

Three.js 시작하기

02 Apr 2021

사실은 three.js 를 하기 전에 WebGL 로 정육면체까지 만들고 싶었는데 아래 글을 읽다가 포기했다..

[webglfundamentals](#)

사실 또 하려면 못할것까진 없지만 애초에 WebGL 은 three.js 를 하기 전에 기초만 떼기 위해 한거라서 시간을 너무 많이 투자하긴 아깝고, 이미 three.js 로 정육면체를 그리는게 얼마나 편한지 알아서 바로 이걸 하는게 낫다고 생각했다. 혹시 나중에 배울 일이 생기면 그때 봐야겠다.

Three.js 란?

WebGL 을 사용해봤으면 알겠지만, WebGL 로 뭔가를 렌더링 하려면 엄청난 양의 코드가 필요하다. 특히 3D 라면 더더욱 어렵고 복잡하다. 그래서 WebGL 로 좀 더 쉽게 3D 렌더링을 할 수 있도록 Three.js, Scene.js 같은 라이브러리들이 등장했다.

Three.js 에선 Scene 위에 객체들을 올리고, Camera 를 통해 화면을 어떻게 렌더링할지 정한 후, Renderer 가 Scene 과 Camera 정보를 이용해 3D 씬을 렌더링하는 방식으로 작동한다.

Three.js 의 구조를 잘 설명한 사진이 있어서 가져와봤다.

(출처:

<https://threejsfundamentals.org/threejs/lessons/kr/threejs-fundamentals.html>)

Three.js 시작

three.js 를 시작하려면 일단 프로젝트에 three.js 파일을 추가해야 한다. .js 파일을 만들어 두고, 아래 링크의 코드를 복사한 다음 붙여넣으면 된다.

<https://threejs.org/build/three.js>

그 후 아래와 같이 body 태그에 포함시킨다.

(DemoThreeJS 폴더에 제공된다.)

```
<body>  
  <script src="three.js"></script>  
</body>
```

이제 Three.js 를 사용해보면 된다.

나는 Three.js 를 처음 사용할 때, 정육면체를 먼저 그려봤다.

1. 먼저, canvas 를 html 요소에 포함한다. canvas 를 만들지 않아도 자동으로 canvas 를 생성해주지만, 어차피 나중에 만들어야 하니 미리 만들어 놓자.

2. <body>

```

3.    <canvas id="c"></canvas>
4.    <script src="js/three.js"></script>
5.    <script>
6.        //이 부분에 코드 작성 시작
7.    </script>
8. </body>

```

코드를 짜는 방법은 html 파일에 script 태그로 넣든 따로 js 파일을 만들든 상관없다.

9. 만들어놓은 canvas 를 불러오고, Three.js 의 핵심이 되는 renderer 를 canvas 위에 생성한다.

```

10.    const canvas =
    document.querySelector("#c");
11.    const renderer = new
    THREE.WebGLRenderer({ canvas });

```

여기서 THREE 가 전부 대문자인걸 주의하자.
나는 Three 라고 써서 엄청 시간을 잡아먹었다..

12. 다음으로 Camera 를 만들어보자.

```

13.    const camera = new
    THREE.PerspectiveCamera(
14.        75,
15.        window.innerWidth /
    window.innerHeight,
16.        0.1,
17.        5
18.    );
19.    camera.position.z = 2; //카메라
    위치 이동, default 로 아래쪽 바라봄.

```

Threejs 에는 몇가지 카메라가 있는데, 여기서는 PerspectiveCamera 를 사용했다.

PerspectiveCamera 의 인자는 차례대로 field of view(시야각), aspect, near, far 이다.

- field of view(시야각): 말 그대로 볼 수 있는 각도.
- aspect: canvas 의 크기 비율, canvas 크기를 따로 설정하지 않으면 2 가 기본값이다.
- near, far: 렌더링 범위를 near~far 로 제한

fov



20. 다음으로는 Scene 을 만들자.

```
21.      const scene = new THREE.Scene();
```

이제 필수적으로 필요한 것들은 다 만들었다.
다음으로 정육면체를 만들어서 Scene 에
추가하면 된다.

간단하게 기하 객체를 만드는 법을 소개하자면,
Geometry 객체와 Material 객체를 생성한 후, 이
둘을 포함한 Mesh 객체를 만들어주면 된다.

22. 그럼 먼저 Geometry(형태)를 만들어보자.

```
23.      const geometry = new  
      THREE.BoxGeometry();
```

이 BoxGeometry 가 정육면체의
Geometry 이다.인자로 순서대로 boxWidth,
boxHeight, boxDepth 를 정해줄 수 있지만,
정해주지 않으면 기본적으로 1 로 된다.

24. 다음으로 Material 을 만들자.

```
25.      const material = new  
      THREE.MeshBasicMaterial({ color:  
      0x00ff00 });
```

color 외에 image 를 줄 수도 있고 밝기를 줄
수도 있는데 일단은 color 만 줬다.

26. 이제 Mesh 를 만들고 Scene 에 추가하자.

```
27.      const mesh = new  
      THREE.Mesh(geometry, material);  
28.      scene.add(mesh);  
29.  
30.      renderer.render(scene, camera);
```

여기까지 하고 실행해보면 내가 만든게 이게 맞나..
싶은 끔찍한 사각형이 보일거다.



입체적으로 보이지 않는 이유는 카메라가 한면만
보고 있기 때문이다. 정사각형이 아닌게 보일 수도

있는데, 아까 camera 를 만들때, canvas 크기 비율이 아닌 화면 크기 비율을 줘서 그렇다.

이제 정육면체의 모든 면을 볼 수 있게, 그리고 canvas 크기를 화면에 맞게 만들어보자.

1. 먼저 canvas 크기를 맞춰보자. css 로 html, body 의 margin 을 없애고, height:100%를 준다. 그 후
2. `renderer.setSize(window.innerWidth, window.innerHeight);`

이 코드를 `const renderer` 을 선언한 부분 다음 줄에 집어넣자. 이제 새로고침하면 화면 크기에 딱 맞는 canvas 가 된다.

화면 size 가 변했을 때 canvas 크기가 변하지 않아 되게 불편한데, 이걸 마지막에 고쳐보자.

3. 이제 정육면체의 모든 면을 보기 위해 회전시켜보자.

```
4. function render(time) {  
5.   time *= 0.001;  
6.  
7.   mesh.rotation.x = time;  
8.   mesh.rotation.y = time;  
9.  
   renderer.render(scene, camera);  
   requestAnimationFrame(render);  
}  
requestAnimationFrame(render);
```

위 코드를 `renderer.render(scene, camera)`가 있던 부분을 지우고 추가한다. 여러 방법이 있는데,

time 을 쓰는 방법을 사용했다. mesh.rotation.x, y 의 값을 적당히 잘 바꿔주기만 하면 된다.

requestAnimationFrame 은 디스플레이 주사율에 맞춰 함수를 반복적으로 실행해주는데, setInterval 함수와 비슷하지만 화면을 떠났을 때 자동으로 멈춰주는 기능이 있어 훨씬 좋다.

이제 실행해보면 정육면체가 빙글 돌고있을 것이다.

여기서 끝내도 나쁘지 않지만, 광원을 추가하고, 화면 사이즈 조절까지 해보겠다.

10. 광원추가

```
11. {
12.     const color = 0xffffffff;
13.     const intensity = 1;
14.     const light = new
    THREE.DirectionalLight(color,
    intensity);
15.     light.position.set(-1, 2, 4);
    //광원 위치 설정
16.     scene.add(light); //Scene 에
    추가
17. }
```

이제 아까 const material = new THREE.MeshBasicMaterial({color: 0x00ff00}); 을 선언한 곳에서 MeshBasicMaterial 을

