

1. 작동 방법

Q를 누르면 물체가 움직임. 왼쪽 마우스로 드래그하면 회전, 오른쪽 마우스로 드래그하면 이동

W를 누르면 카메라가 움직임. 왼쪽 마우스로 드래그하면 회전, 오른쪽 마우스로 드래그하면 이동. 마우스 휠로 카메라 전진 후진

2. 렌더러 설정

```
1 // setting render
2 const RENDER_WIDTH = window.outerWidth;
3 const RENDER_HEIGHT = window.outerHeight;
4 const renderer = new THREE.WebGLRenderer();
5 renderer.setSize(RENDER_WIDTH, RENDER_HEIGHT);
6
7 // push my render in html
8 const container = document.getElementById("myContainer");
9 container.appendChild(renderer.domElement);
```

```
1 <div id="myContainer"></div>
```

렌더러 사이즈는 윈도우 창의 세로, 가로로 설정했다. 이 렌더창을 index.html에 정의되어 있는 myContainer div에 자식으로 넣는다.

3. 카메라 셋팅

Let's place the camera along the z-axis (0, 0, 20) looking at the origin, and up direction is (0, 1, 0)
Projection into (camera) image is performed with vertical FOV 90 deg

과제 조건 : 카메라를 (0,0,20)에 위치시키고 원점을 바라본다. 업벡터는 (0,1,0)이다. 카메라의 fov는 90도이다.

```

1 // camera setting
2 const CAMERA1 = new THREE.PerspectiveCamera(
3   90, //fov
4   RENDER_WIDTH / RENDER_HEIGHT, //aspect ratio
5   1, //near
6   500 //far
7 );
8 CAMERA1.position.set(0, 0, 20); //world의 (0,0,20)에 위치
9 CAMERA1.up.set(0, 1, 0); // up vector 설정~
10 CAMERA1.lookAt(0, 0, 0); // 원점을 바라보고 있다. lookat벡터는 -z축이 될 것.

```

과제 조건을 만족시키는 코드

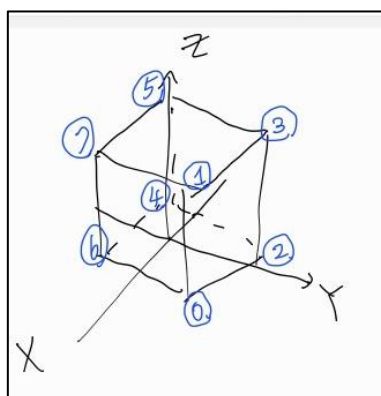
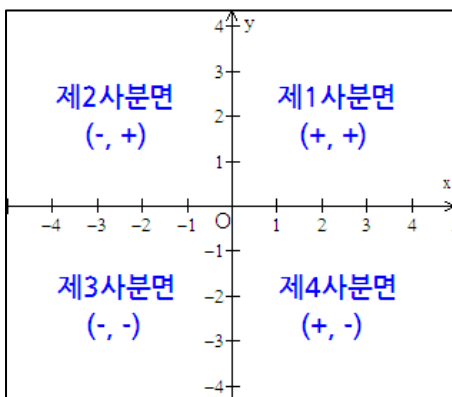
4. vertex로 cube 만들기

정육면체 큐브를 만들려면 정점(Vertex)가 8개 필요하다 8개를 만들겠다.

```

1 //Step 1 . make cube using vertex
2 // geometry setting
3 const VERTICES = []; //vertex를 담을 배열.
4 const CUBE_LENGTH = 10; //cube의 길이
5 const minus = [[1, 1],[-1, 1],[-1, -1],[1, -1]]; //vertex x,y에 곱해질 값.
6
7 for (let i = 0; i < 4; i++) {
8   let x = (CUBE_LENGTH / 2) * minus[i][0];
9   let y = (CUBE_LENGTH / 2) * minus[i][1];
10  VERTICES.push(x, y, CUBE_LENGTH / 2);
11  VERTICES.push(x, y, -CUBE_LENGTH / 2);
12 }

