# Three.js시작하기

02 Apr 2021

사실은 three.js를 하기 전에 WebGL로 정육면체까지 만들고 싶었는데 아래 글을 읽다가 포기했다..

[webglfundamentals](https://webglfundamentals.org/webgl/lessons/ko/webgl-3d-orthographic.html)

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

# Three.js란?

WebGL을 사용해봤으면 알겠지만, WebGL로 뭔가를 렌더링 하려면 엄청난 양의 코드가 필요하다. 특히 3D라면 더더욱 어렵고 복잡하다. 그래서 WebGL로 좀 더 쉽게 3D렌더링을 할 수 있도록 Three.js, Scene.js같은 라이브러리들이 등장했다.

Three.js에선 Scene위에 객체들을 올리고, Camera를 통해 화면을 어떻게 렌더링할지 정한 후, Renderer가 Scene과 Camera정보를 이용해 3D씬을 렌더링하는 방식으로 작동한다.

Three.js의 구조를 잘 설명한 사진이 있어서 가져와봤다.

(출처: https://threejsfundamentals.org/threejs/lessons/kr/threejs-fundamentals.html)

# Three.js 시작

three.js를 시작하려면 일단 프로젝트에 three.js파일을 추가해야 한다. .js파일을 만들어 두고, 아래 링크의 코드를 복사한다음 붙여넣으면 된다.

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

그 후 아래와 같이 body태그에 포함시킨다.

(DemoThreeJS폴더에 제공된다.)

<**body**>

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

</**body**>

이제 Three.js를 사용해보면 된다.

나는 Three.js를 처음 사용할 때, 정육면체를 먼저 그려봤다.

1. 먼저, canvas를 html요소에 포함한다. canvas를 만들지 않아도 자동으로 canvas를 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 html파일에 script태그로 넣든 따로 js파일을 만들든 상관없다.

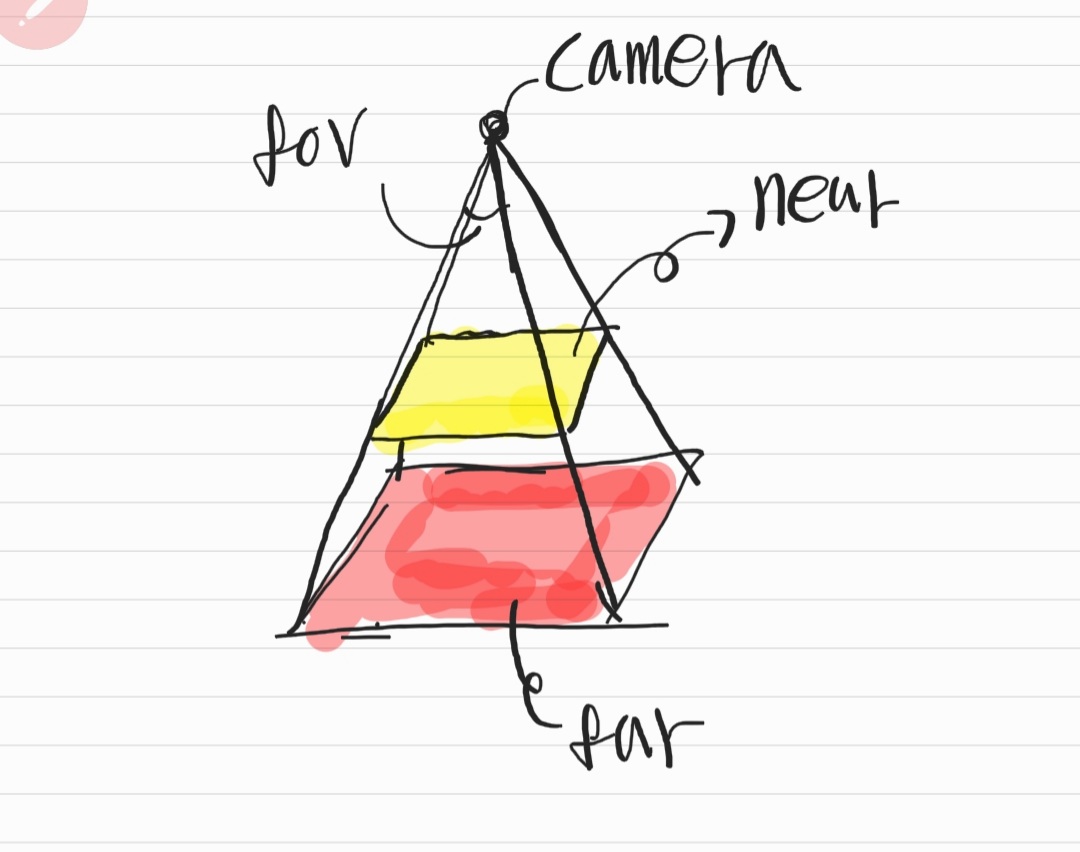
1. 만들어놓은 canvas를 불러오고, Three.js의 핵심이 되는 renderer를 canvas위에 생성한다.
2. **const** canvas = document.querySelector("#c");
3. **const** renderer = **new** THREE.WebGLRenderer({ canvas });

여기서 THREE가 전부 대문자인걸 주의하자. 나는 Three라고 써서 엄청 시간을 잡아먹었다..

1. 다음으로 Camera를 만들어보자.
2. **const** camera = **new** THREE.PerspectiveCamera(
3. 75,
4. window.innerWidth / window.innerHeight,
5. 0.1,
6. 5
7. );
8. camera.position.z = 2; //카메라 위치 이동, default로 아래쪽 바라봄.

Threejs에는 몇가지 카메라가 있는데, 여기서는 PerspectiveCamera를 사용했다. PerspectiveCamera의 인자는 차례대로 field of view(시야각), aspect, near, far이다.

* + field of view(시야각): 말 그대로 볼 수 있는 각도.
  + aspect: canvas의 크기 비율, canvas크기를 따로 설정하지 않으면 2가 기본값이다.
  + near, far: 랜더링 범위를 near~far로 제한



1. 다음으로는 Scene을 만들자.
2. **const** scene = **new** THREE.Scene();

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

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

1. 그럼 먼저 Geometry(형태)를 만들어보자.
2. **const** geometry = **new** THREE.BoxGeometry();

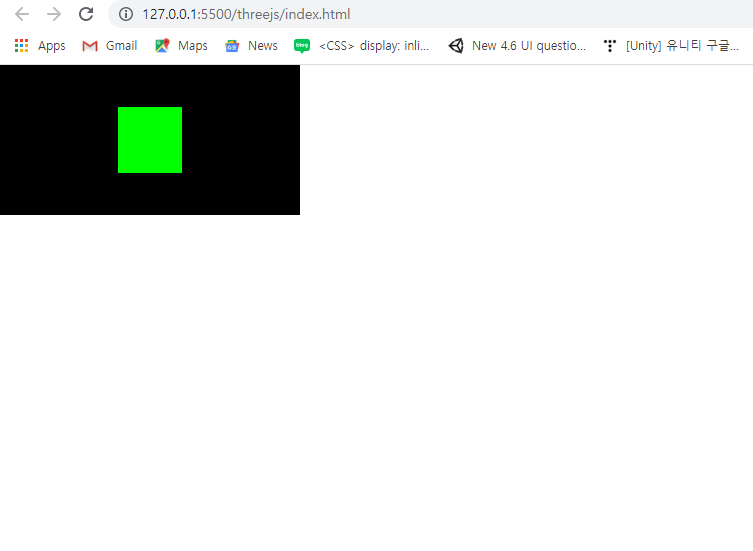
이 BoxGeometry가 정육면체의 Geometry이다.인자로 순서대로 boxWidth, boxHeight, boxDepth를 정해줄 수 있지만, 정해주지 않으면 기본적으로 1로 된다.

1. 다음으로 Material을 만들자.
2. **const** material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });

color외에 image를 줄 수도 있고 밝기를 줄 수도 있는데 일단은 color만 줬다.

1. 이제 Mesh를 만들고 Scene에 추가하자.
2. **const** mesh = **new** THREE.Mesh(geometry, material);
3. scene.**add**(mesh);
4. 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 camera를 만들때, canvas크기 비율이 아닌 화면 크기 비율을 줘서 그렇다.