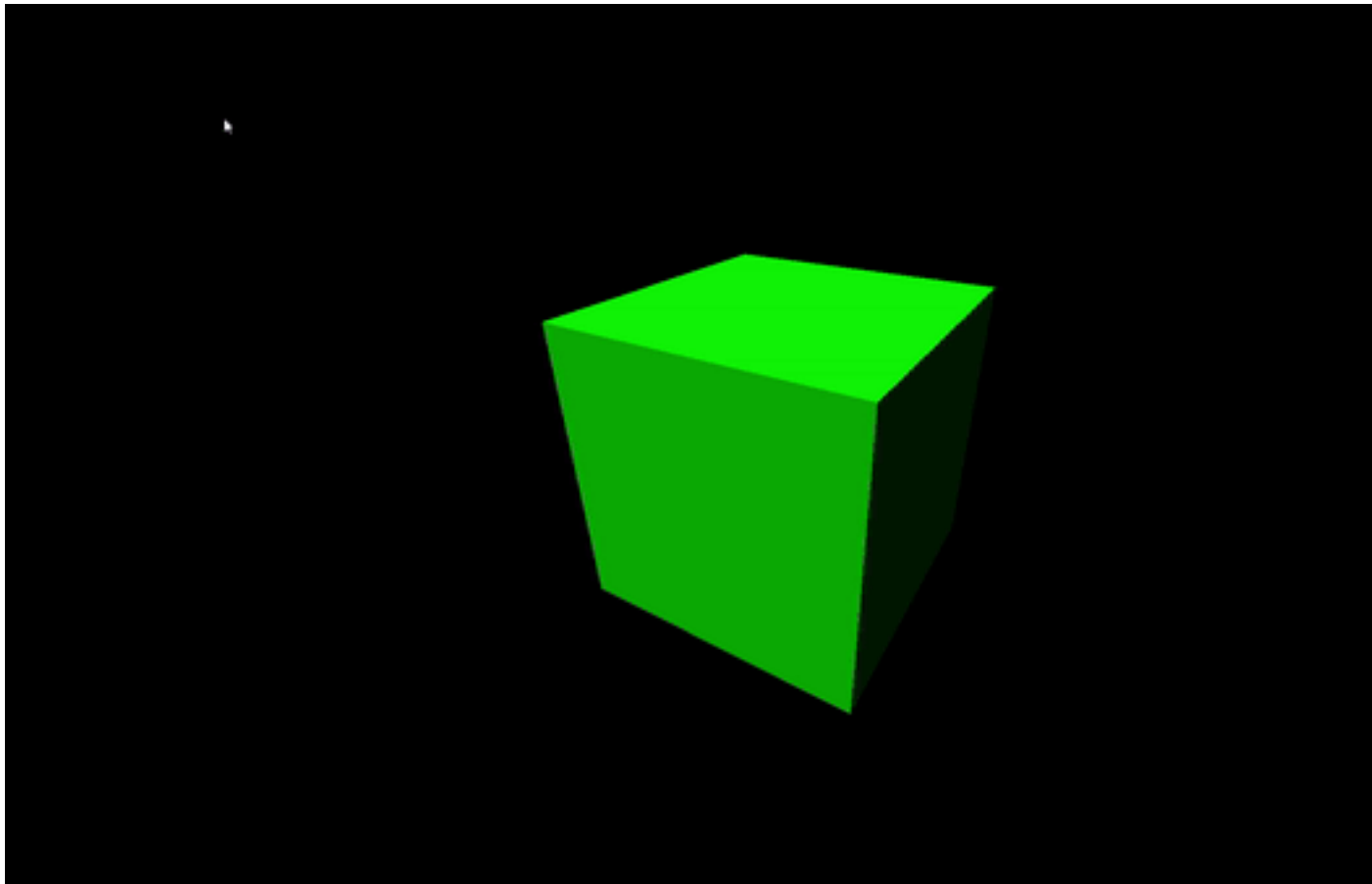
MeshPhongMaterial 으로 바꿔주면 그림자가 적용된다.

18. 화면 resize

```
19. window.addEventListener("resize",
    resize, false);
20.
21. function resize() {
22.
    renderer.setSize(window.innerWidth,
    window.innerHeight);
23.     camera = new
    THREE.PerspectiveCamera(
24.         75,
25.         window.innerWidth /
    window.innerHeight,
26.         0.1,
27.         5
28.     );
29.     camera.position.z = 2;
30. }
```

코드 첫부분에 위 코드를 추가하고, const camera 를 선언했던 부분의 const 를 let 으로 바꿔주면 된다.

이제 진짜 정사각형이 잘 굴러가는데 보인다.



아래는 완성된 코드다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My first three.js
app</title>
    <style>
      html, body{
        margin: 0;
        height: 100%;
      }
    </style>

  </head>
  <body>
```

```
<canvas id="c"></canvas>
```

```
<script  
src="./three.js"></script>
```

```
<script>  
    window.addEventListener('resize',  
resize, false);
```

```
    function resize() {  
        renderer.setSize( window.innerWidth,  
window.innerHeight );  
        camera = new  
THREE.PerspectiveCamera(75,  
window.innerWidth / window.innerHeight,  
0.1, 5);  
        camera.position.z = 2;  
    }
```

```
    const canvas =  
document.querySelector('#c');  
    const renderer = new  
THREE.WebGLRenderer({canvas});  
    renderer.setSize( window.innerWidth,  
window.innerHeight );
```

```
    let camera = new  
THREE.PerspectiveCamera(75,  
window.innerWidth / window.innerHeight,  
0.1, 5);
```

```
        camera.position.z = 2;  
//카메라 위치 이동, default 로 아래쪽 바라봄.
```

```
        const scene = new  
THREE.Scene();
```

```
        const geometry = new  
THREE.BoxGeometry();
```

```
        const material = new  
THREE.MeshPhongMaterial({color:  
0x00ff00});
```

```
        const mesh = new  
THREE.Mesh(geometry, material);  
        scene.add(mesh);
```

```
        {  
            const color = 0xFFFFFF;  
            const intensity = 1;  
            const light = new  
THREE.DirectionalLight(color, intensity);  
            light.position.set(-1, 2,  
4);  
            scene.add(light);  
        }
```

```
        function render(time) {  
            time *= 0.001; //  
convert time to seconds
```

```
            mesh.rotation.x = time;  
            mesh.rotation.y = time;
```

```
renderer.render(scene,  
camera);
```

```
requestAnimationFrame(render);  
}  
requestAnimationFrame(render);
```

```
</script>  
</body>  
</html>
```

D3.js란?

안녕하세요. 자바스크립트 라이브러리 중 하나인 D3.js에 대해

D3.js란 자바스크립트 라이브러리를 처음 들어보신 분도 또는
들을 위해 조금 자세하고 쉽게 설명해드리도록 하겠습니다.

(이미 자주 사용하시고 저보다 더 많은 지식을 알고 계시는 분들

우선 D3는(D3.js를 줄여서 D3라고 함) 완전히 오픈소스이며,

D3라는 명칭으로 부르며, 웹브라우저상에서 동적이고 인터랙

D3의 장점은 새로운 언어 형태가 아니라 자바스크립트 문법을

쉽게 접근이 가능하며, 각종 차트에 필요한 기능들을 함수 단위

요소 검사기를 활용해서 쉽게 디버깅을 할 수 있다는 것입니다

또한 D3는 2012년 8월에 2 버전이 출시되었고 2016년에 4버

<https://d3js.org>)를 보면 대부분의 차트들이 3버전으로 되어

더 자세한 내용이 궁금하시다면 D3 공식 사이트(<https://d3js.org>)

D3 환경 세팅

D3 환경 세팅을 하는 방법에는 두 가지가 있습니다.

첫 번째는, 공식사이트(<https://d3js.org/>)에서 D3를 다운받

두 번째는, 아래와 같이 CDN방식으로 D3를 사용하는 방법

```
<script src="http://d3js.org/d3.v3.js"></script>
```

두 가지 방법 중 편한 방법을 선택해서 사용하게 되면 D3를 사용하는데 문제가 없습니다. 하지만 인터넷 연결이 끊기게 되면 D3를 사용할 수 없기 때문

D3 기본 문법

D3를 사용하려면 기본 문법은 알고 있어야겠죠???

- `d3.select` - 특정 태그 하나를 선택한다.
- `d3.selectAll` - 특정 태그 전체를 선택한다.
- `selection.attr` - 선택 태그의 속성값 지정한다.
- `selection.data` - 집어넣을 즉 차트에 사용할 데이터를 가져
- `selection.enter` - 데이터 개수만큼 태그가 부족할 시에 추가
(2개를 더 추가한다.)
- `selection.append` - 새로운 태그를 추가한다.

위의 6가지 문법은 D3를 사용하는데 있어 가장 기본적인 문법
(<https://github.com/zziuni/d3/wiki/API-Reference>)를 참고

D3 기본 차트 그리기1

```
1  <!DOCTYPE html>
2  <html lang="ko">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1">
6    <meta http-equiv="X-UA-Compatible" content="ie=edge">
7    <title>D3 테스트</title>
8  </head>
9  <body>
10   <svg width="500" height="500"></svg>
11   <script src="http://d3js.org/d3.v3.js"></script>
12   <script>
13     const data = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50];
14
15     const svg = d3.select('svg');
16
17     data.forEach((d, i) => {
18       svg.append('rect')
19         .attr('height', data[i])
20         .attr('width', 40)
21         .attr('x', 50 * i)
22         .attr('y', 100 - data[i])
23     })
24   </script>
25 </body>
26 </html>
```

```
<svg width="500" height="500">
  <rect height="5" width="40" x="0" y="95"></rect>
  <rect height="10" width="40" x="50" y="90"></rect>
  <rect height="15" width="40" x="100" y="85"></rect>
  <rect height="20" width="40" x="150" y="80"></rect>
  <rect height="25" width="40" x="200" y="75"></rect>
  <rect height="30" width="40" x="250" y="70"></rect>
  <rect height="35" width="40" x="300" y="65"></rect>
  <rect height="40" width="40" x="350" y="60"></rect>
  <rect height="45" width="40" x="400" y="55"></rect>
  <rect height="50" width="40" x="450" y="50"></rect>
</svg>
```

한눈에 이해가 가시나요?? 이해가 한 번에 되지 않을 수도 있겠네요

10라인을 보시면 우선 높이와 넓이가 500인 SVG 태그를 하나 선언합니다.

11라인은 D3를 사용하기 위해 CDN 방식을 사용하였습니다.

13라인은 차트에 사용할 데이터를 배열 형태로 미리 만들어 놓습니다.

(간단한 예제이기 때문에 다음과 같이 배열 형태로 만들어 두어도 됩니다.)

형식으로 만들어 사용하며 이 JSON 데이터를 parsing 해서 사용합니다.

15라인은 d3의 문법 중 하나이며 SVG 라는 태그를 선택하라는 의미입니다.

17~22라인을 보시면 data 배열의 수 만큼 반복문을 실행하며

각각의 data를 이용하여 bar의 차트를 생성하게 됩니다. 자세히 보시면 jQuery처럼 d3가

자 이제 코드를 작성했으니 결과물을 확인해 봐야겠죠??



D3의 기본 문법을 이용하여 기본적인 막대차트를 그려 보았습니다.

결과물은 화면에 잘 출력이 되었지만, 혹시 여기서 의문점이 생길 수 있습니다. 위의 코드는 자바스크립트의 문법입니다. D3는 데이터 기반으로, 쉽게 데이터를 처리하는 라이브러리입니다.

그럼 D3의 장점을 살려서 코드를 다시 작성해볼까요??

D3 기본 차트 그리기2