```



-> vertex 배열의 인덱스에 따른 위치

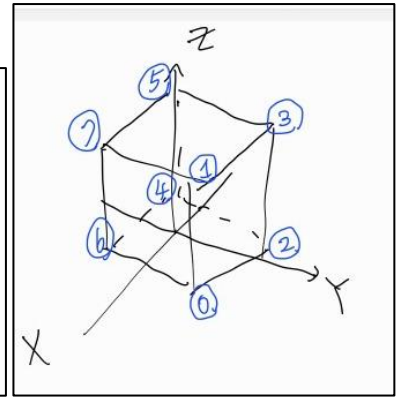
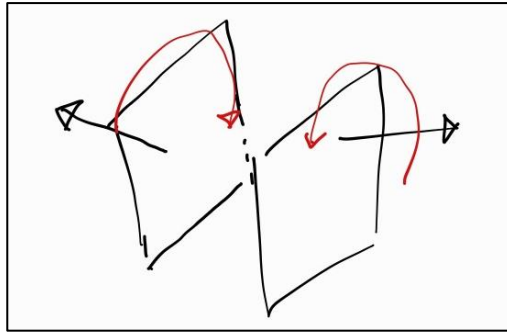
minus 배열은 4분면을 나타냅니다.

x, y에는 큐브 길이 1/2이 들어간다. 여기서 5다. x, y에 각 분면에 가도록 부호를 곱한다. 그러면 원점을 중심으로 하는 길이 10인 큐브의 8개 꼭짓점 x,y 좌표를 구한 것이다. 그리고 z의 부호가 5, -5인 꼭짓점이 각 분면마다 두 개씩 있다. 이렇게 x, y, z 좌표를 구하고 VERTICES 배열에 넣어준다.

```

1 //Index
2 const INDEXES = [
3   0, 1, 7, 7, 6, 0,
4   2, 3, 1, 2, 1, 0,
5   4, 5, 3, 4, 3, 2,
6   6, 7, 5, 6, 5, 4,
7   1, 3, 5, 1, 5, 7,
8   2, 0, 6, 2, 6, 4
9 ];

```



INDEX 배열. 면의 수직인 벡터가 면 바깥쪽을 향하게 정해주었다

```

1 const geometry = new THREE.BufferGeometry();
2 //position attribute 설정
3 geometry.setAttribute(
4   "position",
5   new THREE.Float32BufferAttribute(VERTICES, 3)
6 );
7 //index 설정
8 geometry.setIndex(INDEXES);
9
10 // material setting
11 const material = new THREE.MeshBasicMaterial({
12   color: 0xffff00,
13   wireframe: true,
14 });
15
16 // cube model
17 let myMesh = new THREE.Mesh(geometry, material);

```

VERTICES 배열로 position attribute를 설정하고 index를 설정한다. 또 매터리얼도 설정한다. 그래서 geometry와 매터리얼로 Mesh를 만든다.

5. 큐브 월드에 배치하기


```

4  // axis model
5  const axesHelper = new THREE.AxesHelper(5);
6
7  // create my world (scene)
8  const myScene = new THREE.Scene();
9
10 //scene에 큐브랑 축 추가
11 myScene.add(myMesh);
12 myScene.add(axesHelper);
13
14 function animate() {
15     requestAnimationFrame(animate);
16     renderer.render(myScene, CAMERA1);
17 }
18 animate();

```

씬을 만들고 큐브를 씬에 추가해준다.

6. 이벤트 등록 및 작성



```

1  // register event-callback functions into renderer's dom
2  renderer.domElement.style = "touch-action:none";
3  renderer.domElement.onpointerdown = mouseDownHandler; // -> mouse click
4  renderer.domElement.onpointermove = mouseMoveHandlerTemp; // -> mouse move
5  renderer.domElement.onpointerup = mouseUpHandler; // -> mouse relase
6  renderer.domElement.onwheel = mouseWheelHandler; // -> mouse wheel!
7

```

event를 등록한다.