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

1. 먼저 canvas크기를 맞춰보자. css로 html, body의 margin을 없애고, height:100%를 준다. 그 후
2. renderer.setSize(window.innerWidth, window.innerHeight);

이 코드를 const renderer을 선언한 부분 다음 줄에 집어넣자. 이제 새로고침하면 화면 크기에 딱 맞는 canvas가 된다.

화면 size가 변했을 때 canvas크기가 변하지 않아 되게 불편한데, 이건 마지막에 고쳐보자.

1. 이제 정육면체의 모든 면을 보기 위해 회전 시켜보자.
2. **function** **render**(time) {
3. time \*= 0.001;
4. mesh.rotation.x = time;
5. mesh.rotation.y = time;

renderer.render(scene, camera);

requestAnimationFrame(render);

}

requestAnimationFrame(render);

위 코드를 renderer.render(scene, camera)가 있던 부분을 지우고 추가한다. 여러 방법이 있는데, time을 쓰는 방법을 사용했다. mesh.rotation.x, y의 값을 적당히 잘 바꿔주기만 하면 된다.

requestAnimationFrame은 디스플레이 주사율에 맞춰 함수를 반복적으로 실행해주는데, setInterval함수와 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.

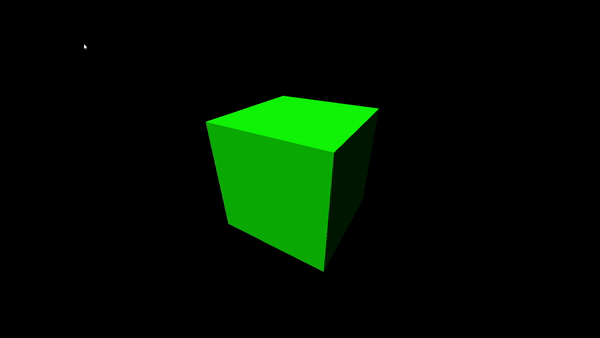
1. 광원추가
2. {
3. **const** color = 0xffffff;
4. **const** intensity = 1;
5. **const** light = **new** THREE.DirectionalLight(color, intensity);
6. light.position.**set**(-1, 2, 4); //광원 위치 설정
7. scene.**add**(light); //Scene에 추가
8. }

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

1. 화면 resize
2. window.addEventListener("resize", resize, false);
3. **function** **resize**() {
4. renderer.setSize(window.innerWidth, window.innerHeight);
5. camera = **new** THREE.PerspectiveCamera(
6. 75,
7. window.innerWidth / window.innerHeight,
8. 0.1,
9. 5
10. );
11. camera.position.z = 2;
12. }

코드 첫부분에 위 코드를 추가하고, const camera를 선언했던 부분의 const를 let으로 바꿔주면 된다.

이제 진짜 정사각형이 잘 굴러가는게 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, default로 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);

}

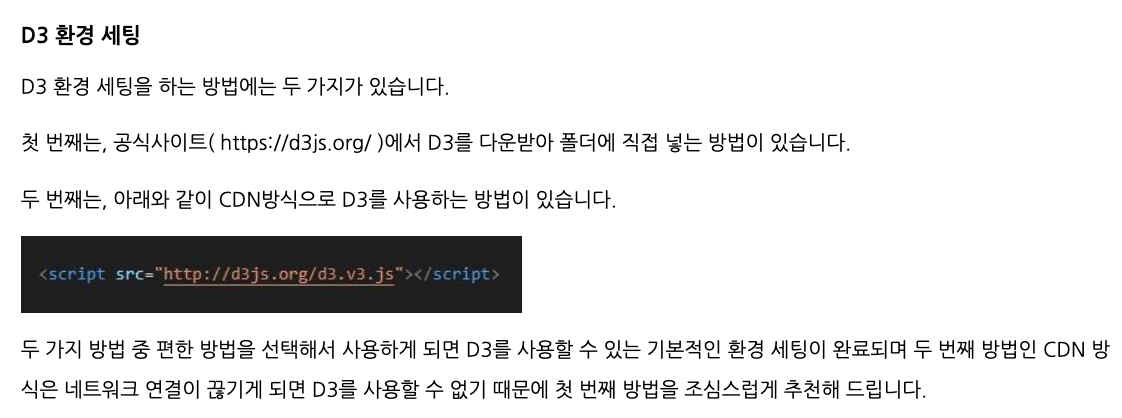
requestAnimationFrame(render);

