과목명	컴퓨터그래픽스
담당교수	송인식 교수님
학과	소프트웨어학과
학번	32153180
이름	이상민
제출일자	2020.05.30

1. 프로젝트 개요

(1) 배경 설명

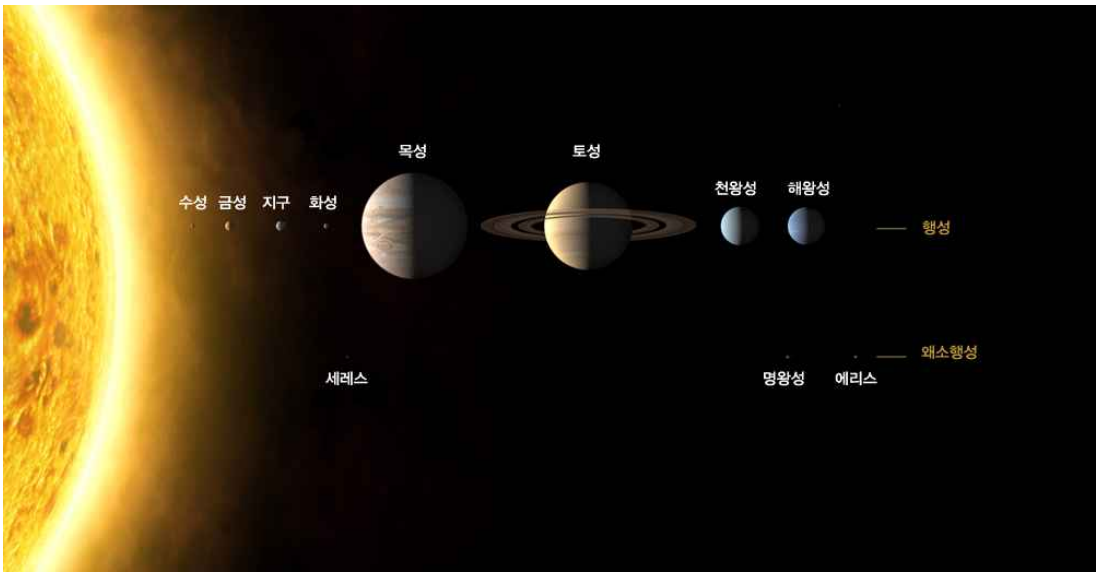
프로젝트의 목표는 태양계를 화면에 띄우는 것입니다. 작은 행성들까지 모두 구현하기에는 시간이 많이 소요될 것 같아 대표적인 내행성계 (수성, 금성, 지구, 화성), 외행성계 (목성, 토성, 천왕성, 해왕성) 그리고 태양만을 모델링하여 표현하겠습니다. 추가로 이들의 자전과 공전까지 시각적으로 확인할 수 있게끔 하는 것이 목표입니다.

(2) 문제 정의

사용자는 태양계의 위에서 언급한 행성들을 둘러볼 수 있습니다. 전체적인 배경도 일반적인 2D 우주 사진을 사용하지 않고 skybox를 사용하여 조금 더 입체적이고 현실적인 장면을 연출합니다.

(3) 기존의 처리 방법

인터넷에 태양계를 검색했을 때 나오는 이미지는 아래와 같습니다.



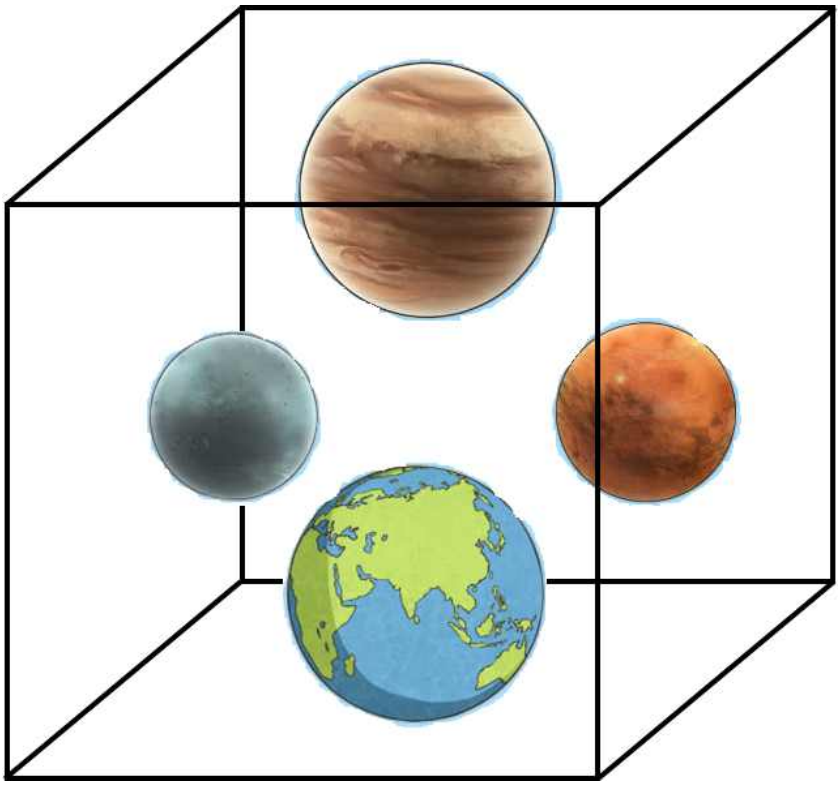
- 출처 : https://astro.kasi.re.kr/learning/pageView/5111

위의 사진은 태양계를 한눈에 들여다볼 수는 있지만, 평면상의 사진이기 때문에 입체적이지 않습니다. 그러므로 이러한 사진을 접하는 것은 현실적이지 않다고 볼 수 있습니다.

찾아보면 아래의 사진처럼 조금 더 입체적인 사진을 발견할 수 있습니다. 하지만 이 또한 사실감이 느껴지지는 않습니다. 저는 이것이 2D의 한계점이라고 생각합니다.

- 출처 :
https://wouldyoulike.org/featured/%ED%83%9C%EC%96%91%EA%B3%84%EC%9D%98-%EC%88%A8%EC%9D%80-%EC%8B%9D%EA%B5%AC/

(4) 해결하고자 하는 방법



위의 이미지처럼 정육면체 모양의 배경 속에 여러 행성을 추가할 것입니다. 그리고 각각의 행성이 자전과 공전을 하도록 구현하여 시각적인 효과를 더해 줄 계획입니다. 그로 인해 더 사실적이고 입체감 있는 이미지를 확인할 수 있습니다.

