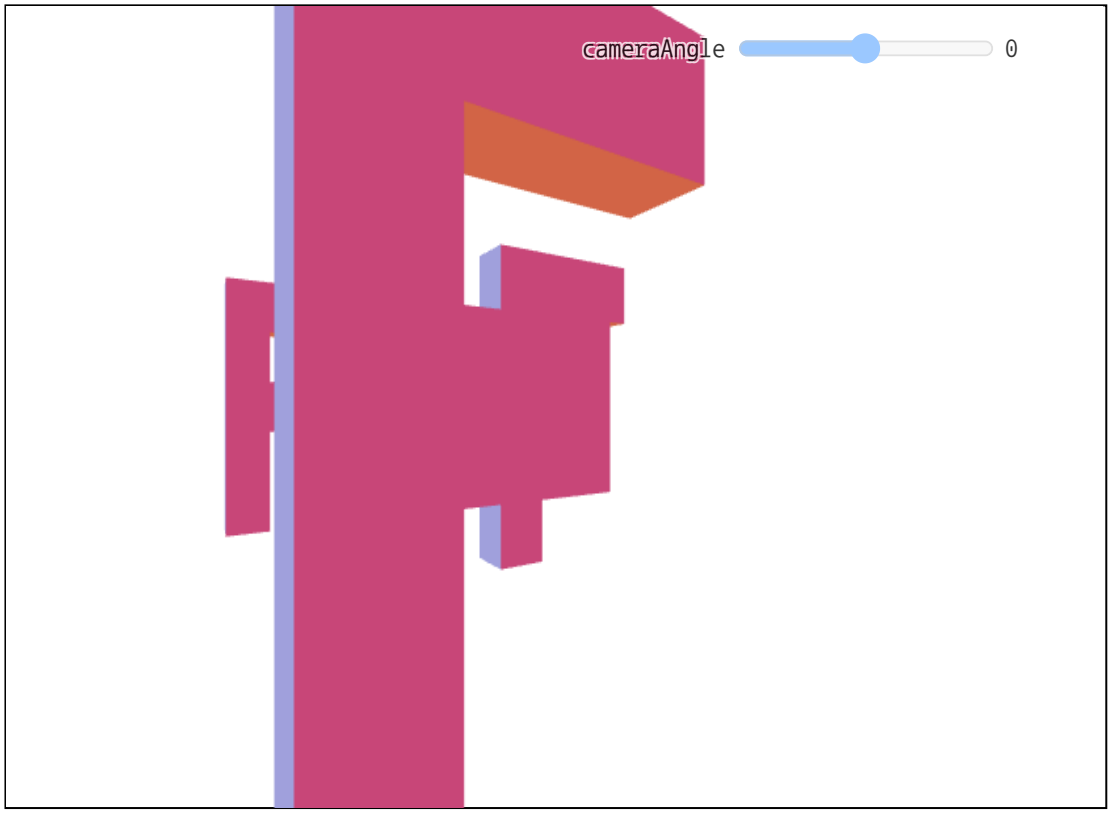
```
var m4 = {
  lookAt: function(cameraPosition, target, up) {
    var zAxis = normalize(
        subtractVectors(cameraPosition, target));
    var xAxis = normalize(cross(up, zAxis));
```

```
  var yAxis = normalize(cross(zAxis, xAxis));

  return [
     xAxis[0], xAxis[1], xAxis[2], 0,
     yAxis[0], yAxis[1], yAxis[2], 0,
     zAxis[0], zAxis[1], zAxis[2], 0,
     cameraPosition[0],
     cameraPosition[1],
     cameraPosition[2],
     1,
  ];
}
```

다음은 카메라를 움직일 때 특정 'F'를 가리키도록 만드는 방법입니다.

```
...

// 첫 번째 F의 위치 계산
var fPosition = [radius, 0, 0];

// 원에서 카메라가 있는 위치를 계산하는 행렬 수학 사용
var cameraMatrix = m4.yRotation(cameraAngleRadians);
cameraMatrix = m4.translate(cameraMatrix, 0, 0, radius * 1.5);

// 계산한 행렬에서 카메라의 위치 가져오기
var cameraPosition = [
  cameraMatrix[12],
  cameraMatrix[13],
  cameraMatrix[14],
];

var up = [0, 1, 0];

// lookAt을 사용하여 카메라 행렬 계산
var cameraMatrix = m4.lookAt(cameraPosition, fPosition, up);

// 카메라 행렬로 뷰 행렬 만들기
var viewMatrix = m4.inverse(cameraMatrix);

...
```

그리고 여기 결과입니다.



새 창을 열려면 여기를 클릭

슬라이더를 드래그해서 카메라가 어떻게 'F'를 따라가는지 확인해보세요.