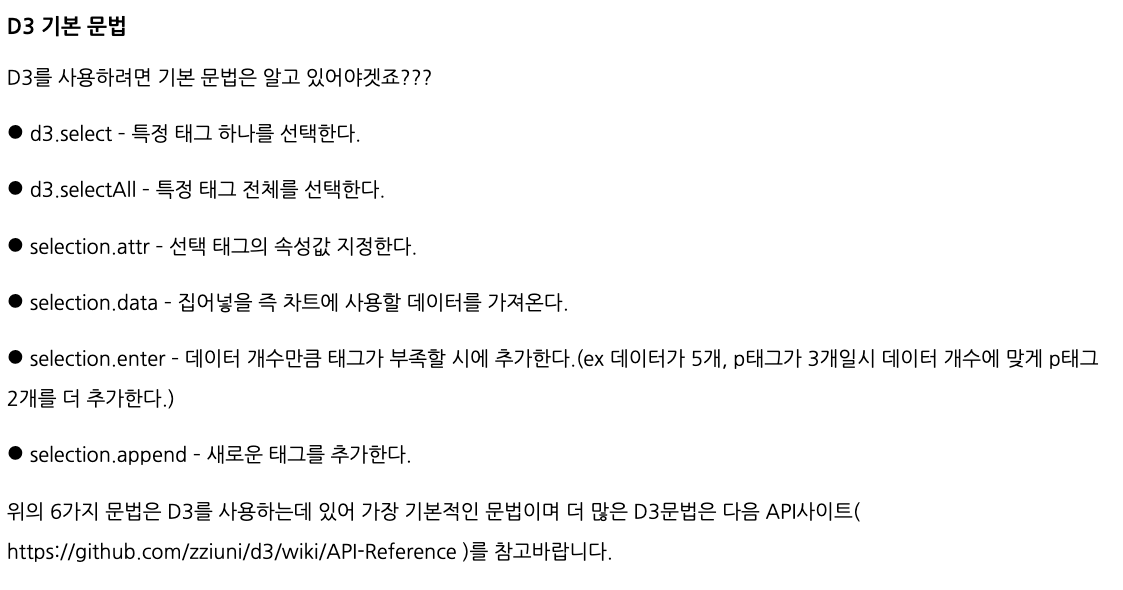
</**script**>

</**body**>

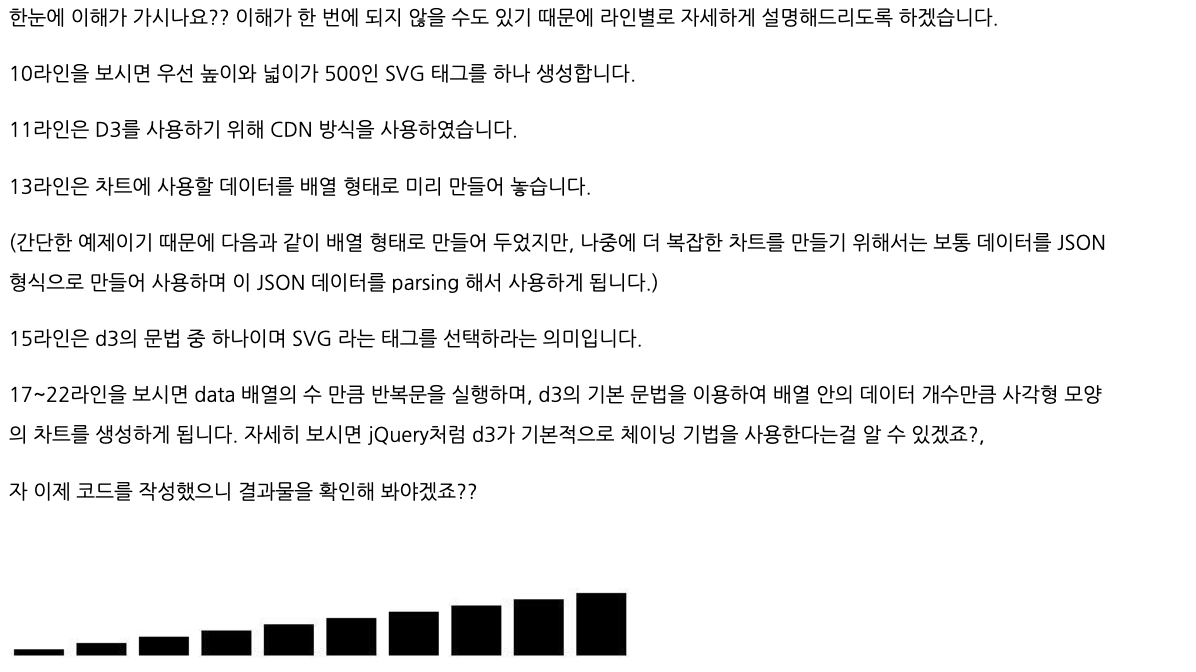
</**html**>

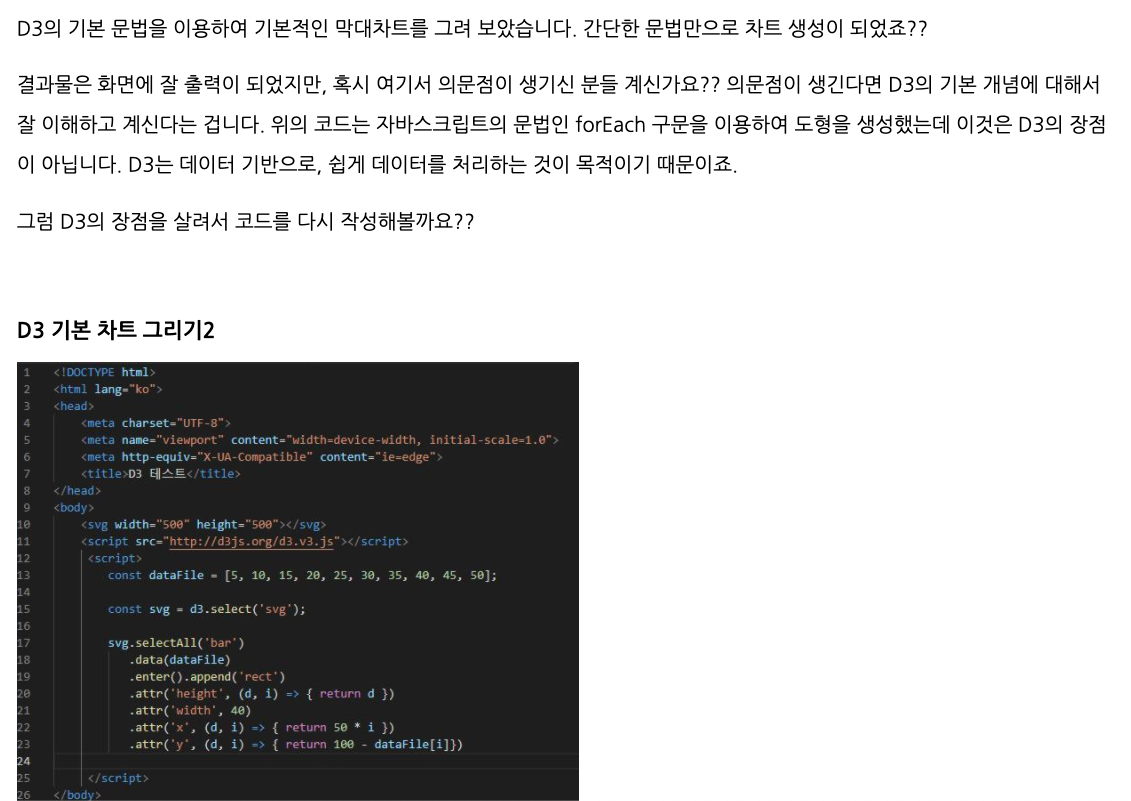


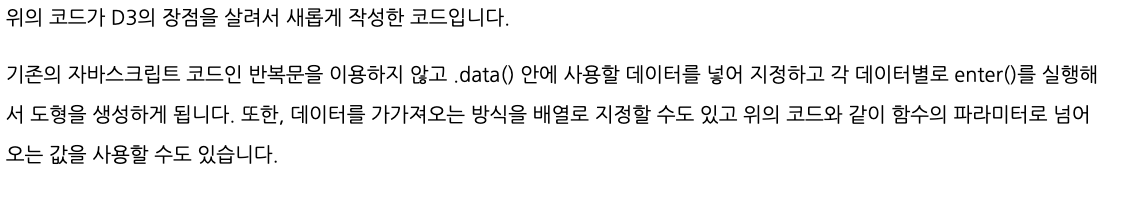


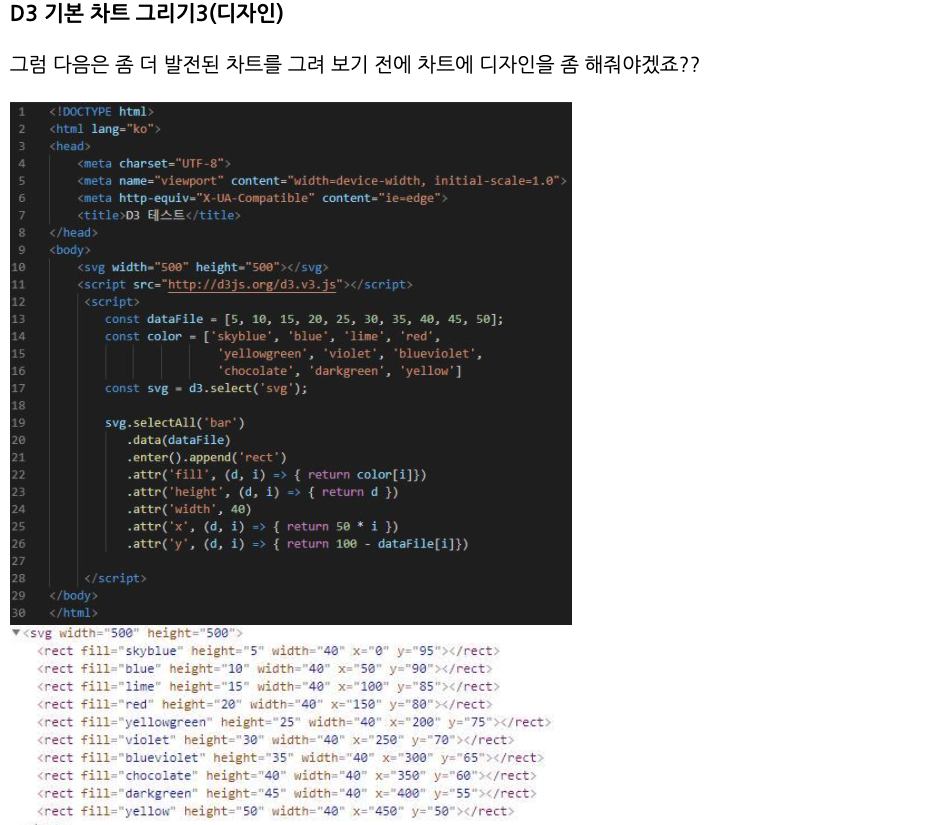












(전체 소스 코드)

<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)