2. 프로젝트 수행

(1) 사용자 인터페이스

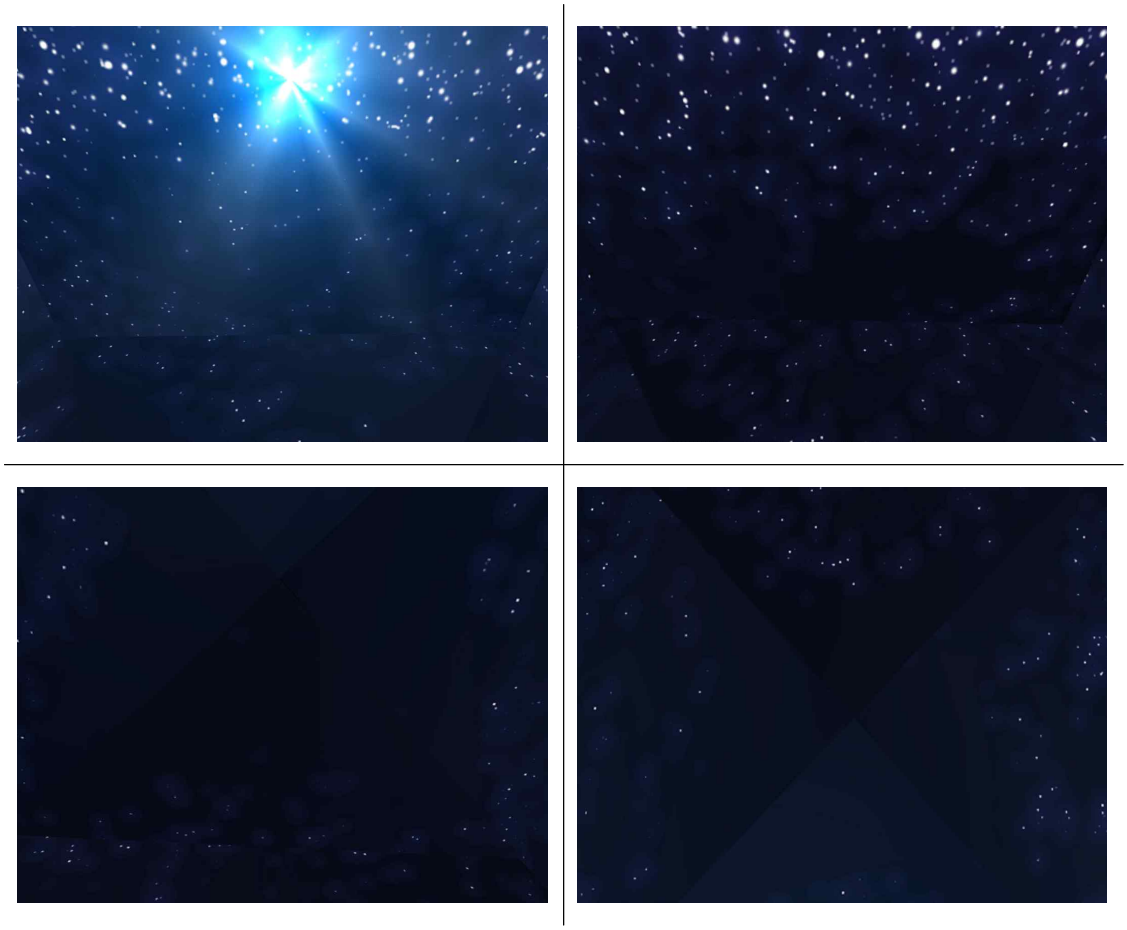
카메라 시점은 위에 있는 정육면체 내부 중앙에 위치합니다. 카메라와 마우스의 상호 작용을 위해 three.js에 있는 OrbitControls 함수를 이용합니다. 그렇게 되면 사용자는 원하는 위치에서 태양계를 관찰할 수 있습니다. 또한 마우스 휠을 이용하여 확대와 축소도 가능하게 합니다.