```
1  <!DOCTYPE html>
2  <html lang="ko">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1">
6    <meta http-equiv="X-UA-Compatible" content="ie=edge">
7    <title>D3 테스트</title>
8  </head>
9  <body>
10    <svg width="500" height="500"></svg>
11    <script src="http://d3js.org/d3.v3.js"></script>
12    <script>
13      const dataFile = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50];
14
15      const svg = d3.select('svg');
16
17      svg.selectAll('bar')
18        .data(dataFile)
19        .enter().append('rect')
20        .attr('height', (d, i) => { return d })
21        .attr('width', 40)
22        .attr('x', (d, i) => { return 50 * i })
23        .attr('y', (d, i) => { return 100 - dataFile[i]})
24    </script>
```

위의 코드가 D3의 장점을 살려서 새롭게 작성한 코드입니다.

기존의 자바스크립트 코드인 반복문을 이용하지 않고 `.data()` 에서 도형을 생성하게 됩니다. 또한, 데이터를 가져오는 방식을 오는 값을 사용할 수도 있습니다.

D3 기본 차트 그리기3(디자인)

그럼 다음은 좀 더 발전된 차트를 그려 보기 전에 차트에 디자인

```
1  <!DOCTYPE html>
2  <html lang="ko">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1">
6    <meta http-equiv="X-UA-Compatible" content="ie=edge">
7    <title>D3 테스트</title>
8  </head>
9  <body>
10   <svg width="500" height="500"></svg>
11   <script src="http://d3js.org/d3.v3.js"></script>
12   <script>
13     const dataFile = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50];
14     const color = ['skyblue', 'blue', 'lime', 'red',
15                   'yellowgreen', 'violet', 'blueviolet',
16                   'chocolate', 'darkgreen', 'yellow']
17     const svg = d3.select('svg');
18
19     svg.selectAll('bar')
20       .data(dataFile)
21       .enter().append('rect')
22       .attr('fill', (d, i) => { return color[i] })
23       .attr('height', (d, i) => { return d })
24       .attr('width', 40)
25       .attr('x', (d, i) => { return 50 * i })
26       .attr('y', (d, i) => { return 100 - dataFile[i] })
27
28   </script>
29 </body>
30 </html>
```

```
▼ <svg width="500" height="500">
  <rect fill="skyblue" height="5" width="40" x="0" y="95"></rect>
  <rect fill="blue" height="10" width="40" x="50" y="90"></rect>
  <rect fill="lime" height="15" width="40" x="100" y="85"></rect>
  <rect fill="red" height="20" width="40" x="150" y="80"></rect>
  <rect fill="yellowgreen" height="25" width="40" x="200" y="75"></rect>
  <rect fill="violet" height="30" width="40" x="250" y="70"></rect>
  <rect fill="blueviolet" height="35" width="40" x="300" y="65"></rect>
```

(전체 소스 코드)

```
<html>
  <head>

</head>
  <body>
    <svg width="500" height="500"></svg>
    <script src="http://d3js.org/d3.v3.js"></script>
    <script>
      const dataFile = [5,10,15,20,25,30,35,40,45,50];
      const svg = d3.select('svg');
      const color = ['skyblue','blue','lime','red',
        'yellowgreen','violet','blueviolet',
        'chocolate','darkgreen','yellow']
      svg.selectAll('bar')
        .data(dataFile)
        .enter().append('rect')
        .attr('fill', (d,i) => { return color[i] })
        .attr('height', (d,i) => { return d })
        .attr('width', 40)
        .attr('x', (d,i) => { return 50 * i })
        .attr('y', (d,i) => { return 100 - dataFile[i] })
    </script>
  </body>
</html>
```

어느 부분이 달라졌는지 파악하셨나요??

14라인에 color라는 새로운 배열이 생겼고, 22라인에 fill이라

무슨 기능인지 알아보까요??

우선 D3의 fill은 CSS의 background-color와 같은 기능을 합
사용하게 됩니다.

이외에도 D3에서 사용하는 고유 CSS 문법이 있지만 자세한 건
니다.

다시 코드로 돌아가서 22번째 라인에서 rect 차트에 color 배
연결하는 게 가장 좋은 방법이지만 D3에서는 다음과 같이 인
리기 위해 다음과 같은 방법을 사용하였습니다.

자 이제 결과물을 볼까요?????



웹차트 만들기 1편에 이어 응용 편으로 파이그래프

파이그래프 그리기



먼저, 파이그래프의 전체적 윤곽을 잡는 코드와 설

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta http-equiv="Content-type" content="text/html; charset=utf-8">
5     <title>파이차트</title>
6     <script src="https://d3js.org/d3.v3.min.js"></script>
7   </head>
8   <body>
9     <div id="pie"></div>
10    <script type="text/javascript">
11      var width = 450,
12          height = 450,
13          outerRadius = Math.min(width, height) / 2,
14          innerRadius = outerRadius * .5,
15          color = d3.scale.category20();
16
17      var dataset = [
18        {name:"동화약품", value:238500},
19        {name:"CJ대한통문", value:160500},
20        {name:"두산", value:100500},
21        {name:"대림산업", value:99900},
22        {name:"삼양홀딩스", value:78100}];
23
24      var vis = d3.select("#pie")
25        .append("svg:svg")
26        .attr("width", width)
27        .attr("height", height)
28        .append("svg:g")
29        .attr("transform", "translate(" + outerRadius + "," + outerRadius + ")")
30        .data([dataset]);
31
32      var arc = d3.svg.arc().innerRadius(innerRadius).outerRadius(outerRadius);
33
34      var pie = d3.layout.pie().value(function(d) { return d.value; });
```

[line6]

D3 라이브러리를 사용하기 위해서 해당 스크립트

```
<script src="https://d3js.org/d3.v3.min.js"> </script>
```

[line11 ~ 15]

파이차트를 그려내기 위해서 고정적으로 사용할 변수

- 파이차트를 그려내기 위한 바깥쪽반지름(outerRadius)
- d3.scale.category20() 으로 d3 표준색상을 지정

[line17 ~ 22]

파이차트에서 사용할 데이터를 dataset변수에 할당

select("#pie")로 id값이 pie인 요소를 선택하고 [line31]

그 자식으로 <svg width=450, height=450></svg> 요소를 생성하고
(.append()함수로 가장 끝에 자식 요소를 추가 합니다.)

그 다음 svg 자식으로 <g transform="translate(0,0)"></g> 요소를 생성하고
(.attr("transform", "translate(" + x축이동거리 + "," + y축이동거리 + ")")로 이동거리를 지정합니다.)

마지막으로 사용할 데이터를 data()함수를 이용해 data 배열로 지정합니다.

[line32]

후에 그려 낼 파이차트의 속성값을 정의 합니다.

outerRadius(바깥쪽 크기), innerRadius(안쪽 크기), startAngle(시작 각도), endAngle(종료 각도)

(예를 들면, `.innerRadius(0).outerRadius(50)`의

[line34]

파이차트의 특별한 형식에 맞는 값으로 추출 할 수
와 파이차트의 조각들을 채울 것입니다.

다음은 실질적으로 파이차트를 그려내는 코드와 설

```
35
36     var arcs = vis.selectAll("g.slice")
37         .data(pie)
38         .enter()
39         .append("svg:g")
40         .attr("class", "slice");
41
42     arcs.append("svg:path")
43         .attr("d", arc)
44         .attr("fill", function(d, i) { return color(i); });
45
46     arcs.append("svg:text")
47         .attr("dy", ".35em")
48         .attr("text-anchor", "middle")
49         .attr("transform", function(d) { return "translate(" + arc.centroid(d) + ")"; })
50         .text(function(d) { return d.data.name; });
51
52     function angle(d) {
53         var a = (d.startAngle + d.endAngle) * 90 / Math.PI - 90;
54         return a > 90 ? a - 180 : a;
55     }
56
57     vis.append("svg:text")
58         .attr("dy", ".35em")
59         .attr("text-anchor", "middle")
60         .text("주식")
61         .attr("class", "title");
62 </script>
```

[line42 ~ 44]

각 `<g class="slice"></g>`요소 자식으로 `<path></path>`요소를 추가해주고 "fill"속성을 이용해 파이차트의 색상을 칠해줍니다.

[line46 ~ 55]

그려진 파이차트 안에 데이터 값을 할당하고 위치 값을 지정해줍니다.
function(d)를 사용해서 파이그래프에 나타날 데이터 값을 지정해줍니다.
나머지 라인은 `<text></text>`를 추가하고[line46] ~ [line55]

[line57 ~ 61]

파이그래프 정중앙에 "주식"이라는 텍스트를 추가해줍니다.
파이그래프 그리기는 여기서 마치고 기회가 된다면

웹팩을 이용해서 그래픽스 라이브러리를 타입스크립트로 번들링하고,

개발 환경을 편리하게 구성하기 위해 웹팩에 대해 공부한 기록을 담았습니다.

웹팩이란

웹팩은 HTML, CSS, JS, 등의 웹 어플리케이션에 필요한 자원을 개별 모듈로 정의하고,

이를 조합해서 하나의 HTML CSS JS 로 합쳐주는 모듈
번들러 입니다.

웹팩을 사용하면 얻게되는 장점이 무엇일까요?

우선 모듈화를 통해 JS 파일을 모듈화하여 코드의 가
독성을 높일 수 있습니다.