[키 입력 이벤트]



```

1  // key 입력 이벤트
2  window.addEventListener("keydown", function (event) {
3      switch (event.code) {
4          case "KeyQ": //Q
5              mode = MODE.OBJECT; //Q를 입력하면 object가 움직입니다.
6              break;
7          case "KeyW": //W
8              mode = MODE.CAMERA1; //W를 입력하면 camera가 움직입니다.
9              break;
10     }
11 });

```

키를 입력 받는 이벤트다. Three.js example에서 키 입력 이벤트를 window 객체에서 하길래 그렇게 했다. event.code에는 사용자가 입력한 키가 저장되어 있다. Q를 입력하면 큐브를 움직이고 W를 입력하면 카메라를 움직인다.

```
1 const MODE = { NONE : -1, OBJECT: 0, CAMERA: 1 }; //Object mode, camera mode
2 var mode = MODE.NONE; // 사용자가 입력한 키에 따라 바뀜. 초기는 None Mode.
```

전역 변수 mode의 초기값은 MODE.NONE이다.

Q 입력 -> 전역 변수 mode가 MODE.OBJECT로 설정

W 입력 -> 전역변수 mode가 MODE.CAMERA로 설정

[마우스 이벤트]

```
1 let left_down_flag = false;
2 let right_down_flag = false;
```

마우스 왼쪽 클릭인지, 오른쪽 클릭인지 판별하는 변수

- 마우스 release 이벤트

```
1 function mouseUpHandler(e) {
2   //마우스를 떼었을 때, 뗐다는 표시를 합니다.
3   if (left_down_flag) left_down_flag = false;
4   else if (right_down_flag) right_down_flag = false;
5 }
```

마우스를 클릭했다는 표시를 false로 바꿔준다.

- 마우스 클릭 이벤트

```
1 function mouseDownHandler(e) {
2   //사용자가 모드를 입력 안했을 시에는 마우스 이벤트를 종료합니다.
3   if (mode == MODE.NONE) return;
4
5   if (e.button == 0) left_down_flag = true; //왼쪽 버튼 누르면 표시
6   else if (e.button == 2) right_down_flag = true; //오른쪽 버튼 누르면 표시
7   //prev_x, prev_y 처음 값 설정
8   prev_x = e.clientX;
9   prev_y = e.clientY;
10 }
```

e.button이 0이면 왼쪽 버튼을 클릭했다는 의미다. left_dwon_flag를 true로 설정한다. e.button이 1이

면 오른쪽 버튼을 클릭했다는 의미이다. right_down_flag를 true로 설정한다.

마우스를 움직이기 전 좌표를 prev_x, prev_y에 저장한다.

- 마우스 이동 이벤트

```
1 function mouseMoveHandlerTemp(e) {
2   if (left_down_flag || right_down_flag) {
3     //현재 mouse 좌표 world space로 변환.
4     let newPosWS = compute_pos_ss2ws(e.clientX, e.clientY);
5     //이전 mouse 좌표 world space로 변환 differenc를 구하기 위함.
6     let prevPosWS = compute_pos_ss2ws(prev_x, prev_y);
7
8     switch (mode) {
9       case MODE.OBJECT:
10        // 물체를 움직이자!
11        if (left_down_flag) {
12          //물체를 회전 하는 함수 호출
13          ObjectRotate(newPosWS, prevPosWS);
14        } else if (right_down_flag) {
15          //물체를 이동 하는 함수 호출
16          ObjectMove(newPosWS, prevPosWS);
17          //지금 좌표를 이전 좌표로 저장.
18        }
19        break;
20       case MODE.CAMERA:
21        //카메라를 움직이자!
22        if (left_down_flag) {
23          //카메라를 회전 함수 호출
24          CameraRotate(newPosWS, prevPosWS);
25        } else if (right_down_flag) {
26          //카메라를 이동 함수 호출
27          CameraMove(newPosWS, prevPosWS);
28        }
29        break;
30       default:
31        console.log("mode off please press Q or W");
32        break;
33     }
34     //현재 좌표를 이전 좌표로 저장.
35     prev_x = e.clientX;
36     prev_y = e.clientY;
37   }
38 }
```

move이벤트에서는 일단 사용자가 버튼을 클릭했는지 먼저 확인한다. 클릭 안 했으면 회전, 이동을 안 시켜야 하기 때문에 그냥 종료한다. 클릭을 했으면 이제 마우스 이동에 따라 각도와 길이를 구해서 회전과 이동을 시킬 것이다.