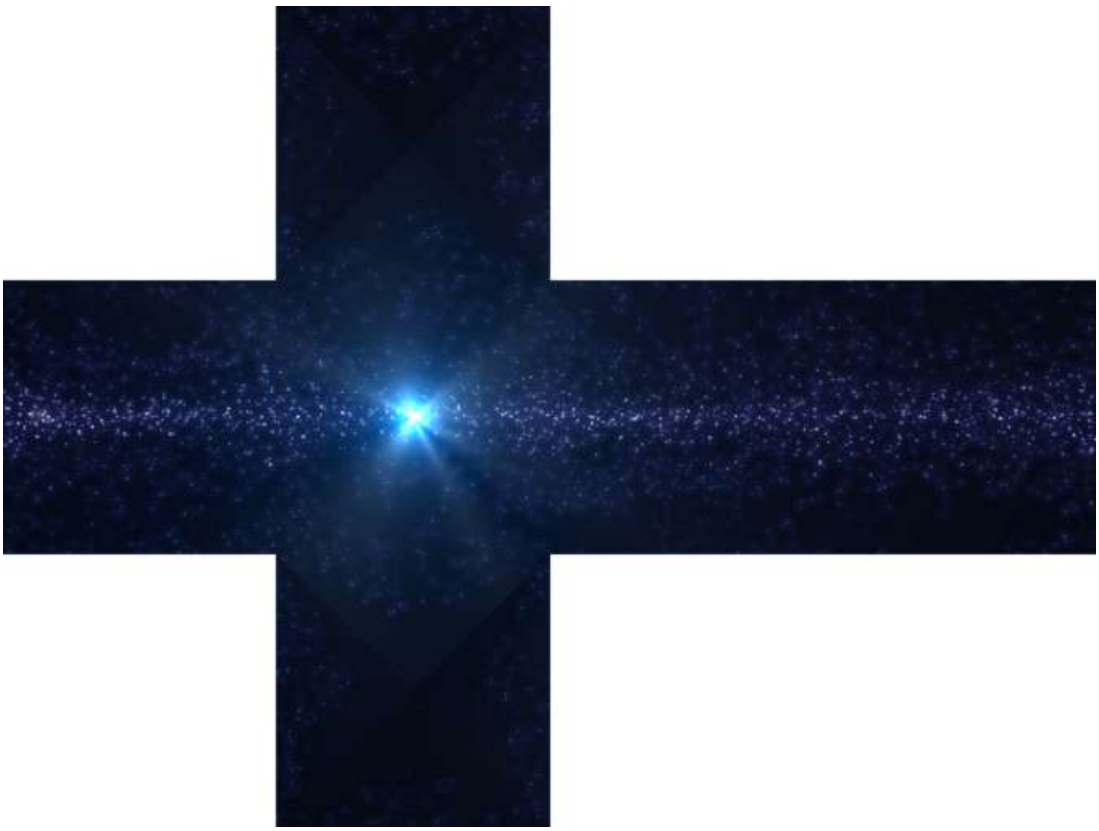
(2) 사용자 시나리오

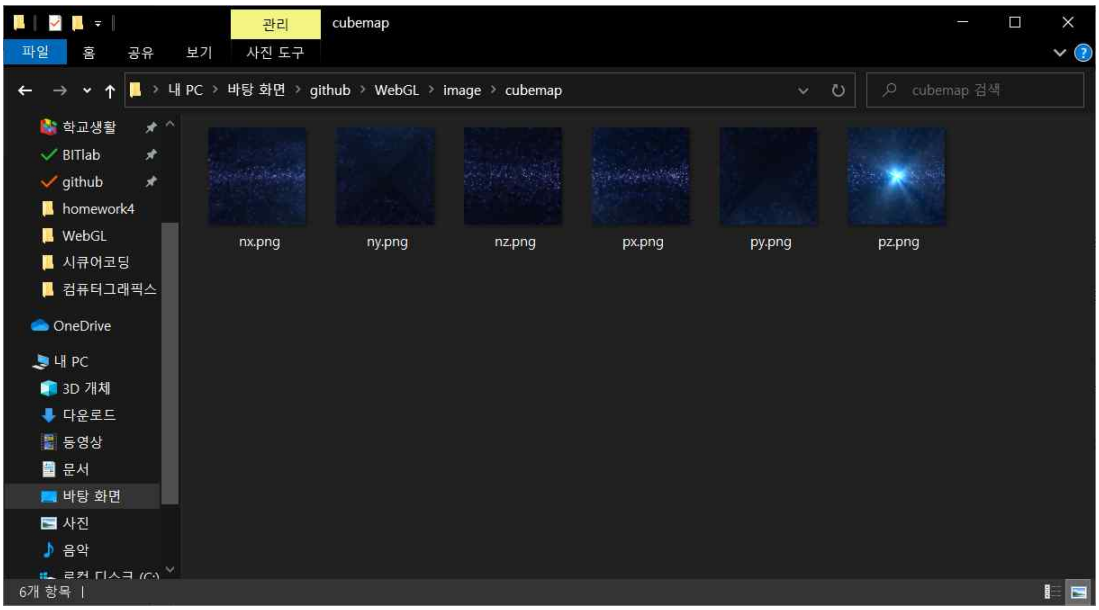
사용자는 시점 이동을 통해 태양계를 넓은 시야로 확인할 수 있습니다. 그리고 3D 모델의 특징상 더욱 생동감 있는 연출이 가능합니다.

(3) 구현 과정

처음에 구상한 3차원의 우주 공간은 아래 사진과 같은 skybox로 구현하여 입체감을 부각했습니다. skybox는 cubemap이라고도 불리며 정육면체로 이루어집니다.