브라우저는 HTTP 요청을 동시에 6 개까지 수행할 수
있는데요 (크롬 기준)

모든 모듈 파일을 하나로 합침으로써 네트워크 자원
을 아끼며 로딩 속도를 더욱 빠르게 하며

자동화 도구들을 통해 웹 개발을 편하게 수행할 수
있습니다.

또한 모듈화를 이용하여 Lazy Loading 을 통해 필요한
자원을 나중에 요청할 수 있어 속도를 더욱 향상시
킬 수 있습니다.

초기설정

우선 웹팩을 사용하기 위한 dependency 를 설치합니
다.

```
npm i webpack webpack-cli -D
```

타입스크립트를 이용하기 위해 관련된 dependency 를
설치합니다.

```
npm install --save-dev typescript ts-  
loader
```

그 후, 해당 프로젝트에 사용할 tsconfig 를 추가합니다.

```
{
  "compilerOptions": {
    "outDir": "./dist/",
    "noImplicitAny": true,
    "module": "es6",
    "target": "es5",
    "jsx": "react",
    "allowJs": true,
    "moduleResolution": "node",
    "strict": false,
    "allowSyntheticDefaultImports": true
  }
}
```

tsconfig.json 까지 설정이 완료되면 프로젝트 최상위에 webpack.config.js 를 구성하고, export 할 객체를 설정합니다.

웹팩 프로그램은 export 하는 설정 객체를 읽어서 해당 설정대로 구성을 하기 때문에,

config 파일에서 본인이 작성한 설정을 export 해야
합니다.

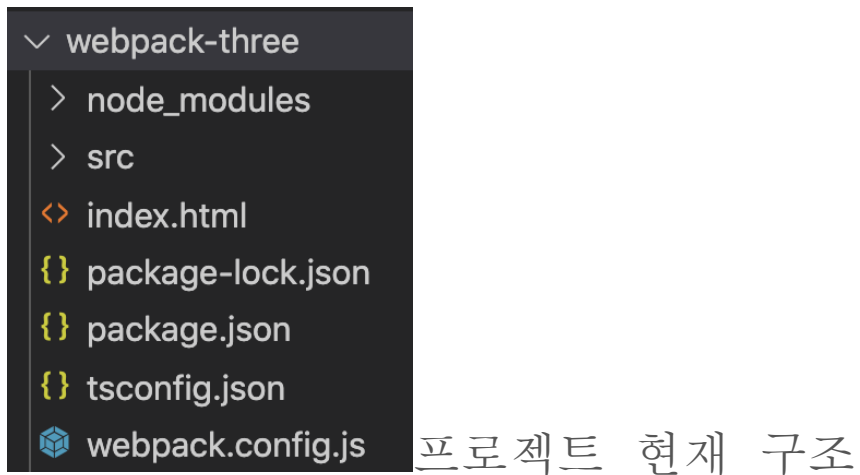
```
var path = require("path");
module.exports = {
  entry: "./src/index.ts",
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: "ts-loader",
        exclude: /node_modules/,
      },
    ],
  },
  resolve: {
    extensions: [".tsx", ".ts", ".js"],
  },
  output: {
    filename: "main.js",
    path: path.resolve(__dirname, "dist"),
  },
};
```

위 웹팩 설정을 반영하기 위해 최상위 폴더에 src
폴더를 추가하고, 그 안에 index.ts 를 추가합니다.

웹팩 설정이 완료되면, 최상위 폴더에 index.html 을
다음과 같이 작성합니다.

```
<html>  
<head>  
<title>Webpack Demo</title>  
</head>  
<body>  
<script src="dist/main.js"></script>  
</body>  
</html>
```

이렇게 하면, 프로젝트의 소스 폴더는 다음과 같이
구성 될 것입니다.



lodash 모듈을 설치한 후, src 폴더 내의 index.ts 에
다음 코드를 작성해봅시다.

(이 소스가 srcWinindex.ts 에 추가되면 된다. 22/1/19
일 체크)

```
// index.js
import _ from "lodash";
function component() {
  var element = document.createElement("div");
  /* lodash is required for the next line to work */
  element.innerHTML = _.join(["Hello", "World"], " ");
  return element;
}
document.body.appendChild(component());
```

작성이 완료되면, 이제 웹팩을 이용해서 빌드해봅시다.

Package.json 에 다음 명령어를 추가합니다.

```
"scripts": {  
  "build": "webpack --mode=none"  
},
```

(기존 소스 위에 위의 코드를 추가한다. 22/1/19 일 체크)

(아래의 lodash 모듈이 없다는 에러가 발생해서 아래 구문을 실행한다.)

```
npm install lodash
```

```
npm i --save-dev @types/lodash
```

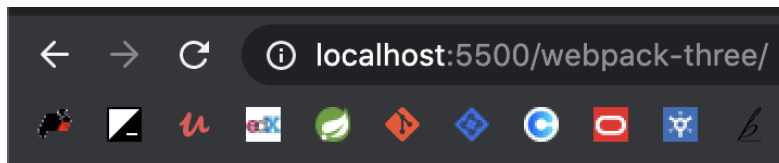
그 후 `npm run build` 명령어를 통해 웹팩을 이용하여 소스코드를 번들링합니다.

이를 수행하고 `live-server` 를 켜서 `index.html` 에 접근하면 다음과 같은 화면을 볼 수 있습니다.

(아래와 같이 `live-server` 를 설치하고 `live-server` 를 실행하면 웹서버가 실행된다. 22/1/19 일 체크)

```
npm install -g live-server
```

```
jongdeokkim@JONGui-MacBookPro DemoWebPack %  
live-server  
Serving "/Users/jongdeokkim/Desktop/DemoWebPack"  
at http://127.0.0.1:8080
```



Hello World

기존 index.html 에서 번들링된 main.js 파일을 불러와 화면에 Hello World 를 표시한 것입니다.

이와 같이 쉽게 타입스크립트 모듈을 자바스크립트로 번들링할 수 있습니다.

그럼 이제 구체적인 설정 방식에 대해서 알아보시다.

웹팩의 4 가지 주요 속성

entry - 자바스크립트의 진입점이라고 할 수 있습니다.

html 에서 js 를 불러오면 그때 페이지가 렌더링되는

데, 불러오는 개별 js 모듈의 entry 를 지정하여 서로 다른 js 번들을 구성할 수 있습니다.

output - 웹팩을 실행한 결과물의 파일 경로를 의미하며, filename 과 path 를 정해줍니다. output filename 엔 entry, 모듈 ID, 해시 등을 적용할 수 있습니다.

loader - 자바스크립트가 아닌 파일에 대한 변환을 도와줍니다. (HTML, CSS, Image, 폰트 등) 타입스크립트 파일도 ts-loader 를 이용하여 js 로 로드합니다.

module 의 rules 영역에 개별 객체로 지정해주며, test 항목에 파일의 확장자를, use 항목에 loader 의 이름을 지정해줍니다.

loader 를 적용할 때 한 파일에 여러 로더를 적용하게 되면, loader 의 순서를 올바르게 지정해야 합니다.

(오른쪽에서 왼쪽의 순서로 로딩됨)

plugin - 웹팩의 기본적인 동작에 추가적인 기능을 제공합니다. 결과물의 형태를 바꾸거나, 편의성을 제공하는 등의 요소들은 플러그인으로 구현되어 있습니다.

추가적인 속성

Resolve - 경로 및 확장자를 처리할 수 있게 도와줍니다. 정확히 말하면, js 에서 모듈을 불러올 때

(import ~ from ~) 이걸 정확히 어떤 위치에서 어떻게 로드할지를 지정을 해주는 것입니다.

alias 를 두어 지정한 경로를 alias 로 설정하여
import 할 수 있습니다.

```
resolve: { alias: { module:  
path.resolve(__dirname,  
"./app/module/") } }
```

폴더 구조가 복잡해질 때 import ../../../module/~~
로 상대 위치를 고려해서 import 했던 것을

단순히 module/~~ 로 작성해도 불러와지게끔 설정
을 해주는 것입니다.

Provide plugin 을 사용해서 특정 모듈을 전역으로 사
용할 수 있게 만들 수 있는데, JQuery 나 lodash 와

같이 일반적으로 사용되는 모듈을 그렇게 설정할 수 있을 것입니다.

```
plugins: [ new webpack.ProvidePlugin({ $: "jquery" }) ]
```

Webpack Dev Server

개발을 하면서 매번 빌드하고 그 결과물을 재확인하는 과정이 그렇게 편하지는 않을 것입니다.

Dev 서버를 이용해서 매번 빌드하지 않고 결과물을 볼 수 있게 해주는 플러그인을 사용해봅시다.

먼저 필요한 패키지를 설치해줍니다 .

```
npm install webpack-dev-server -D
```

다음으로, devServer 에 대한 설정을 webpack.config.js 에 추가해줍니다.

```
module.exports = {  
  ...  
  devServer: {  
    port: 5000,  
  },  
};
```

여러 설정이 있겠지만, 여기서는 port 만 지정하겠습니다.

그 후, package.json 의 script 에 dev 명령어를 추가해줍니다.

```
"scripts": {  
  "build": "webpack --mode=none",  
  "dev": "webpack-dev-server"  
},
```

추가가 완료되면 npm run dev 를 이용해 명령어를 실행시켜줍니다. 실행하게 되면 다음과 같이 뜹니다.

```
> webpack-three@1.0.0 dev /Users/riverandeye/Desktop/KUCC/web-graphics/webp
> webpack-dev-server

i [wds]: Project is running at http://localhost:5000/
i [wds]: webpack output is served from /
i [wds]: Content not from webpack is served from /Users/riverandeye/Desktop
i [wdm]: Hash: 3440225b41d5b23b3719
Version: webpack 4.44.2
Time: 4446ms
```

웹팩 dev server 실행

그리고 나서, 웹팩에 지정한 source 들을 수정하고 실행하게 되면, 파일이 변경되면 (저장되면) 빌드되고

그 결과물이 localhost:5000 에 나타나는 것을 확인할 수 있습니다.