left mouse 클릭, right mouse 클릭을 하면 일단 그 클릭한 스크린스페이스상 좌표를 월드 스페이스 좌표로 변환한다. 그리고 Mode가 Object인지 Camera인지에 따라 switch문으로 분기했다. 각 분기문 안에서는 또 왼쪽 클릭인지 오른쪽 클릭인지에 따라 이동시키는 함수, 회전시키는 함수를 호출한다. 이 함수들의 인수로는 스크린스페

이스에서 마우스 클릭좌표를 월드 스페이스로 변환한 좌표 값이다.

- mouse wheel 이벤트

```
1 function mouseWheelHandler(e) {
2   if (mode !== MODE.CAMERA) return;
```

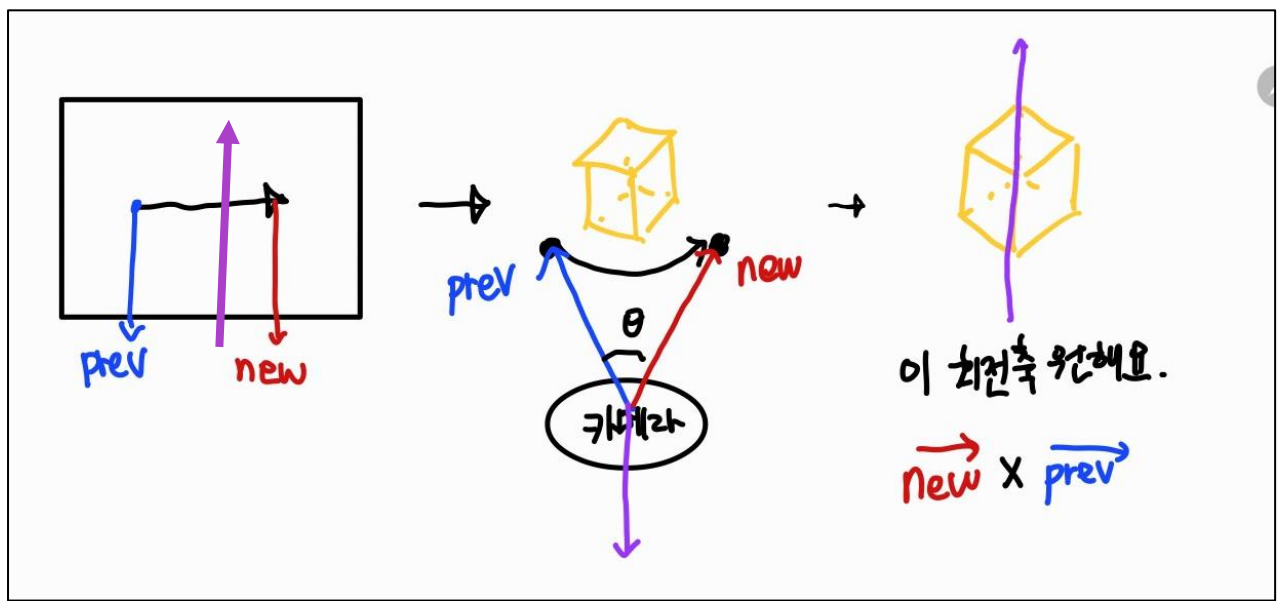
현재 mode가 CAMERA 모드가 아니면 WHEEL 이벤트에 아무 일도 일어나지 않는다.

mouseWheelHandler의 함수 전문은 7-4에서 설명하겠다. e.deltaY를 통해 카메라의 z축상의 좌표를 바꾸어 카메라를 전진 후진 한다.