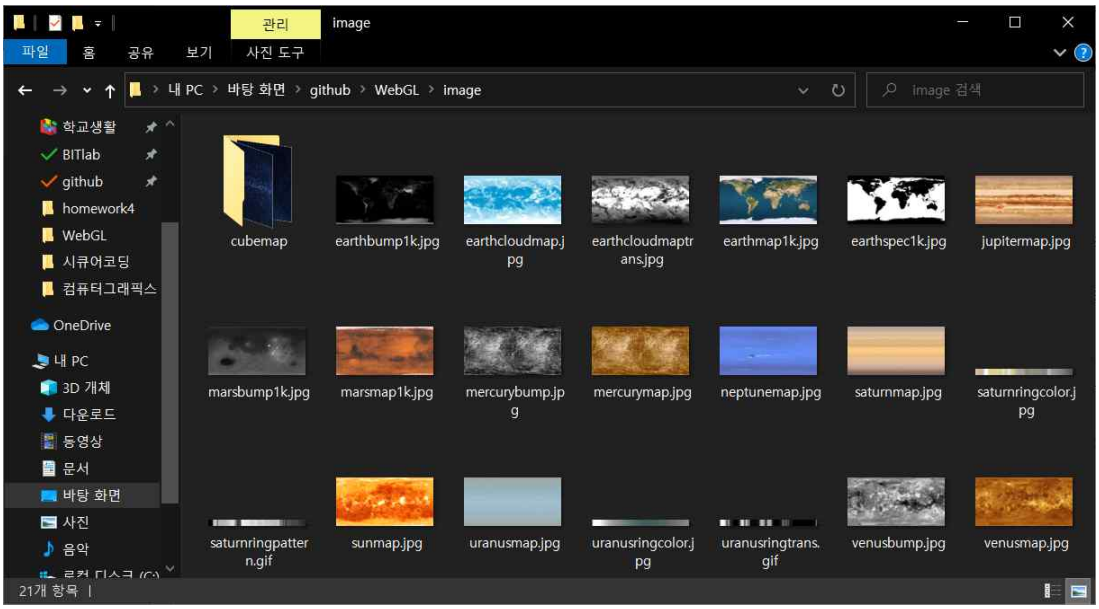






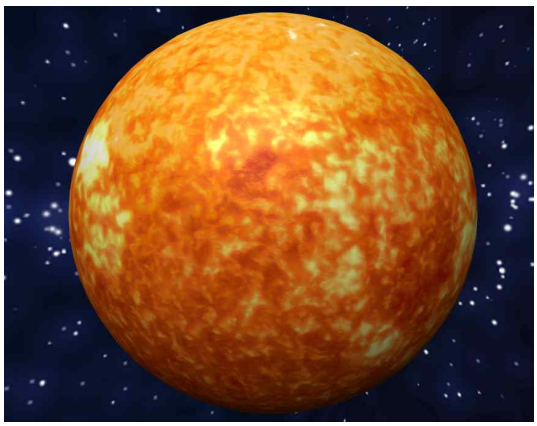
출처 : https://imgur.com/gallery/5cbZh

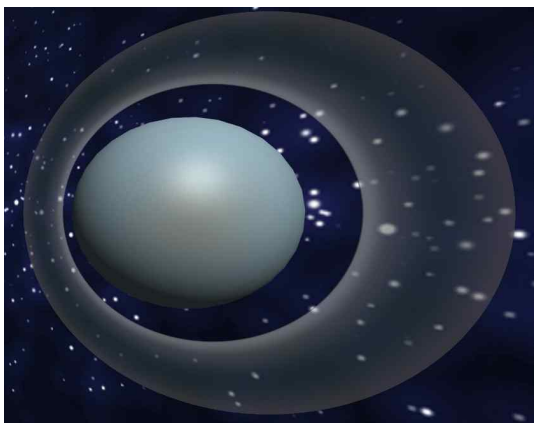
각 행성들은 three.js를 이용하여 제작했습니다. 행성들은 모두 구형이기 때문에 SphereGeometry를 통해 모델링 했고, 행성에 맞는 텍스쳐를 구해 매핑했습니다. 몇몇 행성은 질감 효과를 주기 위해 범프 매핑을 적용했습니다.



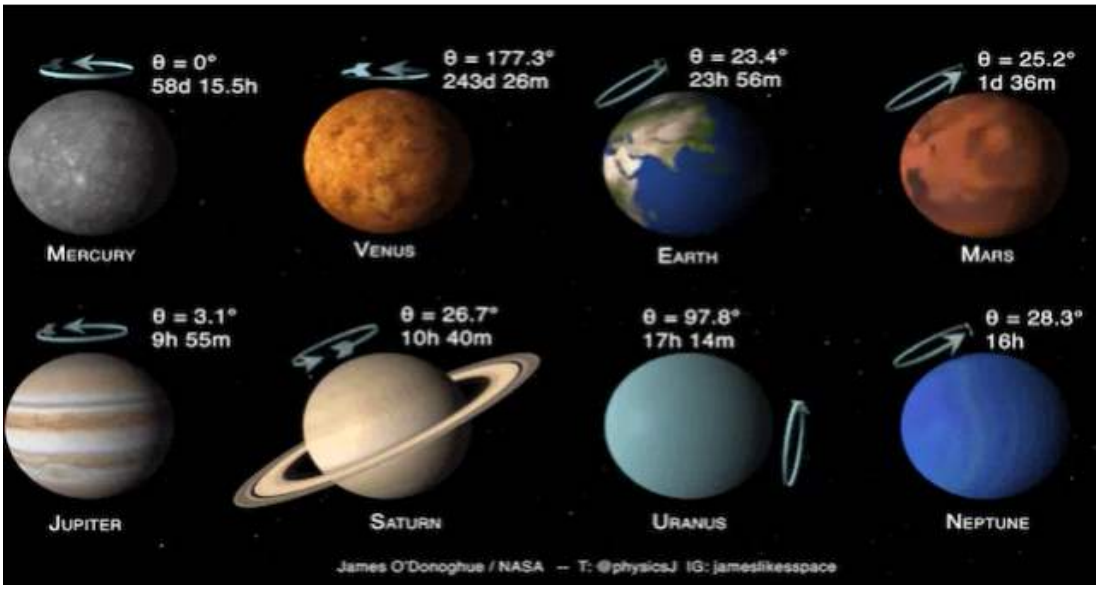
출처 : http://planetpixelemporium.com/sun.html

이렇게 모델링한 행성들은 아래와 같습니다.

태양	수성
	

금성	지구
	
화성	목성
	
토성	천왕성
	

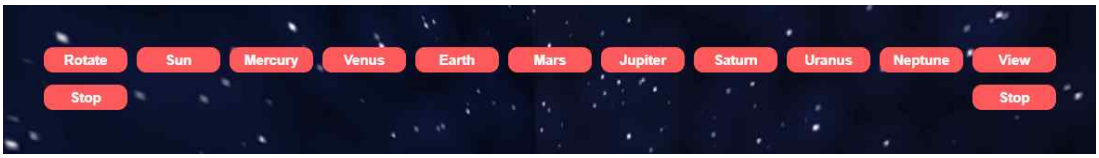
최대한 현실성을 높이기 위해 행성의 기울기, 자전 속도 및 공전 속도를 실제 값과 비례하게 제작했습니다.



태양계 행성의 평균공전속도

- 수성 : 47.36 km/s
- 금성 : 35.020 km/s
- 지구 : 29.783 km/s
- 화성 : 24.077 km/s
- 목성 : 13.05624 km/s
- 토성 : 9.639 km/s
- 천왕성 : 6.795 km/s
- 해왕성 : 5.43 km/s

출처 : https://www.fmkorea.com/best/1867442946
https://blog.naver.com/PostView.nhn?blogId=lsmnmh25&logNo=220966704962&proxyReferer=https:%2F%2Fwww.google.com%2F



화면 상단부에 여러 버튼을 두었고, 각 버튼의 기능은 아래와 같습니다.

Rotate	: 행성들이 태양을 중심으로 공전
Stop	: 공전 중지
Sun ~ Neptune	: 각 행성을 가까이에서 볼 수 있도록 시점 이동
View	: 전체적인 장면을 볼 수 있도록 회전
Stop	: 회전 중지

(4) 코드 설명

본 프로젝트는 three.js 파일을 제외하고 총 7개의 파일로 구성했습니다.

galaxy.html	기본 HTML 파일로 여러 버튼 포함
style.css	기본 CSS 파일
main.js	장면과 카메라 등 생성
planet.js	모든 행성 생성 및 텍스쳐 매핑
orbit.js	행성의 공전 궤도 그리는 함수 작성
rotate.js	자전 및 공전 함수 작성
viewpoint.js	카메라 시점 변경 함수 작성

```
<button id="revolution" onclick="revolution()">Rotate</button>
<button id="stop" onclick="stop()">Stop</button>

<button id="sun" onclick="sun()">Sun</button>
<button id="mercury" onclick="mercury()">Mercury</button>
<button id="venus" onclick="venus()">Venus</button>
<button id="earth" onclick="earth()">Earth</button>
<button id="mars" onclick="mars()">Mars</button>
<button id="jupiter" onclick="jupiter()">Jupiter</button>
<button id="saturn" onclick="saturn()">Saturn</button>
<button id="uranus" onclick="uranus()">Uranus</button>
<button id="neptune" onclick="neptune()">Neptune</button>

<button id="view" onclick="view()">View</button>
<button id="stop2" onclick="stop2()">Stop</button>
```

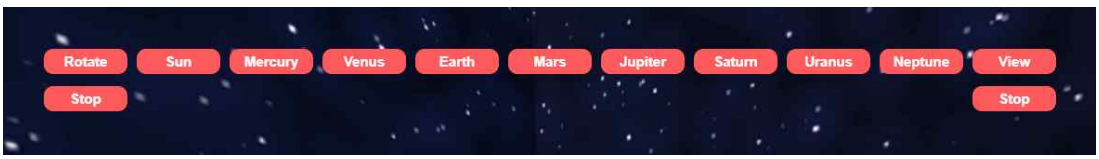
앞장에서 서술한 버튼들을 생성했고 각각의 버튼을 클릭했을 때, onclick 속성을 이용해 함수가 실행되도록 했습니다.