[2] Three.js - 시작하기

by Riverandeye 2020. 9. 22.

Three.js 는 자바스크립트 3d 라이브러리로 많은 곳

에서 사용되고 있습니다.

다양한 라이브러리가 있지만, Three.js 를 사용하는건
아무래도 커뮤니티가 크고 TypeScript 지원이 잘 된
다는 것이 그 이유입니다.

Three.js 엔 여러 장점이 있는데 우선 대부분의 브라
우저에서 동작하고

3d 를 구현하기 위해 개별 플러그인이 필요하지 않
습니다.

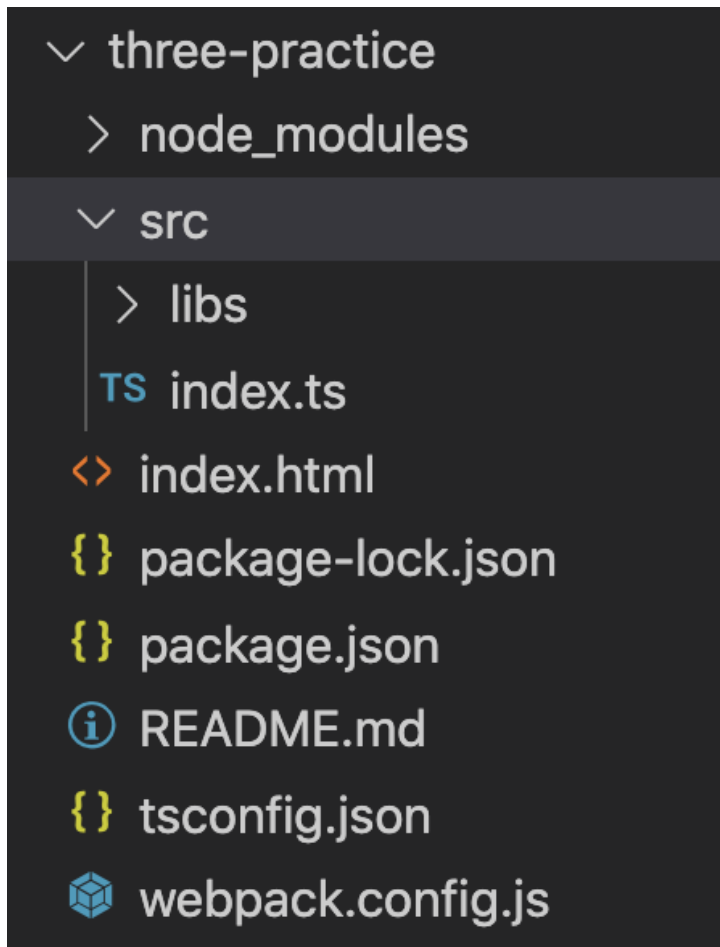
무엇보다 가장 큰 장점은 WebGL 을 몰라도 쉽게 사
용할 수 있다는 큰 장점이 있습니다.

WebGL 은 일반 캔버스와는 달리 GPU 자원을 사용할
수 있어 성능상의 이점을 보입니다.

저는 간단한 웹팩 설정으로 ts 를 이용하여 로컬에서 작업하였습니다. 설정 방식은 이 글을 참고하시면 좋을 것 같습니다.

설정하기

위 웹팩 설정을 완료했다면, 프로젝트의 구조는 다음과 같을 것입니다.



libs 폴더는

무시하셔도 됩니다

우선 index.ts 에 작업을 할 것이고, html 은 다음과 같

이 설정합니다.

```
<html>
<head></head>
<style>
body {
margin: 0;
overflow: hidden;
```

```
}  
</style>  
<body>  
<script src="main.js"></script>  
</body>  
</html>
```

body 에 적용된 css 는 webpack 의 css-loader 를 직접
추가해서서 개별 css 파일로 분리하셔도 좋습니다.

앞으로 작성하는 모든 코드는 index.ts 에 추가할 것
입니다.

따라하기

우선 간단하게 따라해봅시다.

0. three.js 패키지를 설치합니다. three 라이브러리 내에 타입 정의가 있기 때문에, 타입 definition 을 따로 설치하지 않아도 됩니다.

```
npm install three
```

1. main 함수를 추가하고, 해당 main 함수를 window.onload 에 할당합니다.

```
const main = () => {  
  }  
window.onload = main;
```

2. html 의 body 에 output 의 id 를 가진 간단한 div 를 추가합니다

```
const main = () => {  
  const output = document.createElement('div');  
  output.id = 'output'  
  document.getElementsByTagName("body")[0].appendChild
```

```
(div);  
}
```

3. three.js 를 import 하고, scene 과 renderer 를 선언합니다.

```
import * as three from "three"; // 최상위에 해줄 것  
const main = () => {  
  //...  
  const scene = new three.Scene();  
  const renderer = new three.WebGLRenderer();  
}
```

여기서 주목할 부분은, 렌더링해주어야 할 대상(scene)과 렌더링해주는 대상(renderer)이 분리되어 있습니다.

4. renderer 에 배경 색깔과 크기를 지정해주고, 그림자를 enable 해줍니다.

```
renderer.setClearColor(0xeeeeee);  
renderer.setSize(window.innerWidth,
```

```
    window.innerHeight);  
    renderer.shadowMap.enabled = true;
```

그림자를 넣는 작업은 자원을 많이 소모하여 default
로 false 가 설정되어 있기 때문에 true 로 변경해주
어야 합니다.

5. plane, sphere 과 cube 를 추가해줍니다.

```
const planeGeometry = new three.PlaneGeometry(60, 20);  
const planeMaterial = new  
three.MeshLambertMaterial({ color: 0xcccccc });  
const plane = new three.Mesh(planeGeometry,  
planeMaterial);  
const cubeGeometry = new three.BoxGeometry(4, 4, 4);  
const cubeMaterial = new  
three.MeshLambertMaterial({ color: 0xff0000 });  
const cube = new three.Mesh(cubeGeometry,  
cubeMaterial);  
const cubeGeometry = new three.BoxGeometry(4, 4, 4);  
const cubeMaterial = new  
three.MeshLambertMaterial({ color: 0xff0000 });  
const cube = new three.Mesh(cubeGeometry,  
cubeMaterial);
```

위에서 Geometry 와 Material 을 따로 구성한 후 하나의 Mesh 로 합치는 점에서, 이전에 학습했던 WebGL Fundamental 의 Vector Shader 와 Fragment Shader 가 분리되어 있는 것이 떠오릅니다.

6. plane, sphere, cube 의 그림자를 설정해줍니다. 그림자를 받기 + 그림자를 생성하기를 모두 추가해줍니다.

```
plane.castShadow = true;  
plane.receiveShadow = true;  
cube.castShadow = true;  
cube.receiveShadow = true;  
sphere.castShadow = true;  
sphere.receiveShadow = true;
```

7. plane, sphere, cube 의 위치를 지정해줍니다.

```
plane.position.set(15, 0, 0);  
cube.position.set(-4, 3, 0);  
sphere.position.set(20, 4, 2);
```

8. camera 를 생성하고, 카메라의 위치를 지정하고, 카메라의 시점을 조정합니다.

```
const camera = new three.PerspectiveCamera(45,  
window.innerWidth / window.innerHeight, 0.2, 1000);  
camera.position.set(-30, 40, 30);  
camera.lookAt(scene.position);
```

9. 광원을 생성하고, 광원의 위치를 지정하고, 광원에 의해 그림자가 생성되게 설정합니다.

```
const spotLight = new three.SpotLight(0xffffff);  
spotLight.position.set(-40, 60, -10);  
spotLight.castShadow = true;
```

10. 생성한 개별 객체들을 scene 에 추가해줍니다.

```
scene.add(plane);  
scene.add(cube);  
scene.add(sphere);  
scene.add(spotLight);
```

camera 는 render 할 때 camera 를 추가해줍니다. (렌

더링되는 영역 - 시야와 세계는 분리되어 구성됩니다)

11. Renderer 의 domElement 를 output div 의 하위 컴포넌트로 넣어줍니다. (꼭 이렇게 안해도 됨)

```
document.getElementById("output").appendChild(renderer.domElement);
```