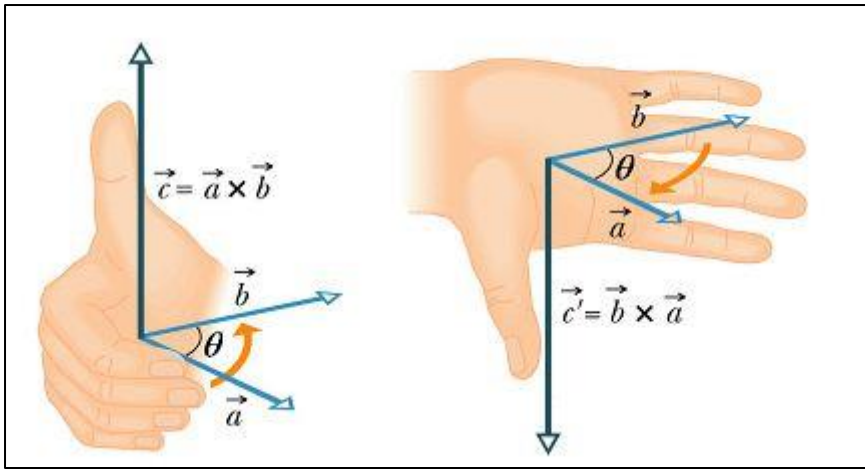
7. 이동, 회전 함수 작성하기

큐브와 카메라는 바로 위의 부모가 월드이다. three.js의 .matrix는 로컬 트랜스폼을 위한 매트릭스다. 큐브와 카메라는 바로 위의 부모가 월드이기 때문에 .matrix가 결국 월드로의 변환 매트릭스가 된다. .matrix나 .position, .scale, .rotation등을 수정하면 결국 물체를 월드에 어떻게 배치할지 수정 되는 것이다.

7-1) object 회전

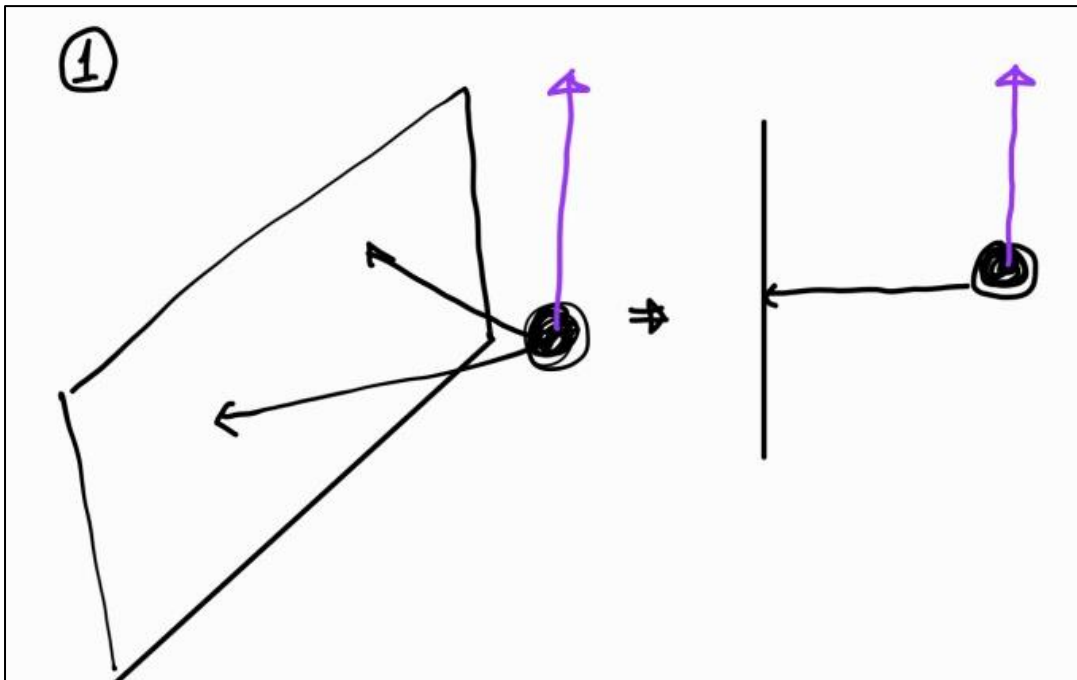


사용자가 왼쪽에서 오른쪽으로 마우스를 드래그하면 큐브는 반시계 방향으로 회전한다. 회전각은 카메라에서 prev로 가는 마우스 벡터, 카메라에서 new로 가는 마우스 벡터를 내적해서 $\cos\theta$ 를 구하고, 거기에 아크코사인해서 θ 를 구할 수 있다. 그리고 이 두 벡터로 회전 축을 구한다. 마우스를 왼쪽에서 오른쪽으로 드래그할 때, 반시계 방향으로 회전을 해야하니 $\vec{new} \times \vec{prev}$ 로 외적을 진행한다.