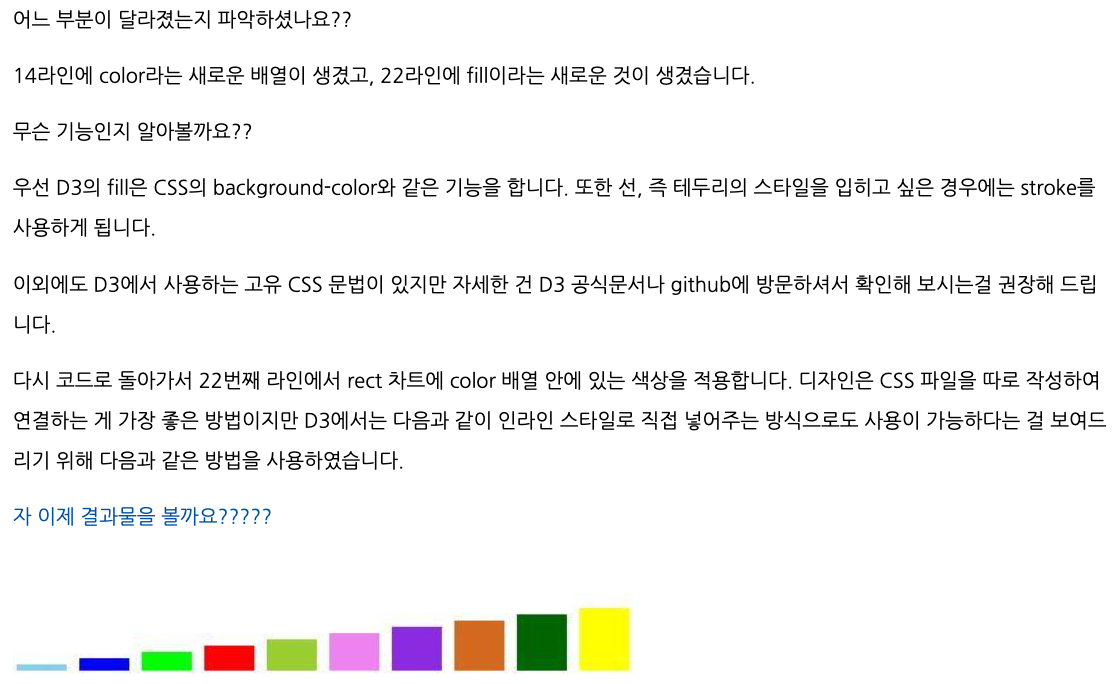
.attr('x', (d,i) => { return 50 \* i })

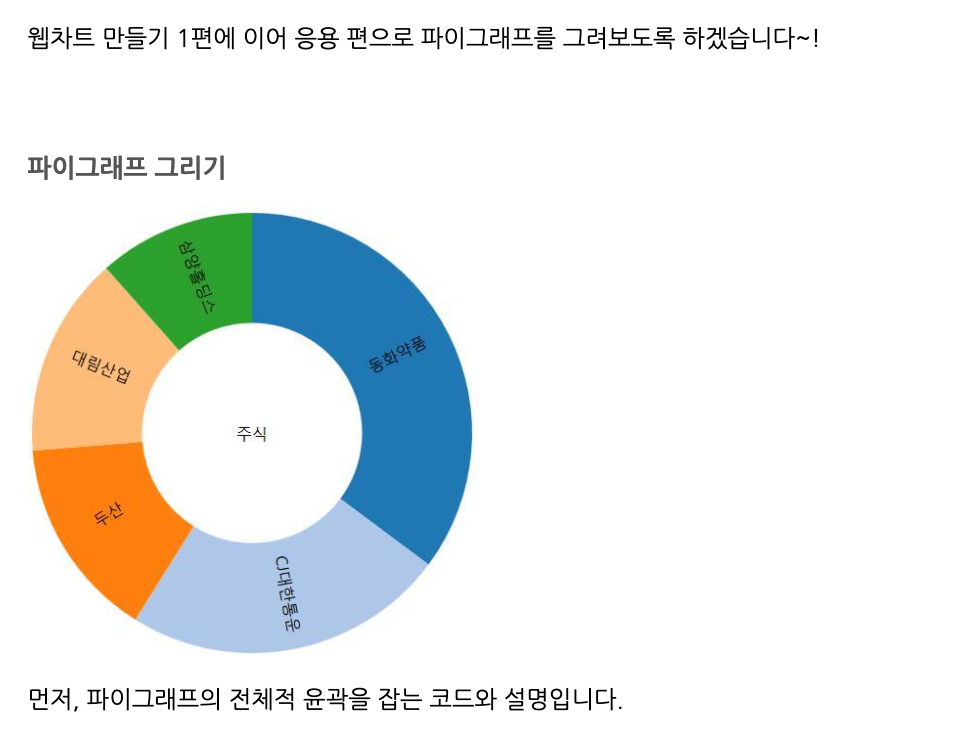
.attr('y', (d,i) => { return 100 - dataFile[i] })