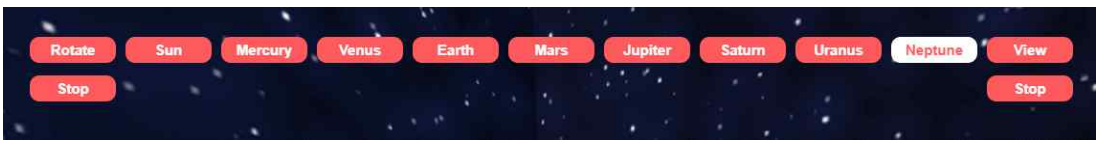
```
<script src="three.js-master/three.js"></script>
<script src="three.js-master/OrbitControls.js"></script>

<script src="planet.js"></script>
<script src="main.js"></script>
<script src="viewpoint.js"></script>
<script src="orbit.js"></script>
<script src="rotate.js"></script>
```

style.css

```css
button {
    width: 90px;
    background-color: #f8585b;
    color: #ffffff;
    font-weight: bold;
    border: none;
    border-radius: 10px;
    text-align: center;
    text-decoration: none;
    font-size: 15px;
    padding: 5px;
    cursor: pointer;
}
button:hover {
    background-color: #ffffff;
    color: #f8585b;
}
```

css 파일을 이용해 버튼을 디자인했고, hover 속성을 통해 마우스가 버튼 위에 올라가면 아래와 같이 색상이 변경되는 효과를 주었습니다.



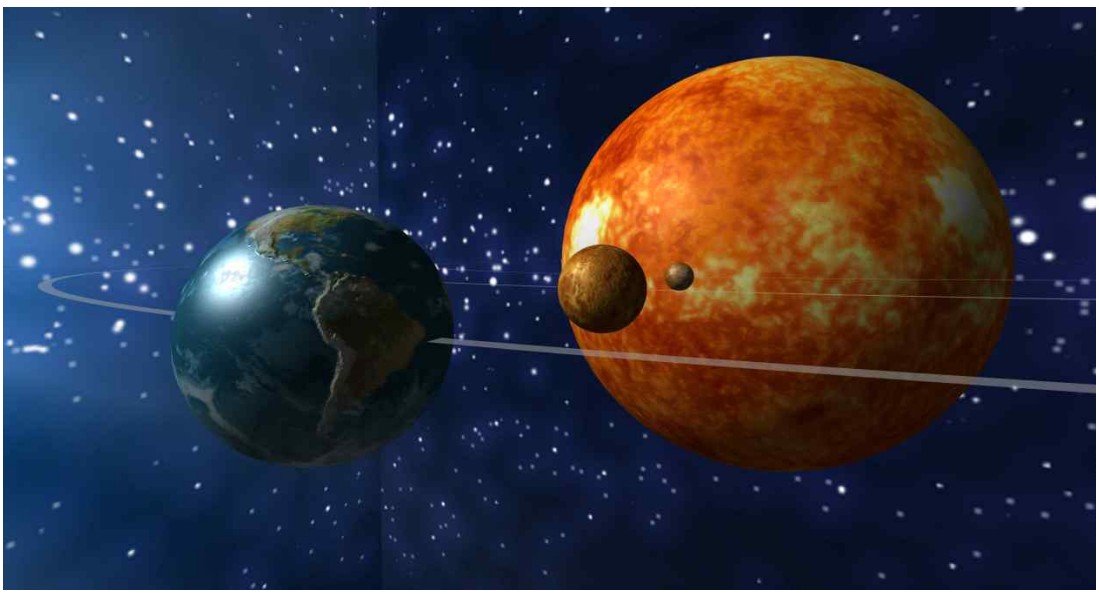


main.js

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1, 1000);
camera.position.set(100, 30, 180);

var renderer = new THREE.WebGLRenderer({ antialias: true });
renderer.setClearColor(0xffffff);
renderer.setPixelRatio(window.devicePixelRatio);
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
renderer.shadowMapEnabled = true;

var light = new THREE.AmbientLight(0x2e2e2e);
scene.add(light);
var light = new THREE.DirectionalLight(0xcccccc, 1);
light.position.set(5, 5, 5);
scene.add(light);
```

렌더링에 사용할 장면과 카메라를 선언해주고 WebGL 렌더러 오브젝트를 생성했습니다.
조명으로 AmbientLight와 DirectionLight를 추가했습니다.

AmbientLight : 모든 오브젝트에 전역으로 빛 방출
DirectionLight : 특정 방향으로 빛 방출



이로 인해 위의 사진처럼 어두운 부분과 밝은 부분이 다르게 나타납니다.

```
scene.background = new THREE.CubeTextureLoader()
.setPath('image/cubemap/')
.load(
    [
        'px.png',
        'nx.png',
        'py.png',
        'ny.png',
        'pz.png',
        'nz.png'
    ]
);
```

CubeTextureLoader 함수를 이용해 6개의 사진을 불러와 skybox를 구성하여 전체적인 배경을 구상했습니다.

```
var sunMesh = Planet.sun();
sunMesh.position.set(0, 0, 0);
scene.add(sunMesh);

var mercuryMesh = Planet.mercury();
mercuryMesh.position.set(50, 0, 0);
scene.add(mercuryMesh);

var venusMesh = Planet.venus();
venusMesh.position.set(65, 0, 0);
scene.add(venusMesh);

var earthMesh = Planet.earth();
earthMesh.position.set(80, 0, 0);
earthMesh.rotation.x = 0.2;
earthMesh.rotation.z = 0.3;
scene.add(earthMesh);
var earthCloud = Planet.earthCloud();
earthCloud.position.set(80, 0, 0);
earthCloud.rotation.x = 0.2;
earthCloud.rotation.z = 0.3;
scene.add(earthCloud);
```

모든 행성은 planet.js 파일에서 생성해주었고 Mesh를 반환값으로 받아 각 행성의 위치를 설정하여 장면에 추가했습니다.