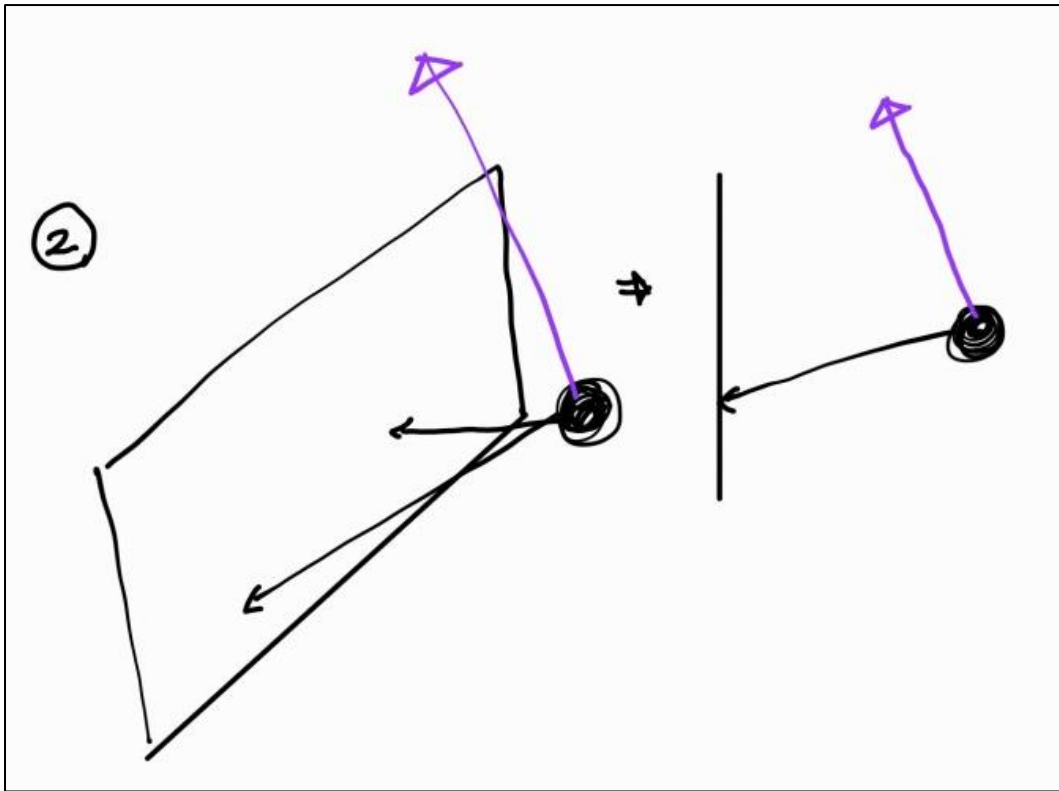


외적은 오른손 법칙을 따른다.

근데 카메라의 위치로 벡터를 구하면 문제가 있다!!