</script>

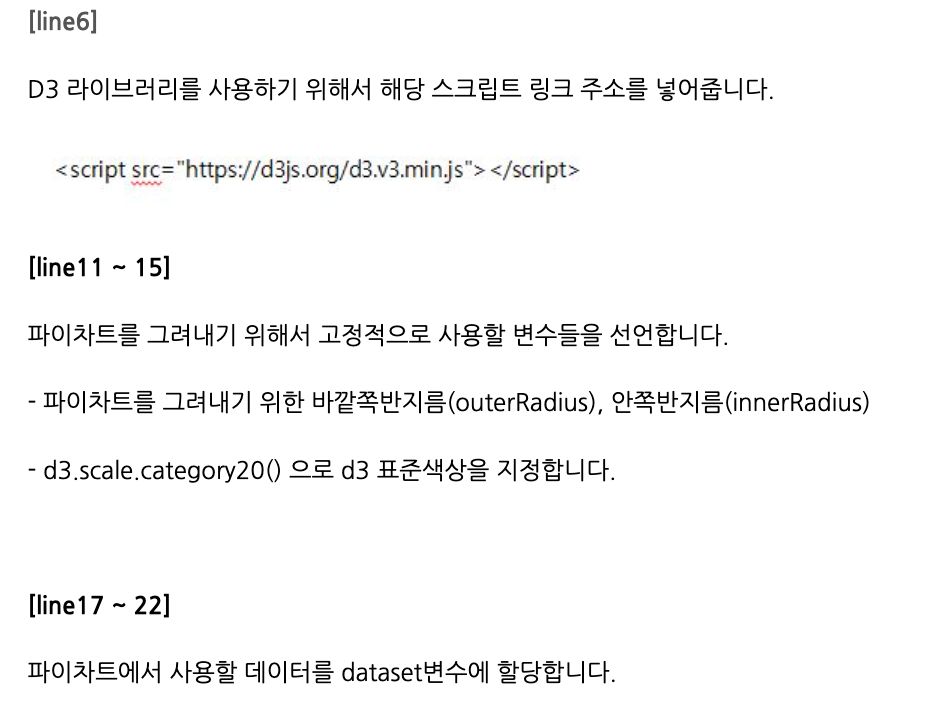
</body>

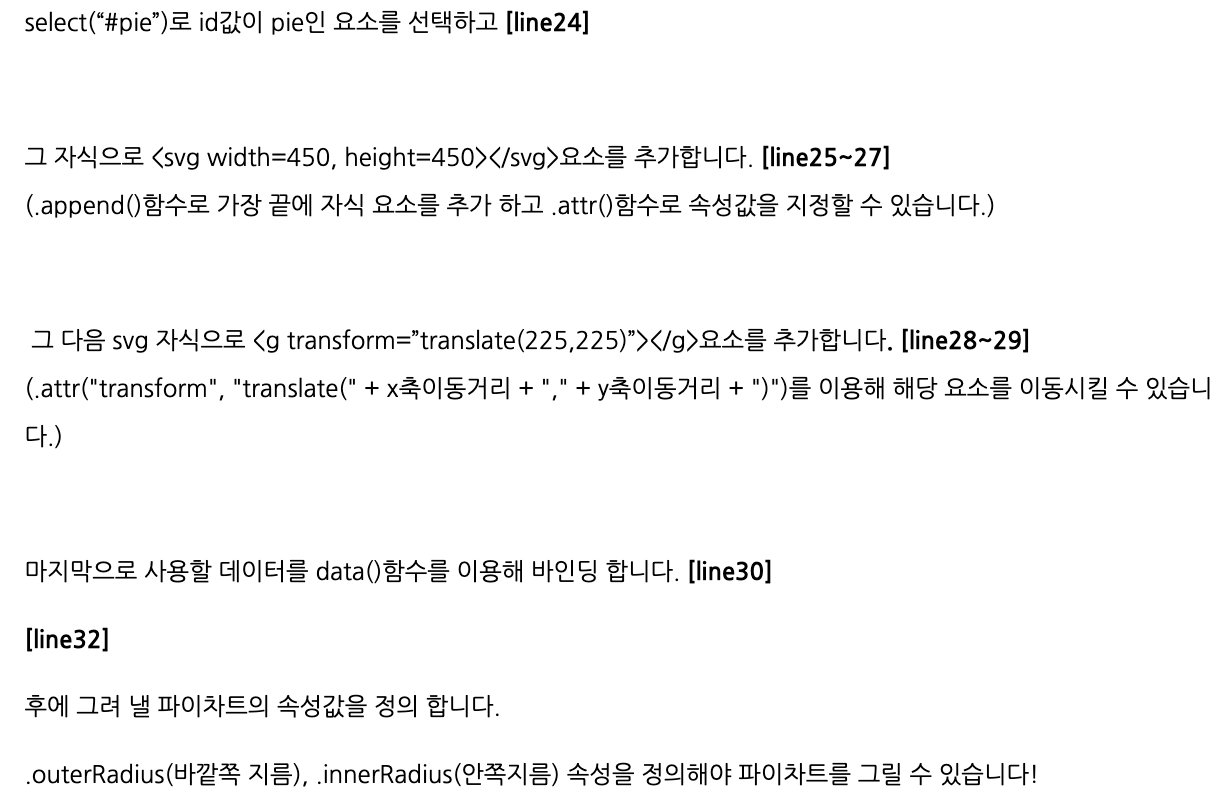
</html>



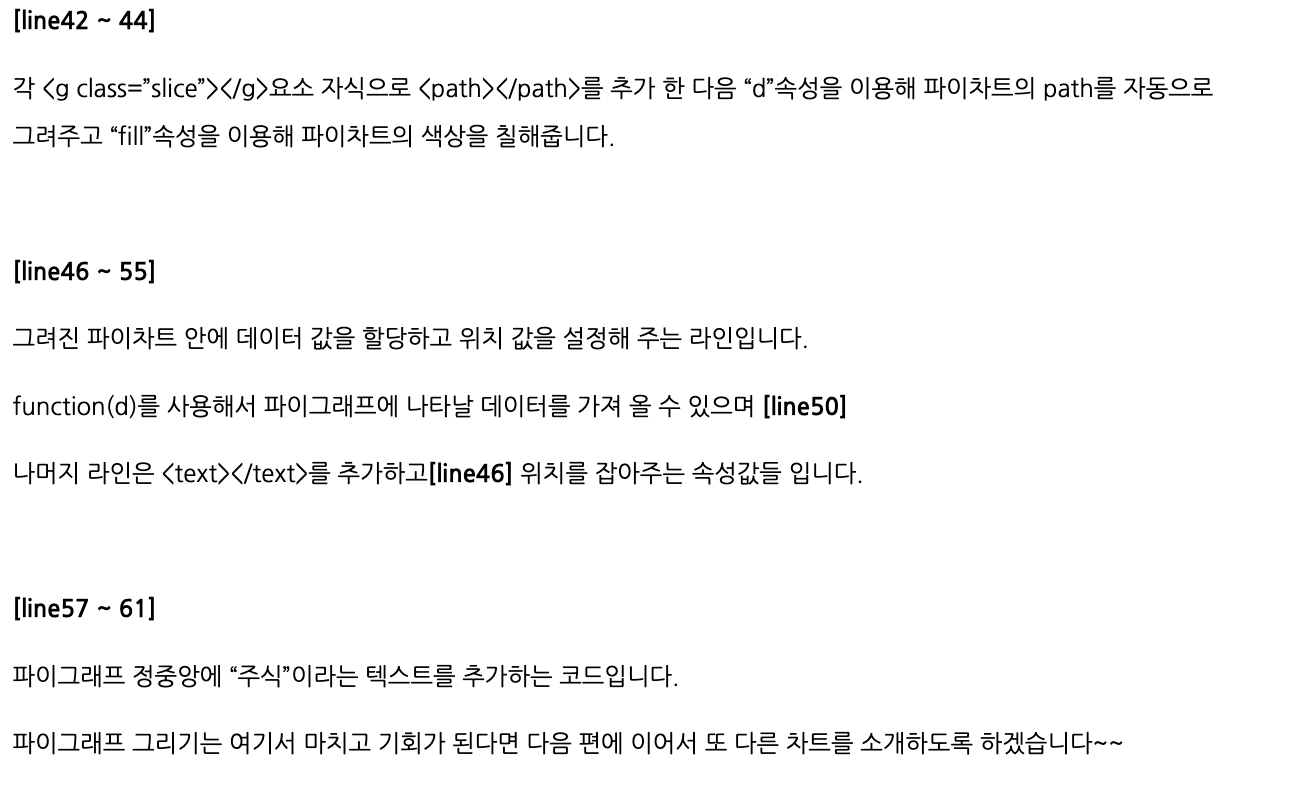












웹팩을 이용해서 그래픽스 라이브러리를 타입스크립트로 번들링하고,

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 JS파일을 모듈화하여 코드의 가독성을 높일 수 있습니다.

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

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

자동화 도구들을 통해 웹 개발을 편하게 수행할 수 있습니다.

또한 모듈화를 이용하여 Lazy Loading을 통해 필요한 자원을 나중에 요청할 수 있어 속도를 더욱 향상시킬 수 있습니다.

초기설정

우선 웹팩을 사용하기 위한 dependency를 설치합니다.

npm i webpack webpack-cli -D

타입스크립트를 이용하기 위해 관련된 dependency를 설치합니다.

npm install --save-dev typescript ts-loader

그 후, 해당 프로젝트에 사용할 tsconfig를 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 export하는 설정 객체를 읽어서 해당 설정대로 구성을 하기 때문에,

config 파일에서 본인이 작성한 설정을 export 해야 합니다.

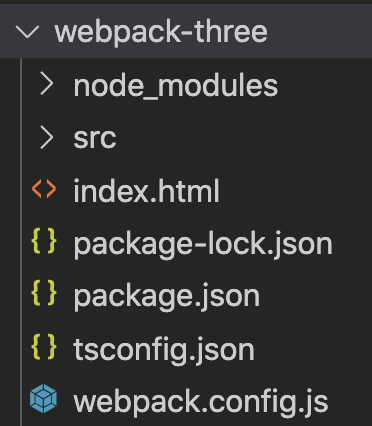
|  |  |
| --- | --- |
|  | var path = require("path"); |
|  |  |
|  | module.exports = { |
|  | entry: "./src/index.ts", |
|  | module: { |
|  | rules: [ |
|  | { |
|  | test: /\.tsx?$/, |
|  | use: "ts-loader", |
|  | exclude: /node\_modules/, |
|  | }, |
|  | ], |
|  | }, |
|  | resolve: { |
|  | extensions: [".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.html을 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.

프로젝트 현재 구조

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

 (이 소스가 src\index.ts에 추가되면 된다. 22/1/19일 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.json에 다음 명령어를 추가합니다.

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

(기존 소스 위에 위의 코드를 추가한다. 22/1/19일 체크)

(아래의 lodash모듈이 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.

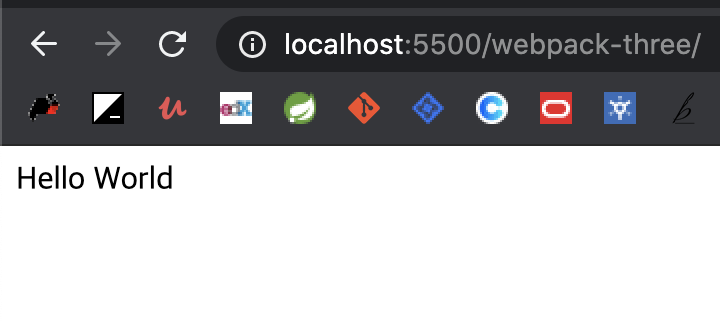
이를 수행하고 live-server를 켜서 index.html에 접근하면 다음과 같은 화면을 볼 수 있습니다.

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

npm install -g live-server

jongdeokkim@JONGui-MacBookPro DemoWebPack % live-server

Serving "/Users/jongdeokkim/Desktop/DemoWebPack" at http://127.0.0.1:8080



기존 index.html에서 번들링된 main.js 파일을 불러와 화면에 Hello World를 표시한 것입니다.

이와 같이 쉽게 타입스크립트 모듈을 자바스크립트로 번들링할 수 있습니다.

그럼 이제 구체적인 설정 방식에 대해서 알아봅시다.