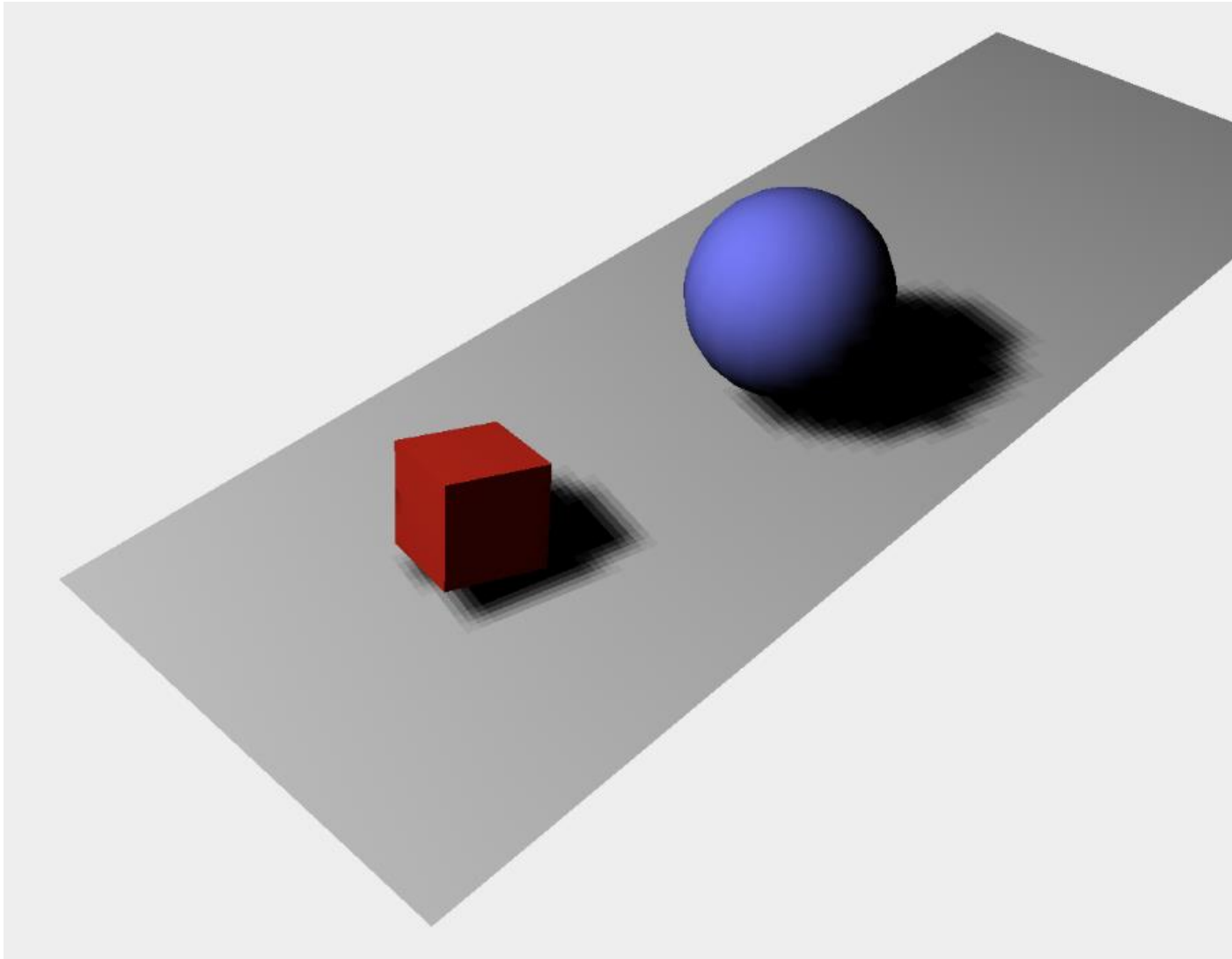
이렇게 하면, div id="output"의 하위 dom 객체로

renderer 가 렌더링하는 canvas 가 나타나게 됩니다.

12. 마지막으로, renderer 로 렌더링해줍니다.


```
renderer.render(scene, camera);
```

이렇게 하면 다음과 같은 화면을 볼 수 있을 것입니다.



보이는 것은 조금 다를 수 있음!

지금까지 무엇을 했는지 정리해보면

1. Scene 을 정의하였습니다.

2. Renderer 를 정의하였고, 그림자를 설정해주었습니다.

3. 각 Object 의 Geometry, Material 을 정의하고 position 을 지정하였으며 그림자를 설정해주었습니다.

4. 광원을 정의하고 위치를 지정한 후 그림자를 설정해주었습니다.

5. 카메라를 정의하고, 카메라의 위치 및 보는 방향을 지정하였습니다.

6. 렌더러를 돔에 추가하고, 렌더링하였습니다.

다음 글에는 각 컴포넌트의 정확한 동작 방식에 대해 이해하도록 해보겠습니다.

[3] Three.js - Scene, Light Sources
by Riverandeye 2020. 9. 23.

이전 시작 단계에서 간단한 튜토리얼을 통해

Three.js 를 어떻게 사용하는지 알아보는데요

이번 단계에서는 Three.js 의 Scene 을 구성하는데 사용되는 Object 와 역할

three.Scene Object 의 역할

geometric 과 mesh 와의 관계

orthographic camera 와 perspective camera 의 차이 를
알아보겠습니다.

Scene api

화면에 무언가를 보여주려면 다음 3 가지 요소가 필요
합니다.

- Camera -> 화면에 어떤 요소가 렌더링되는지를 결정
- Lights -> 물체가 어떻게 보이고, 그림자를 어떻게
생성하는지를 결정
- Objects -> Camera 에 비춰지는 물체들

Scene 객체는 이런 서로 다른 요소들을 담는 역할을 합니다.

Scene 에 추가된 Object 를 관리하는 메소드들은 많은데, 이전 예시에서 Scene 에 Object 를 추가하는 scene.add 메소드도 이에 포함됩니다.

대표적으로 다음이 있습니다.

Scene.add








Scene.remove

Scene.children

Scene.getBy...

이건 뭐 대단한 지식은 아니고, TS 를 사용하면 제공되는 메소드를 쉽게 참조하여 확인할 수 있습니다.

```
scene.get|
document.|
console.l
let step
const ren
  if (con
```

	getById	(method) Object3D.
	getByName	
	getByProperty	
	getWorldDirection	
	getWorldPosition	
	getWorldQuaternion	
	getWorldScale	

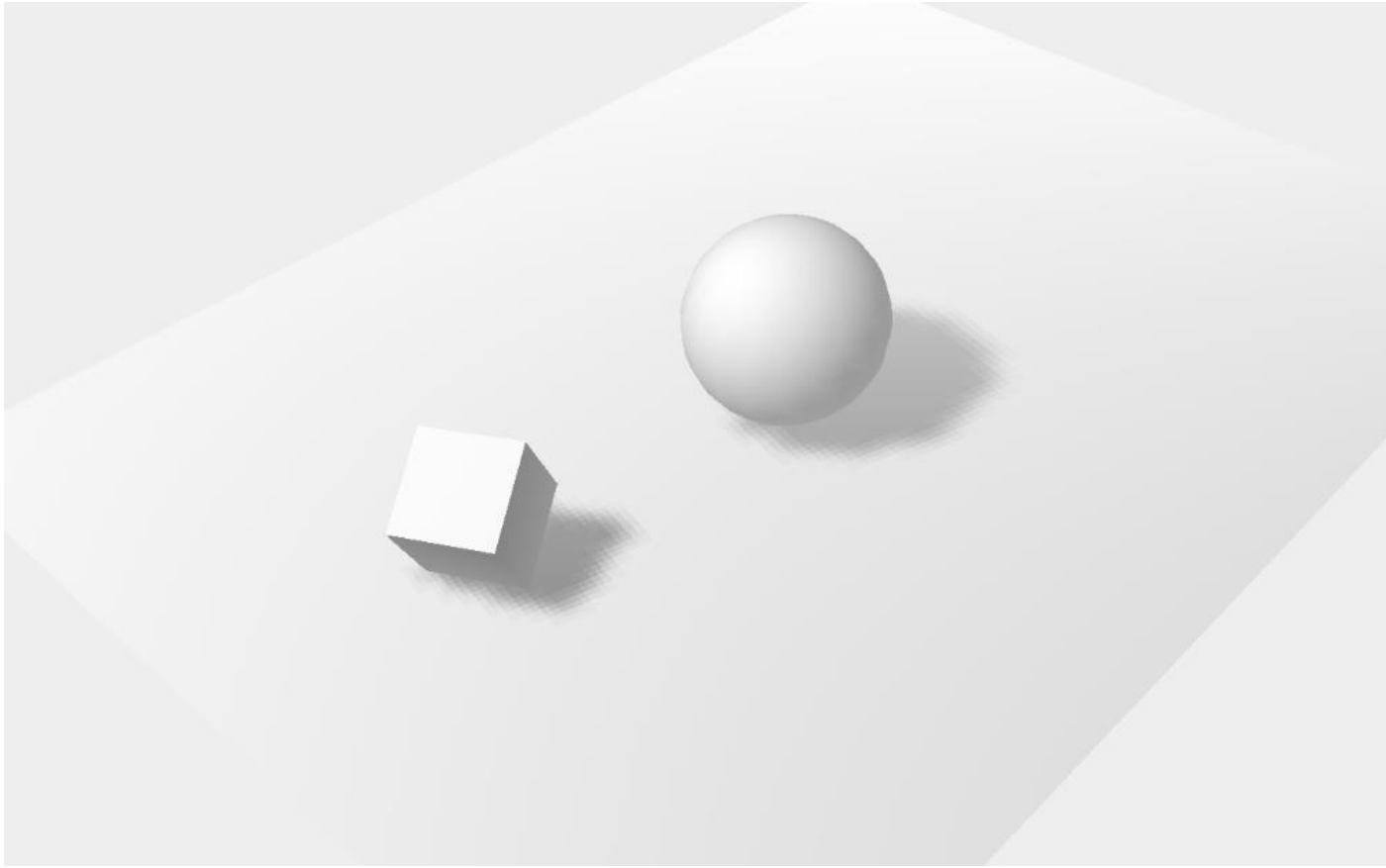
이런식으로..
타입으로 제공되는 api 를 적극 활용하고, 공식문서를
통해 학습하는 것이 좋은 방법일 수 있습니다.

Scene.traverse 는 함수를 입력받아 모든 Child element 에 적용합니다.

Scene.fog 를 통해 안개 효과를 추가할 수 있습니다.