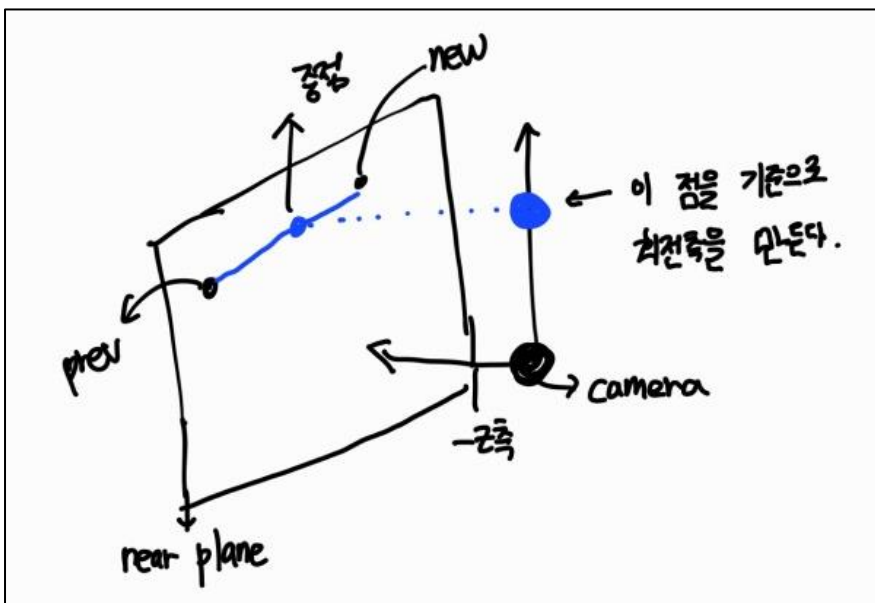


1번그림과 같이 스크린 좌표 두 개의 중점이 카메라위치와 같다면 회전축이 near plane에 평행하도록 잘 구해진다.



2번그림과 같이 스크린 좌표 두 개의 중점이 카메라위치와 같지 않다면 회전축이 near plane에 평행하지 않는다. 사용자 입장에서는 스크린 스페이스에서의 회전축으로 회전을 하고 싶을 텐데, 이런 현상을 고려하지 않는다면 예상하지 못한 회전이 나올 것이다.

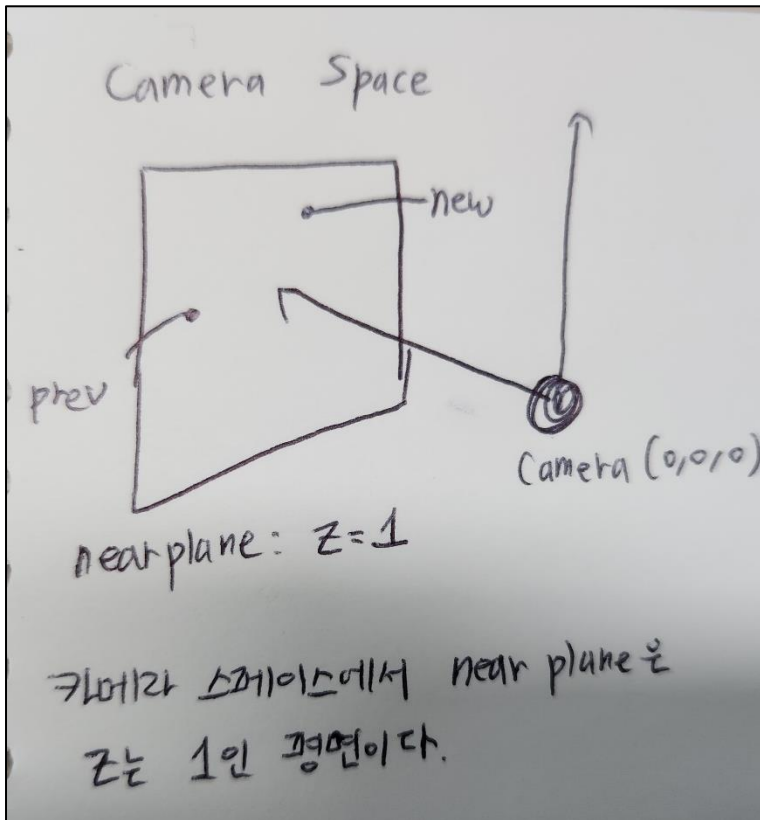
(해결 방법)



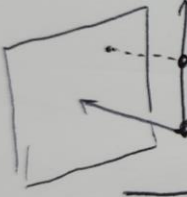
스크린 좌표는 카메라 스페이스에서 near plane에 멧히게 된다. near plane은 카메라의 lookat 방향으로 near만큼 떨어져 있다. 이전 스크린 좌표와 새로운 스크린 좌표의 중점을 구한다. 그 중점을 Center라 하겠다. Center를 마이너스 lookat 방향으로 near만큼 이동하면 파란색 동그라미 점을 구할 수 있다! 이 점을 Temp_Cam이라 하겠다. Temp_Cam에서 이전 스크린 좌표, Temp_Cam에서 새로운 스크린 좌표로 가는 벡터를 구하고 회전축을 구한다.