웹팩의 4가지 주요 속성

**entry** - 자바스크립트의 진입점이라고 할 수 있습니다. html에서 js를 불러오면 그때 페이지가 렌더링되는데, 불러오는 개별 js 모듈의 entry를 지정하여 서로 다른 js 번들을 구성할 수 있습니다.

**output** - 웹팩을 실행한 결과물의 파일 경로를 의미하며, filename과 path를 정해줍니다. output filename엔 entry, 모듈 ID, 해시 등을 적용할 수 있습니다.

**loader** - 자바스크립트가 아닌 파일에 대한 변환을 도와줍니다. (HTML, CSS, Image, 폰트 등) 타입스크립트 파일도 ts-loader를 이용하여 js로 로드합니다. module의 rules 영역에 개별 객체로 지정해주며, test 항목에 파일의 확장자를, use 항목에 loader의 이름을 지정해줍니다.

loader를 적용할 때 한 파일에 여러 로더를 적용하게 되면, loader의 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, js에서 모듈을 불러올 때 (import ~ from ~) 이걸 정확히 어떤 위치에서 어떻게 로드할지를 지정을 해주는 것입니다.

alias를 두어 지정한 경로를 alias로 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 설정을 해주는 것입니다.

Provide plugin을 사용해서 특정 모듈을 전역으로 사용할 수 있게 만들 수 있는데, Jquery나 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, devServer에 대한 설정을 webpack.config.js 에 추가해줍니다.

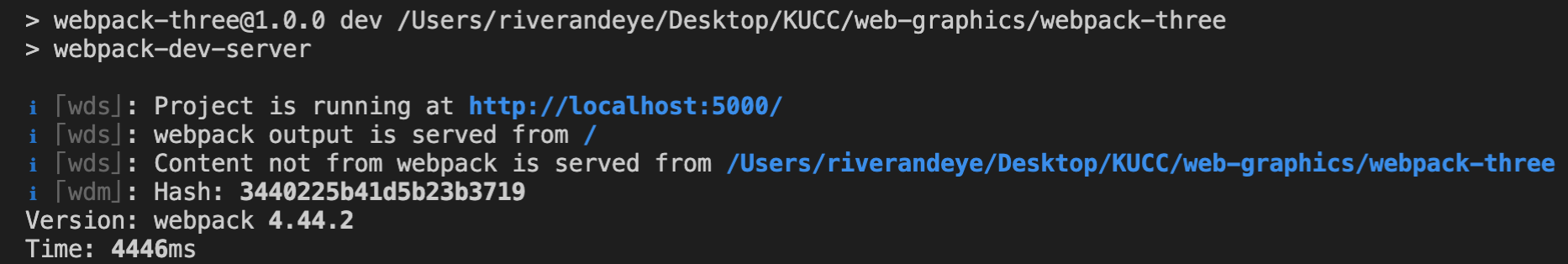
|  |  |
| --- | --- |
|  | module.exports = { |
|  | ... |
|  | devServer: { |
|  | port: 5000, |
|  | }, |
|  | }; |

여러 설정이 있겠지만, 여기서는 port만 지정하겠습니다.

그 후, package.json 의 script에 dev 명령어를 추가해줍니다.

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

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

웹팩 dev server 실행

그리고 나서, 웹팩에 지정한 source들을 수정하고 실행하게 되면, 파일이 변경되면 (저장되면) 빌드되고 그 결과물이 localhost:5000에 나타나는 것을 확인할 수 있습니다.

[2] Three.js - 시작하기

by Riverandeye 2020. 9. 22.

Three.js 는 자바스크립트 3d 라이브러리로 많은 곳에서 사용되고 있습니다.

다양한 라이브러리가 있지만, Three js를 사용하는건 아무래도 커뮤니티가 크고 TypeScript 지원이 잘 된다는 것이 그 이유입니다.

Three.js 엔 여러 장점이 있는데 우선 대부분의 브라우저에서 동작하고

3d 를 구현하기 위해 개별 플러그인이 필요하지 않습니다.

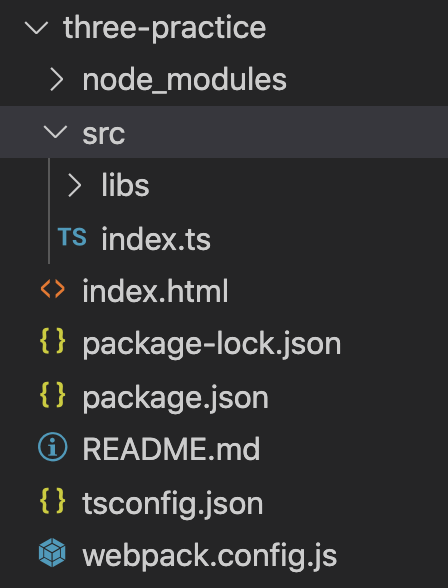
무엇보다 가장 큰 장점은 WebGL을 몰라도 쉽게 사용할 수 있다는 큰 장점이 있습니다.

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

저는 간단한 웹팩 설정으로 ts를 이용하여 로컬에서 작업하였습니다. 설정 방식은 [이 글](https://riverandeye.tistory.com/entry/1-Webpack-%EC%9D%84-%EC%9D%B4%EC%9A%A9%ED%95%9C-JavaScript-%EB%B2%88%EB%93%A4%EB%A7%81?category=899698)을 참고하시면 좋을 것 같습니다.

설정하기

위 웹팩 설정을 완료했다면, 프로젝트의 구조는 다음과 같을 것입니다.

libs 폴더는 무시하셔도 됩니다

우선 index.ts에 작업을 할 것이고, html은 다음과 같이 설정합니다.

|  |  |
| --- | --- |
|  | <html> |
|  | <head></head> |
|  | <style> |
|  | body { |
|  | margin: 0; |
|  | overflow: hidden; |
|  | } |
|  | </style> |
|  | <body> |
|  | <script src="main.js"></script> |
|  | </body> |
|  | </html> |

body에 적용된 css는 webpack의 css-loader를 직접 추가하셔서 개별 css 파일로 분리하셔도 좋습니다.

앞으로 작성하는 모든 코드는 index.ts에 추가할 것입니다.

따라하기

우선 간단하게 따라해봅시다.

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

npm install three

1. main 함수를 추가하고, 해당 main함수를 window.onload에 할당합니다.

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

2. html의 body에 output의 id를 가진 간단한 div를 추가합니다

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

3. three.js 를 import하고, scene과 renderer를 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. renderer에 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 default로 false가 설정되어 있기 때문에 true로 변경해주어야 합니다.

5. plane, sphere과 cube를 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 Geometry와 Material을 따로 구성한 후 하나의 Mesh로 합치는 점에서, 이전에 학습했던 [WebGL Fundamental](https://riverandeye.tistory.com/entry/1-WebGL-Fundamental?category=964359)의 Vector Shader와 Fragment Shader가 분리되어 있는 것이 떠오릅니다.

6. plane, sphere, cube의 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, cube의 위치를 지정해줍니다.

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

8. camera를 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 scene에 추가해줍니다.

|  |  |
| --- | --- |
|  | scene.add(plane); |
|  | scene.add(cube); |
|  | scene.add(sphere); |
|  | scene.add(spotLight); |

camera는 render할 때 camera를 추가해줍니다. (렌더링되는 영역 - 시야와 세계는 분리되어 구성됩니다)

11. Renderer의 domelement를 output div의 하위 컴포넌트로 넣어줍니다. (꼭 이렇게 안해도 됨)

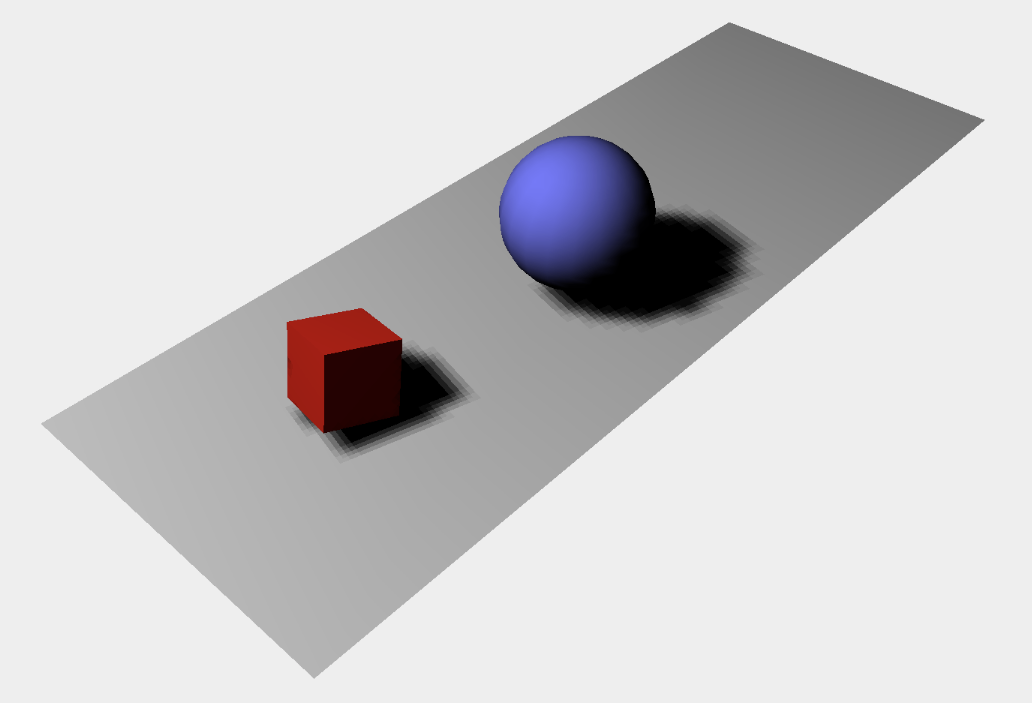
document.getElementById("output").appendChild(renderer.domElement);

이렇게 하면, div id="output"의 하위 dom 객체로 renderer가 렌더링하는 canvas가 나타나게 됩니다.