Scene.overrideMaterial 을 이용하여 각 Child Object 에 적용된 Material 을 Override 할 수 있습니다.

```
scene.overrideMaterial = new  
three.MeshLambertMaterial({ color: 0xffffff });  
scene.fog = new three.Fog(0xffffff, 0.015, 100);
```

이전 예시에 `overrideMaterial` 추가

Geometries and meshes

geometry 는 3d space 의 point 의 집합을 의미하며,
이런 점들을 연결하는 삼각형의 면(mesh)들을 함께
의미합니다.

cube 의 경우 cube 는 8 개의 점으로 이루어져있고
한 면은 삼각형의 면 2 개로 구성됩니다.

Three.js 를 이용하면 일일이 vertice 와 face 를 정의할 필요 없이 해당 object 를 정의하는 key feature 만 정의하면 됩니다.

예를 들어 cube 는 width height depth 겠지요.

임의로 vector 와 face 를 정의한 후 이를 이용해서 Geometry 를 구성할 수 있습니다.

임의의 Geometry 를 구성한 후 computeFaceNormals 를 호출하여 각 face 의 normal vector 를 계산하여야 광원에 대해 비치는 빛을 구성할 수 있습니다.

translation

rotation

이런것들은 필요할때마다 따로 적용해보면서 공부하면 됩니다.

Orthographic camera vs Perspective camera

Perspective Camera 는 natural view 라고 생각하면 되고, 카메라의 위치에 대해 멀리 있을 수록 작게 렌더링된다.

Orthographic Camera 는 동일한 물체는 항상 동일한 크기로 보여진다. 카메라와 물체의 거리와는 상관 없이 말이다.

심시티나 롤러코스터타이쿤 이런 게임에

Orthographic Camera 가 적용된다.

Perspective Camera 엔 다음 속성을 적용할 수 있다.

fov (field of view) : 카메라의 위치에서 볼 수있는 장면의 각도 (시야각) 을 의미한다. 기본값은 50 이고, 일반적으로 60~90 도를 채택한다. 게임에서 해상도 선택시 시야가 달라지는 것은 곧 fov 가 달라진다고 볼 수 있다.

aspect - horizontal 과 vertical size 의 비율을 정한다.

일반적으로 $(\text{window.innerWidth} / \text{window.innerHeight})$ 를 사용한다.

near - 얼마나 가까이 있는 물체까지 렌더링해야 하는지를 정한다

far - 얼마나 멀리 있는 물체까지 렌더링해야 하는지를 결정한다.

zoom - 말그대로 zoom 이다.

Orthographic Camera 는 aspect ration 나 fov 에 관한 것이 아니라 모든 object 가 동일한 크기로 렌더링되기 때문에

렌더링되어야 하는 큐브 영역만 지정해주면 된다.

left, right, top, bottom 을 통해 렌더링되어야 하는 박스 영역을 지정해준다.

near 과 far 를 통해 박스의 width 를 지정해준다.

zoom 은 크기를 확대 혹은 축소한다.

Light Sources

WebGL 자체에 lighting 을 위한 Support 는 없고, 직접 구현해야 한다. (근데 엄청 힘들.. 참고)

Lights 의 종류

- AmbientLight : 기본적인 빛, global 하게 빛이 적용됨.

방향이란 개념이 없음.

- PointLight : Single Point 로부터 발산되는 빛 (Shadow 를 형성할 수 없음)

- SpotLight : 스포트라이트 같이 원뿔형 불빛이 나타남
- DirectionalLight : 평행하는 빛이 비추어진다 (멀리서 오는 태양빛처럼)
- HemisphereLight : 반구 형태로 빛을 전파함
- AreaLight : single point 대신 사용되어
- Lensflare : 빛은 아니지만 Light 에 Lensflare 효과를 지정할 수 있다.

[4] Three.js - Material
by Riverandeye 2020. 9. 28.

Material 은 Object 의 피부같은 것으로, Geometry 가 투명한지, Metallic 한지 "피부"같은 것을 결정한다.

다양한 Material 이 있는데, 대표적으로 다음과 같다.

MeshBasicMaterial -> 단순한 색깔 및 와이어프레임을
부여

MeshDepthMaterial -> 카메라와의 거리를 이용해 색상을
결정한다

MeshNormalMaterial -> Normal Vector 방향을 색상으로
표현함. 해당 지점의 법선 벡터가 가르키는 방향에
대해 색상이 변경된다.

MeshFaceMaterial -> 사용자가 개별 Face 에 대해 독립적인 material 을 지정할 수 있음

MeshLambertMaterial -> Vertice 에서 lighting 을 계산하여 Shiny 하지 않음

MeshPhongMaterial -> 모든 픽셀에 대해 lighting 을 계산하여 Shiny 하게 보인다.

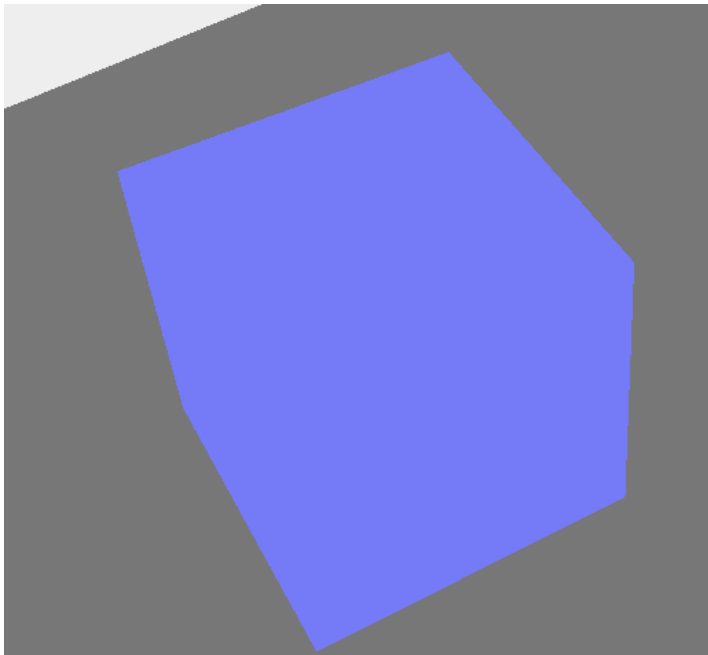
ShaderMaterial -> 직접 Shader Program 을 작성하여 어떻게 Vertice 가 위치하고 픽셀이 색칠되는지를 결정한다.

LineBasicMaterial -> Line Geometry 에 사용되는 Material

LineDashMaterial -> Line 에 Dash 추가해줌.

각각을 직접 보면서 어떤 질감을 띄는지 이해해보자.

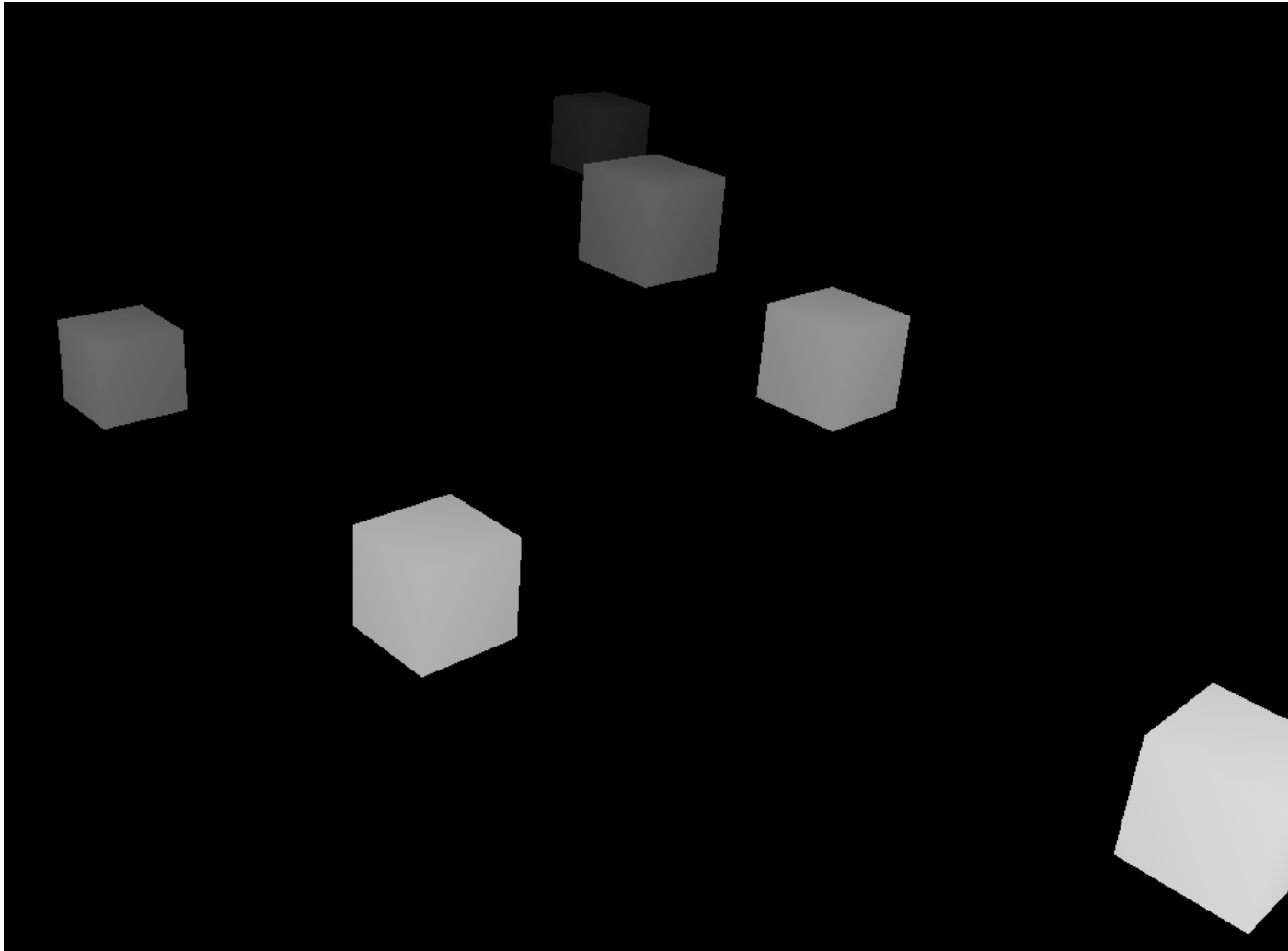
MeshMaterial



BasicMeshMaterial

카메라나 빛의 위치에 대해 표면의 색상 변화가 없다. 이게 기본형.