```
var controls = new THREE.OrbitControls(camera);
controls.update();

var render = function() {
    requestAnimationFrame(render);
    controls.update();
    renderer.render(scene, camera);
};

render();
```

OrbitControls 함수를 이용해 마우스 드래그와 휠로 카메라 시점을 변경하게 하였고, 최종적으로 render 함수를 통해 장면을 볼 수 있습니다.

planet.js

```
Planet = {};

Planet.sun = function() {
    var geometry = new THREE.SphereGeometry(30, 32, 32);
    var material = new THREE.MeshPhongMaterial();
    material.map = THREE.ImageUtils.loadTexture("image/sunmap.jpg");
    material.bumpMap = THREE.ImageUtils.loadTexture("image/sunmap.jpg");
    material.bumpScale = 0.1;

    var mesh = new THREE.Mesh(geometry, material);
    return mesh;
}
```

위의 코드처럼 각 행성의 생성을 함수로 구현하여 Planet이라는 객체에 하나씩 넣었습니다.

Mesh(물체)는 쉽게 Geometry(뼈대)와 Material(표면)의 합으로 이루어진다고 볼 수 있습니다. 행성의 모양은 구이기 때문에 SphereGeometry를 이용해 뼈대를 만들었고, 표면을 MeshPhongMaterial로 만들었습니다. 이렇게 만들어진 Mesh를 반환하여 main.js 파일에서 사용합니다.

```
Planet.earth = function() {
    var geometry = new THREE.SphereGeometry(2.5, 32, 32);
    var material = new THREE.MeshPhongMaterial();
    material.map = THREE.ImageUtils.loadTexture("image/earthmap1k.jpg");
    material.bumpMap = THREE.ImageUtils.loadTexture("image/earthbump1k.jpg");
    material.bumpScale = 0.3;
    material.specularMap = THREE.ImageUtils.loadTexture("image/earthspec1k.jpg");
    material.specular = new THREE.Color("gray");

    var mesh = new THREE.Mesh(geometry, material);
    return mesh;
}

Planet.earthCloud = function() {
    var geometry = new THREE.SphereGeometry(2.5, 32, 32);
    var material = new THREE.MeshPhongMaterial();
    material.map = THREE.ImageUtils.loadTexture("image/earthcloudmap.jpg");
    material.side = THREE.DoubleSide;
    material.transparent = true;
    material.opacity = 0.4;

    var mesh = new THREE.Mesh(geometry, material);
    return mesh;
}
```

지구 같은 경우에는 earthCloud 함수를 제작해 텍스쳐 매핑으로 구름을 덧붙였습니다. opacity 옵션을 주어 투명도를 설정해주었습니다.

토성과 천왕성의 고리를 제작할 때 RingGeometry를 사용했는데 아래의 사진처럼 고리가 계속 이상하게 나왔습니다.

찾아본 결과 UV를 다시 계산해줘야 하는 문제였습니다. 해당 문제는 인터넷에 있는 코드로 해결할 수 있었습니다.

```
var material = new THREE.ShaderMaterial({
    uniforms: {
    texture: { value: texture },
    innerRadius: { value: 7 },
    outerRadius: { value: 11 },
    uOpacity: { value: 0.7 }
},
vertexShader: `
    varying vec3 vPos;

    void main() {
        vPos = position;
        vec3 viewPosition = (modelViewMatrix * vec4(position, 1.)).xyz;
        gl_Position = projectionMatrix * vec4(viewPosition, 1.);
    }
`,
fragmentShader: `
    uniform sampler2D texture;
    uniform float innerRadius;
    uniform float outerRadius;
    uniform float uOpacity;

    varying vec3 vPos;
```

다른 행성들과 다르게 ShaderMaterial로 표면을 만들어주는 코드입니다.

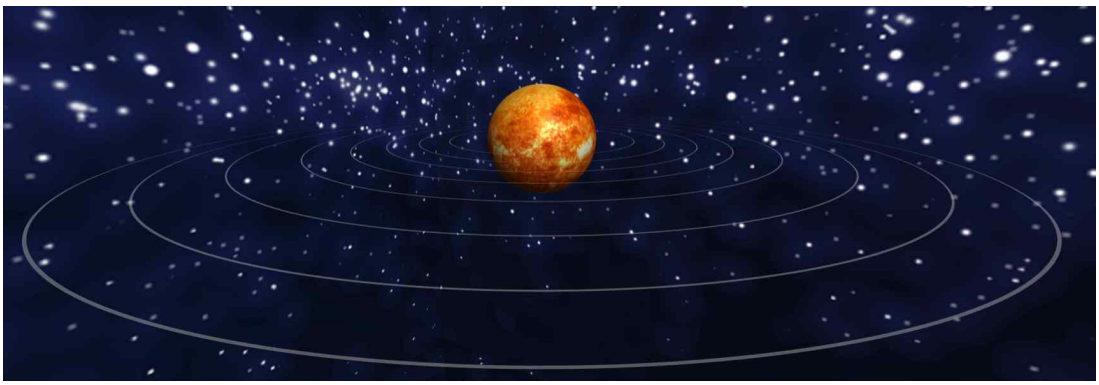
코드 출처

: https://codepen.io/prisoner849/pen/wvwVMEo?editors=0010

위의 코드 덕분에 원하는 모양의 고리를 만들 수 있었습니다. 해왕성도 마찬가지로 동일한 코드를 사용하여 고리를 만들었습니다.