12. 마지막으로, renderer로 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. Scene을 정의하였습니다.

2. Renderer를 정의하였고, 그림자를 설정해주었습니다.

3. 각 Object의 Geometry, Material을 정의하고 position을 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 Scene을 구성하는데 사용되는 Object와 역할

three.Scene Object의 역할

geometric과 mesh와의 관계

orthographic camera와 perspective camera의 차이 를 알아보겠습니다.

Scene api

화면에 무언가를 보여주려면 다음 3가지 요소가 필요합니다.

- Camera -> 화면에 어떤 요소가 렌더링되는지를 결정

- Lights -> 물체가 어떻게 보이고, 그림자를 어떻게 생성하는지를 결정

- Objects -> Camera에 비춰지는 물체들

Scene 객체는 이런 서로 다른 요소들을 담는 역할을 합니다.

Scene에 추가된 Object를 관리하는 메소드들은 많은데, [이전 예시](https://riverandeye.tistory.com/entry/2-Threejs-%EC%8B%9C%EC%9E%91%ED%95%98%EA%B8%B0)에서 Scene에 Object를 추가하는 scene.add 메소드도 이에 포함됩니다.

대표적으로 다음이 있습니다.

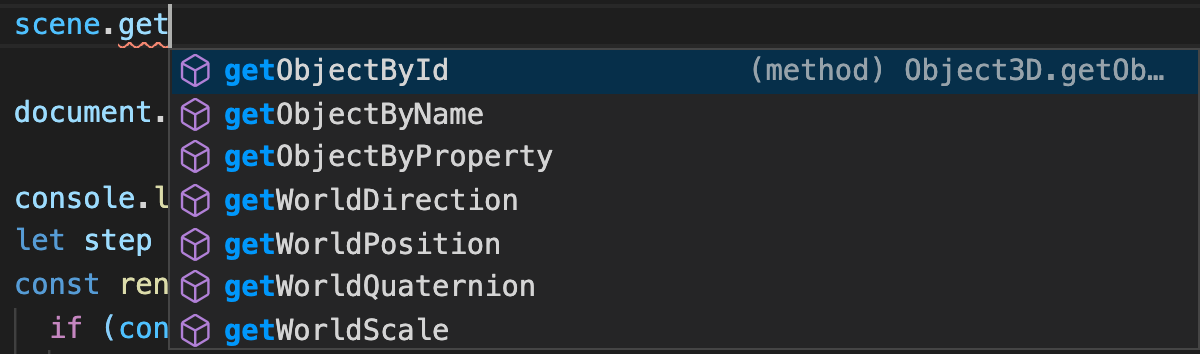
Scene.add

Scene.remove

Scene.children

Scene.getObjectBy...

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

이런식으로..

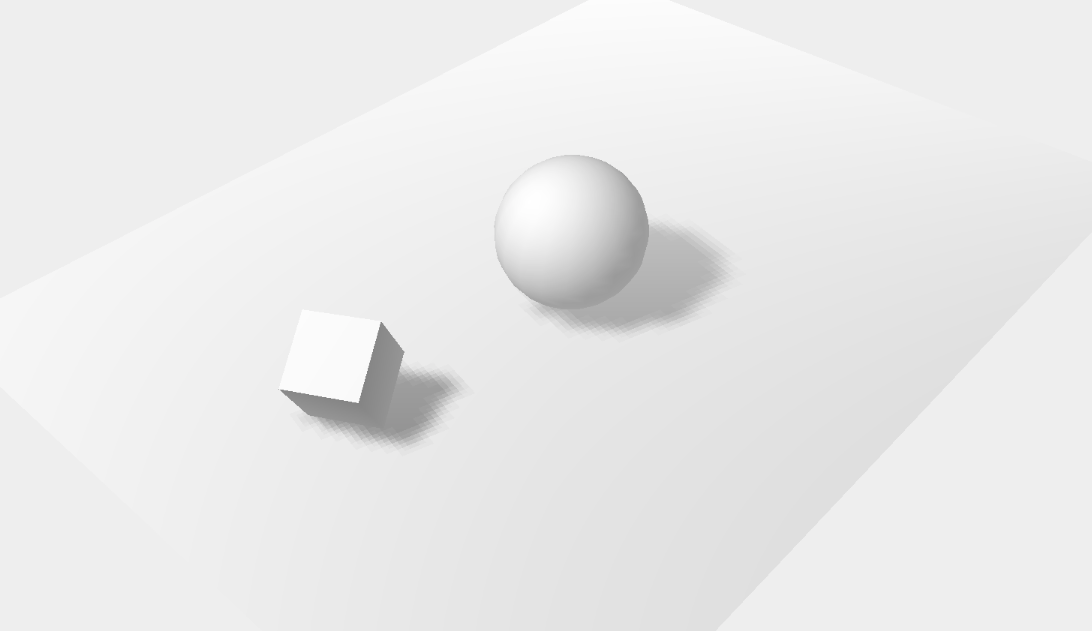
타입으로 제공되는 api를 적극 활용하고, 공식문서를 통해 학습하는 것이 좋은 방법일 수 있습니다.

Scene.traverse 는 함수를 입력받아 모든 Child element에 적용합니다.

Scene.fog를 통해 안개 효과를 추가할 수 있습니다.

Scene.overrideMaterial 을 이용하여 각 Child Object에 적용된 Material을 Override할 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

geometry는 3d space의 point의 집합을 의미하며, 이런 점들을 연결하는 삼각형의 면(mesh)들을 함께 의미합니다.

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

Three.js를 이용하면 일일이 vertice와 face를 정의할 필요 없이 해당 object를 정의하는 key feature만 정의하면 됩니다.

예를 들어 cube는 width height depth 겠지요.

임의로 vector와 face를 정의한 후 이를 이용해서 Geometry를 구성할 수 있습니다.

임의의 Geometry를 구성한 후 computeFaceNormals 를 호출하여 각 face의 normal vector를 계산하여야

광원에 대해 비치는 빛을 구성할 수 있습니다.

translation

rotation

이런것들은 필요할때마다 따로 적용해보면서 공부하면 됩니다.

Orthographic camera vs Perspective camera

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

Orthographic Camera는 동일한 물체는 항상 동일한 크기로 보여진다. 카에라와 물체의 거리와는 상관 없이 말이다.

심시티나 롤러코스터타이쿤 이런 게임에 Orthographic Camera가 적용된다.

**Perspective Camera**엔 다음 속성을 적용할 수 있다.

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

aspect - horizontal 과 vertical size의 비율을 정한다. 일반적으로 (window.innerWidth / window.innerHeight) 를 사용한다.

near - 얼마나 가까이 있는 물체까지 렌더링해야 하는지를 정한다

far - 얼마나 멀리 있는 물체까지 렌더링해야 하는지를 결정한다.

zoom - 말그대로 zoom이다.

**Orthographic Camera** 는 aspect ration나 fov에 관한 것이 아니라 모든 object가 동일한 크기로 렌더링되기 때문에

렌더링되야 하는 큐브 영역만 지정해주면 된다.

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

near과 far를 통해 박스의 width 를 지정해준다.

zoom은 크기를 확대 혹은 축소한다.