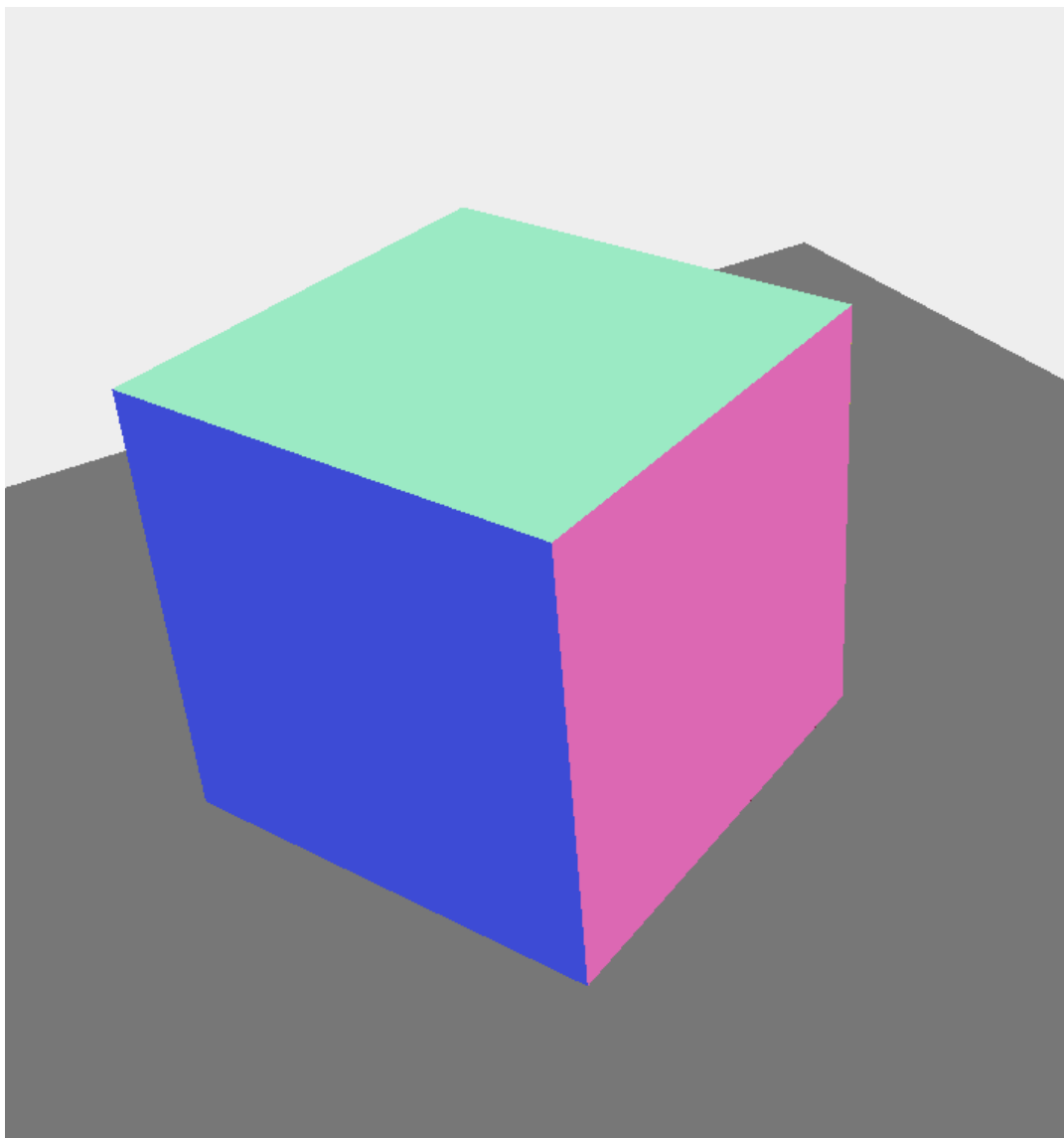
DepthMaterial



카메라와의 거리에 따라 빛의 크기가 달라진다.
AmbientLight 를 준 상태에서 DepthMaterial 은 다음과 같이 카메라와의 거리에 따라 색상이 달라보인다.

멀수록 잘 안보이는 그런 느낌을 준다.

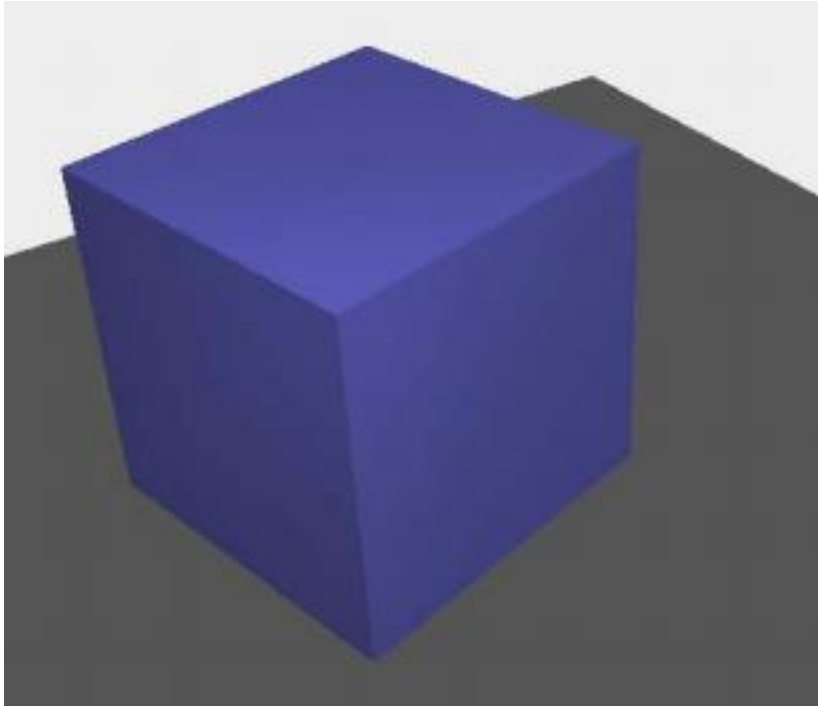
MeshNormalMaterial



표면이 가리키는 방향에 따라 색상이 달라진다.
x 축을 R, y 축을 G, z 축을 B 로 매핑해서, 현재 표면
의 법선벡터가 가리키는 방향으로 색상을 표현한다.

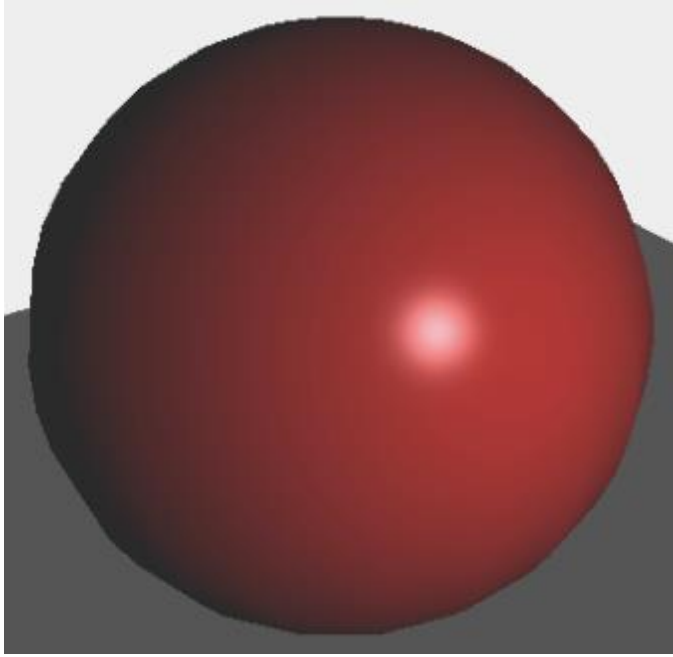
디버깅할때 사용하면 좋을듯..? 어떤 용도로 쓸지는
아직 감이 안온다.

MeshLambertMaterial



다소 밋밋한
입체
빛에 따라 색상이 변하지만, 다소 밋밋하고 광이 나
지 않는다.

MeshPhongMaterial



광이 난다

MeshLambertMaterial 보다 다소 광이 나는 것을 확인할 수 있다.

Material 과 Geometry 는 아무래도 직접 Blender 로 모델링을 한 후 Import 해서 가져오게 될 것이므로

라이브러리에서 지원하는 영역이 크게 중요하지 않아 간단하게 필요한 부분만 메모하고 넘어가려고 한다.

<https://continuous-development.tistory.com/80>