```
// Mercury
var geometry = new THREE.RingGeometry(49.5, 50, 512);
var material    = new THREE.MeshPhongMaterial({
        side        : THREE.DoubleSide,
        transparent : true,
        opacity     : 0.5,
    });
var mesh = new THREE.Mesh(geometry, material);
scene.add(mesh);
mesh.lookAt(new THREE.Vector3(0, 1, 0));
```

RingGeometry를 이용하여 각 행성의 궤도를 생성했습니다.



태양으로부터의 거리를 각 행성의 위치에 맞추어 각기 다른 8개의 행성 궤도를 만들었습니다.

rotate.js

```
// Revolution (공전)
var mercuryOrbit, venusOrbit, earthOrbit, earthCloudOrbit, marsOrbit, jupiterOrbit,
    saturnOrbit, saturnRingOrbit, uranusOrbit, uranusRingOrbit, neptuneOrbit;
var orbits = [mercuryOrbit, venusOrbit, earthOrbit, earthCloudOrbit, marsOrbit, jupiterOrbit,
              saturnOrbit, saturnRingOrbit, uranusOrbit, uranusRingOrbit, neptuneOrbit];
var planets = [mercuryMesh, venusMesh, earthMesh, earthCloud, marsMesh, jupiterMesh,
               saturnMesh, saturnRing, uranusMesh, uranusRing, neptuneMesh, sunMesh];
var second = [0.0047, 0.0035, 0.003, 0.003, 0.0024, 0.0013,
          0.001, 0.001, 0.0007, 0.0007, 0.0005];
var timerRevolution = null;

for (var i = 0; i < 11; i++) {
    orbits[i] = new THREE.Group();
    orbits[i].add(planets[i]);
    scene.add(orbits[i]);
}

var orbitRotate = function() {
    for (var i = 0; i < 11; i++)
        orbits[i].rotation.y += second[i];
}

var revolution = function() {
    requestAnimationFrame(render);
    timerRevolution = setInterval(orbitRotate, 1);
    controls.update();
    renderer.render(scene, camera);
}

var stop = function() {
    clearInterval(timerRevolution);
}
```

공전 함수는 setInterval을 통해 구현했습니다. 행성마다 공전 속도가 다르기 때문에 상대적인 값을 계산해서 second 배열에 넣었습니다.

orbitRotate 함수에서 각 행성을 Y축 기준으로 일정한 값만큼 회전시키고, 이를 setInterval 함수로 호출했습니다. 여기에서 Y축이란 태양을 뜻합니다. 따라서 태양을 기준으로 원운동하는 모습을 연출할 수 있었습니다.

```
// Rotation (자전)
var frameSecond = 60;
var rotateSpeed = [0.001, 0.001, 0.01, 0.01, 0.005, 0.05,
                   0.045, 0.045, 0.025, 0.025, 0.025, 0.02];

var rotation = function () {
    setTimeout(function() {
        requestAnimationFrame(rotation);
    }, 1000 / frameSecond);

    for (i = 0; i < 12; ++i){
        if (planets[i] == saturnRing || planets[i] == uranusRing) {
            planets[i].rotateZ(rotateSpeed[i]);
            continue;
        }
        planets[i].rotateY(rotateSpeed[i]);
    }

    renderer.render( scene, camera );
};

rotation();
```

자전 함수는 setTimeout을 통해 구현했습니다. 공전 속도가 모두 다른 것처럼 자전 속도도 행성마다 다릅니다. 이 또한 상대적인 값을 계산해서 rotateSpeed 배열에 넣었습니다.

고리가 없는 행성은 해당 객체의 Y축을 기준으로 회전하도록 했습니다. 동작 결과 고리가 있는 토성과 해왕성은 Z축을 기준으로 회전해야 같이 회전하는 결과를 보였습니다.

viewpoint.js

```
// planet viewpoint function
var sun = function() {
    requestAnimationFrame(render);
    camera.position.set(51, 0, 0);
    controls.update();
    renderer.render(scene, camera);
}

var mercury = function() {
    requestAnimationFrame(render);
    camera.position.set(52, 0, 0);
    controls.update();
    renderer.render(scene, camera);
}

var venus = function() {
    requestAnimationFrame(render);
    camera.position.set(70, 0, 0);
    controls.update();
    renderer.render(scene, camera);
}
```

위 코드에서 보이는 함수들은 galaxy.html 파일에 있는 행성 버튼에 대한 onclick 함수입니다. 카메라 시점을 행성과 근접한 위치로 잡아 특정한 행성 버튼을 클릭했을 때 바로 앞에서 볼 수 있도록 구현했습니다.

```
// look around function
var start = 0;
var add = 0.004;
var timerView = null;

var lookAround = function() {
    camera.lookAt(new THREE.Vector3(0, 0, 0));
    camera.position.x = 240 * Math.sin(start);
    camera.position.z = 240 * Math.cos(start);
    start += add;
}

var view = function() {
    camera.position.set(0, 80, 240);
    requestAnimationFrame(render);
    timerView = setInterval(lookAround, 1);
    controls.update();
    renderer.render(scene, camera);
}