Light Sources

WebGL 자체에 lighting을 위한 Support는 없고, 직접 구현해야 한다. (근데 엄청 힘듬.. [참고](https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Lighting_in_WebGL))

**Lights의 종류**

- AmbientLight : 기본적인 빛, global하게 빛이 적용됨. 방향이란 개념이 없음.

- PointLight : Single Point로부터 발산되는 빛 (Shadow를 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 Light에 Lensflare 효과를 지정할 수 있다.

[4] Three.js - Material

by Riverandeye 2020. 9. 28.

Material은 Object의 피부같은 것으로, Geometry가 투명한지, Metallic한지 "피부"같은 것을 결정한다.

다양한 Material이 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 Face에 대해 독립적인 material을 지정할 수 있음

MeshLambertMaterial -> Vertice에서 lighting을 계산하여 Shiny 하지 않음

MeshPhongMaterial -> 모든 픽셀에 대해 lighting을 계싼하여 Shiny하게 보인다.

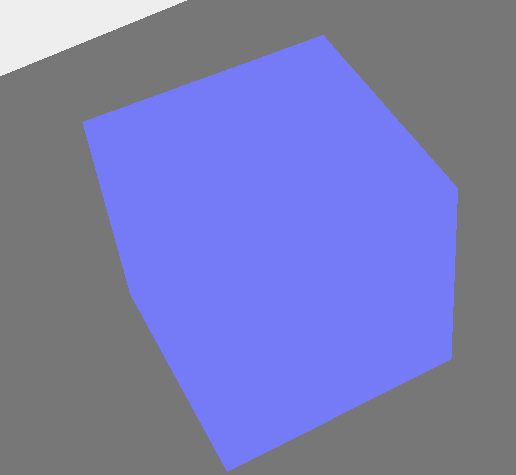
ShaderMaterial -> 직접 Shader Program을 작성하여 어떻게 Vertice가 위치하고 픽셀이 색칠되는지를 결정한다.

LineBasicMaterial -> Line Geometry에 사용되는 Material

LineDashMaterial -> Line에 Dash 추가해줌.

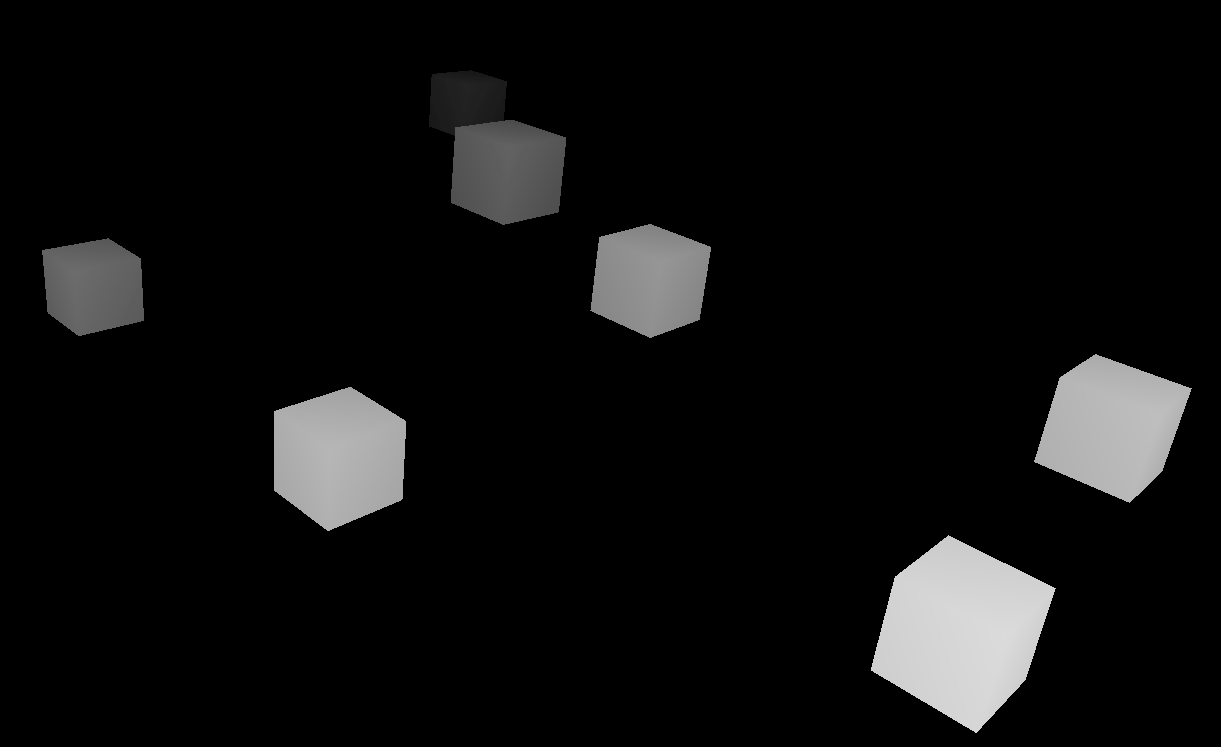
각각을 직접 보면서 어떤 질감을 띄는지 이해해보자.

**MeshMaterial**

BasicMeshMaterial

카메라나 빛의 위치에 대해 표면의 색상 변화가 없다. 이게 기본형.

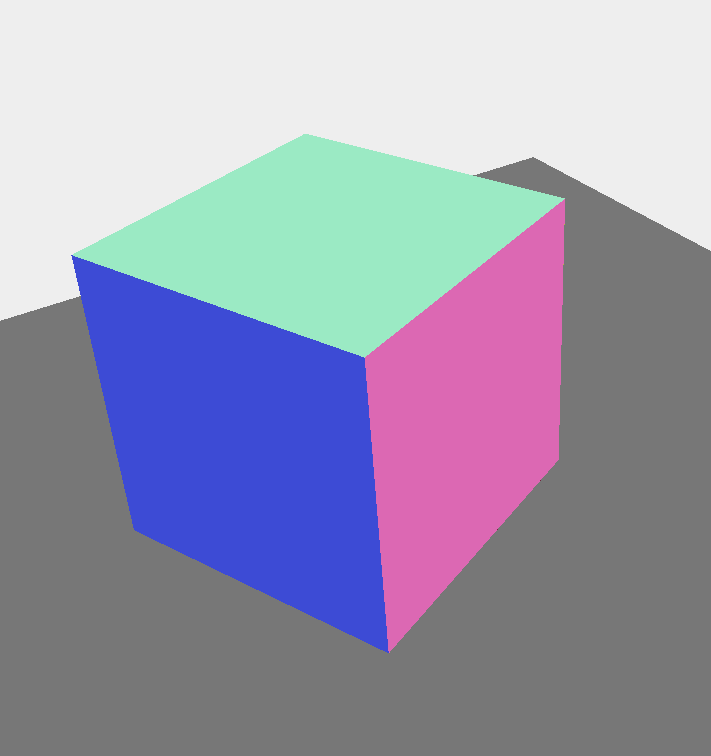
**DepthMaterial**

카메라와의 거리에 따라 빛의 크기가 달라진다.

AmbientLight를 준 상태에서 DepthMaterial은 디음과 같이 카메라와의 거리에 따라 색상이 달라보인다.

멀수록 잘 안보이는 그런 느낌을 준다.

**MeshNormalMaterial**

표면이 가리키는 방향에 따라 색상이 달라진다.

x축을 R, y축을 G, z축을 B로 매핑해서, 현재 표면의 법선벡터가 가리키는 방향으로 색상을 표현한다.

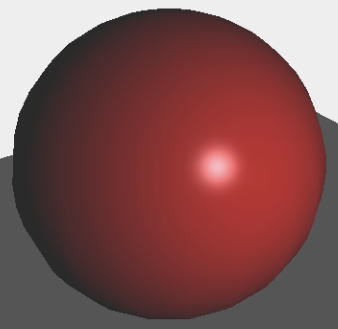
디버깅할때 사용하면 좋을듯..? 어떤 용도로 쓸지는 아직 감이 안온다.

**MeshLambertMaterial**

다소 밋밋한 입체

빛에 따라 색상이 변하지만, 다소 밋밋하고 광이 나지 않는다.

**MeshPhongMaterial**

광이 난다

MeshLambertMaterial보다 다소 광이 나는 것을 확인할 수 있다.

Material과 Geometry는 아무래도 직접 Blender로 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>