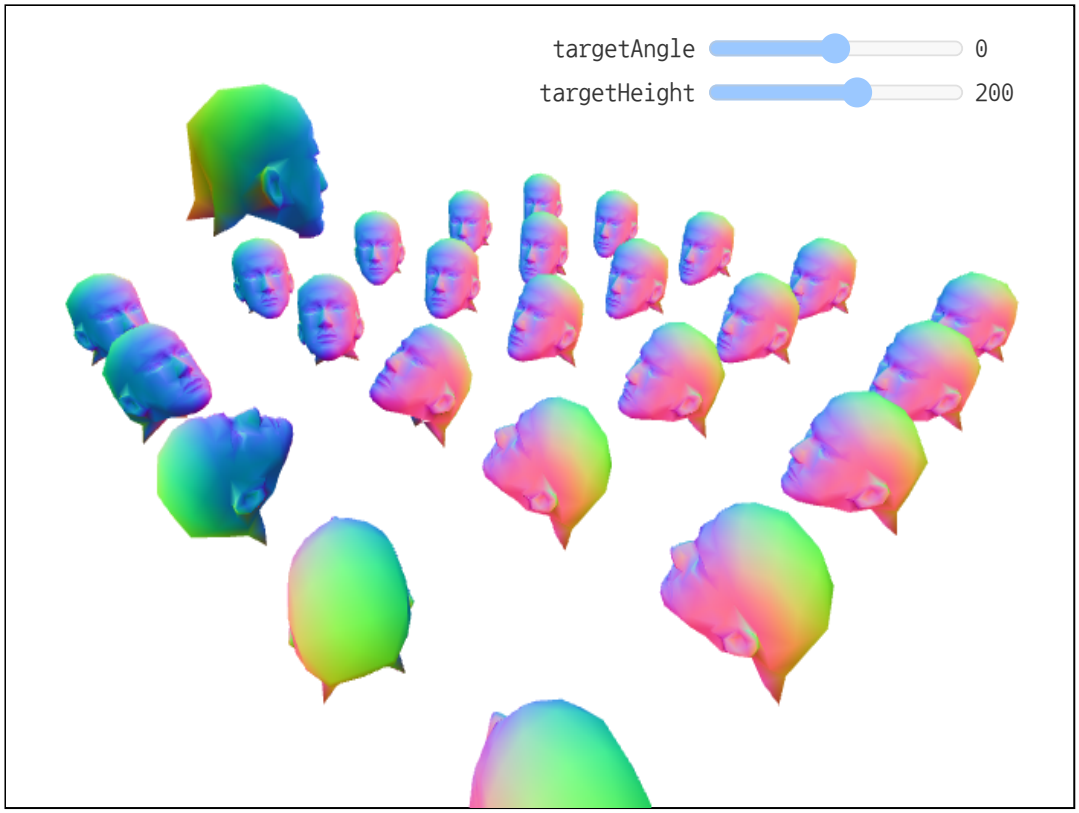
참고로 카메라 이외에도 "lookAt" 수식을 사용할 수 있습니다. 일반적인 용도는 누군가를 따라가는 캐릭터의 머리를 만드는 겁니다. 목표를 겨냥하는 포탑을 만들 수 있고, 개체가 경로를 따라가게 만들 수도 있는데요. 먼저 경로상에 대상이 있는 위치를 계산합니다. 그런 다음 잠시 후에 경로상에 대상이 어디에 있는지 계산합니다. 두 값을 `lookAt` 함수로 연결하면 개체가 경로를 따라가고 경로를 향하도록 하는 행렬을 얻을 수 있습니다.

다음은 애니메이션에 대해 배워보겠습니다.

lookAt 표준

대부분의 3D 수학 라이브러리는 `lookAt` 함수를 가지고 있습니다. 종종 "카메라 행렬"이 아닌 "뷰 행렬"을 만들기 위해 설계된 함수들이 있는데요. 다시 말해 카메라 자체를 움직이는 행렬이 아니라 카메라 앞에 있는 모든 걸 움직이는 행렬을 만듭니다.

저는 이게 덜 유용하다고 생각합니다. lookAt 함수는 많은 용도로 사용되는데요. 뷰 행렬이 필요할 때 `inverse`를 호출하기 쉽긴 하지만, 어떤 캐릭터의 머리가 다른 캐릭터를 따라가게 하거나 어떤 포탑이 목표를 겨냥하게 만들기 위해 `lookAt`을 사용한다면, `lookAt`이 월드 공간에서 개체의 방향과 위치를 지정하는 행렬을 반환하는 경우에 훨씬 더 유용합니다.



새 창을 열려면 여기를 클릭