var stop2 = function() {
    clearInterval(timerView);
}
```

view 함수도 공전 함수처럼 setInterval로 구현했습니다. lookAround 함수에서 카메라의 X, Y 시점을 일정한 크기로 변화시켜 원운동하는 모습을 표현할 수 있었습니다. lookAt 속성을 이용해 항상 원점인 태양을 바라보도록 만들었습니다.

3. 프로젝트 결과

(1) 예상 문제점

사실 태양의 크기가 다른 행성에 비해 비교 불가능할 정도로 크기 때문에 현실적인 크기를 고려하기 어려울 것 같습니다. 그리고 각 행성의 자전을 구현하는 것보다 공전을 구현하는 것에 훨씬 많은 시간을 투자해야 할 것이라고 판단됩니다.

(2) 대응 방안

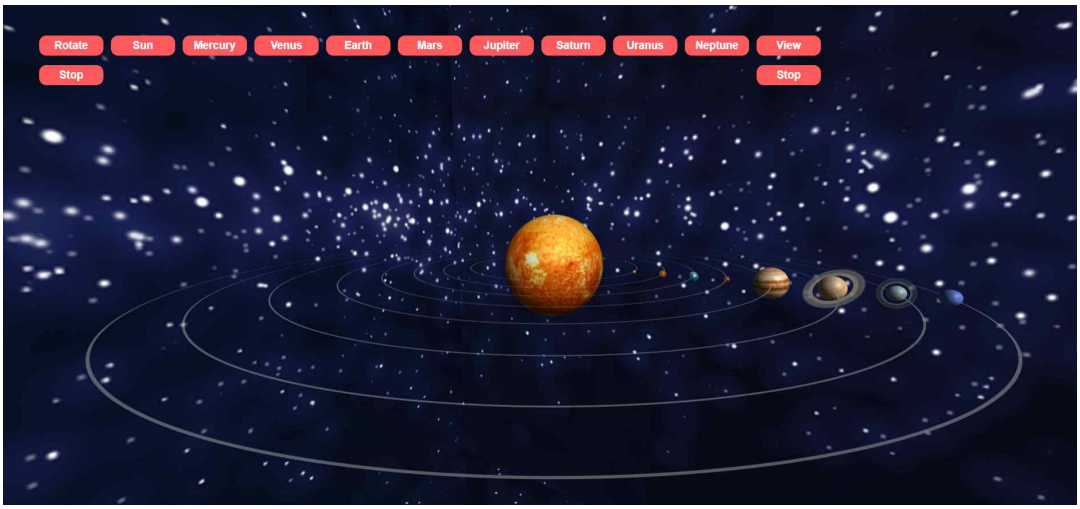
행성의 크기는 현실적인 비율로 구현하지 못하는 것뿐이지 제작하는 것 자체에 대한 어려움은 없을 것 같습니다.

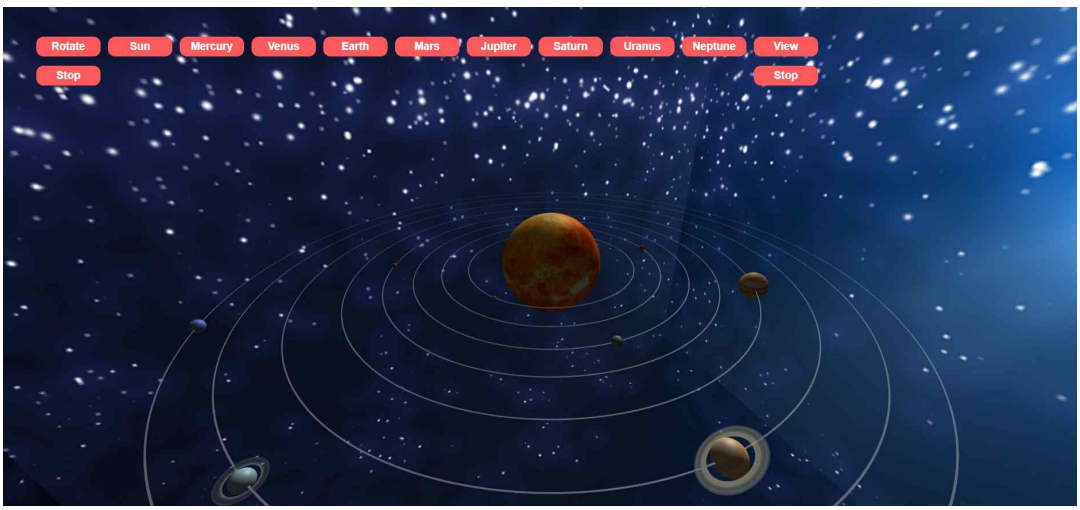
행성의 공전 같은 경우에는 다양한 애니메이션 효과를 찾아보고 공부해야 할 것 같습니다. 추가적으로 태양 빛만을 고려할지 아니면 일괄적인 조명을 비춰줘야 할지도 고민해봐야겠습니다.

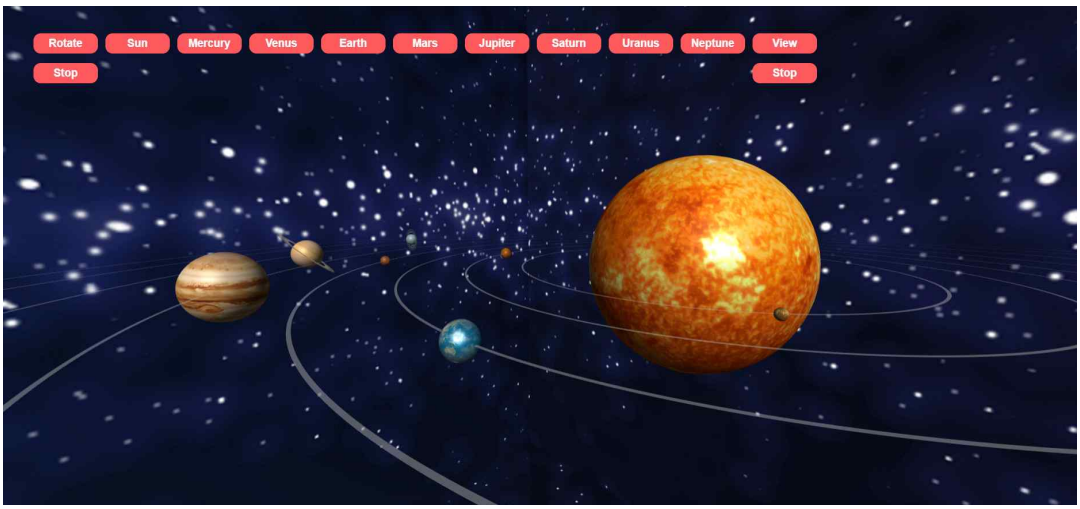
(3) 추진계획 및 실적

업무내용	5					6				
	1	2	3	4	5	1	2	3	4	5
아이디어 구상										
제안서 작성										
프로그램 설계										
테스트										
수정 및 보완										
최종제작										
최종보고서 작성										

(4) 과제 결과물


Rotate
Sun
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
View
Stop
Stop


Rotate
Sun
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
View
Stop
Stop


Rotate
Sun
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
View
Stop
Stop

4. 후기

(1) 느낀점

이번 프로젝트를 통해 WebGL, 특히 three.js의 전반적인 구조를 이해할 수 있었습니다. 객체라는 것이 어떻게 모델링되고, 어떻게 화면에 비춰지며, 어떠한 방식으로 움직이게 할 수 있는지 등 프로젝트만으로 큰 공부가 되었습니다.

아쉬웠던 점은 태양을 사실적으로 표현하지 못했다는 것입니다. 단순하게 텍스쳐 매핑만 한 것이기 때문에 실제 태양처럼 아주 강렬한 빛을 뿜어내는 것처럼 보이게 하지는 못했습니다. 그리고 cubemap을 통해 구현한 배경에서 각 사진의 경계가 보이지 않도록 만들어야 하는데 그렇지 못한 것 또한 문제라고 생각합니다. 기능을 더 추가하는 것보다는 이런 사소한 것을 수정해 더 깔끔한 결과물을 완성하고 싶습니다.

그래도 제가 초기에 구상한 것 이상의 결과물을 뽑아냈다고 생각합니다. 종종 오류가 발생해 객체가 사라질 때마다 가슴 철렁했지만 오류를 고치는 과정에서 많이 성장할 수 있었습니다. 시각적으로 보이는 것을 좋아하는 저에게는 재미있게 작업할 수 있는 프로젝트였습니다.