이렇게 구한 회전축은 near plane에 평행한다.

(증명)



TempCam ->prev와 TempCam->new의 벡터를 통해 구한 회전축 벡터가 near plane에 있어야 한다. 회전축 벡터를 near plane의 법선벡터와 내적하면 0인 것을 증명하면 된다.

$(0,0,0)$ $\text{new} = (x_1, y_1, 1)$
 $\text{prev} = (x_2, y_2, 1)$
 $\text{중점} = \left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}, 1 \right)$
 은
 중점을 y 축에 그대로 정사영 한다.

 $\left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}, 0 \right)$
 이 점을 ~~center~~ T라고 칭함.
 $\vec{T_{\text{New}}} = \left(\frac{x_1-x_2}{2}, \frac{y_1-y_2}{2}, 1 \right)$
 $\vec{T_{\text{prev}}} = \left(\frac{x_2-x_1}{2}, \frac{y_2-y_1}{2}, 1 \right)$
 $\vec{T_{\text{New}}} \times \vec{T_{\text{prev}}} = \text{Rot-axis}$
 $\text{Rot-axis} \cdot (0,0,1) = 0$ 인것을 보이자.

near plane의 법선벡터는 $(0,0,1)$ 이다. 구한 회전축과 $(0,0,1)$ 의 내적이 0이면 된다.

$$\vec{T_{\text{New}}} \times \vec{T_{\text{prev}}} = \left(\frac{y_1-y_2-y_2+y_1}{2}, \frac{x_2-x_1-x_1+x_2}{2}, \frac{(x_1-x_2)(y_2-y_1)}{4} - \frac{(y_1-y_2)(x_2-x_1)}{4} \right)$$

↓
 $(0,0,1)$ 과 내적하면

$$0 + 0 + \frac{(x_1-x_2)(y_2-y_1)}{4} - \frac{(y_1-y_2)(x_2-x_1)}{4} \text{ 이다.}$$

$$\frac{(x_1-x_2)(y_2-y_1)}{4} - \frac{(y_2-y_1)(x_1-x_2)}{4} = 0.$$

이렇게 구한 회전축은 항상
 near plane에 ~~평행~~ ~~수직~~
 평행한다.

이로써 항상 평행하게 맺히는 것이 증명되었다.

```

1  function ObjectRotate(new_posWorld, prev_posWorld) {
2      //스크린 좌표들의 중점을 구한다.
3      let center = new THREE.Vector3(
4          (new_posWorld.x + prev_posWorld.x) / 2,
5          (new_posWorld.y + prev_posWorld.y) / 2,
6          (new_posWorld.z + prev_posWorld.z) / 2
7      );
8      // camera의 -z축을 구하자.
9      let lookAt = new THREE.Vector3();
10     //getWorldDirection이라는 친절한 함수를 사용하여 -z축을 구함.
11     CAMERA1.getWorldDirection(lookAt);
12     //CENTER의 z축으로 1만큼 이동
13     center = new THREE.Vector3(
14         center.x + -lookAt.x,
15         center.y + -lookAt.y,
16         center.z + -lookAt.z
17     );

```

Temp_cam을 구하는 좌표다. 월드를 기준으로 한 스크린 좌표의 중점을 구한다.카메라의 getWorldDirection으로 lookat을 구했다. 중점에 마이너스 lookat 방향으로 1(near) 만큼이동한다.

```

19     //c는 임시 카메라 좌표, n은 새로운 스크린좌표를 의미. 임시 카메라에서 스크린 new 방향의 벡터입니다.
20     let CN = new THREE.Vector3();
21     CN = CN.subVectors(new_posWorld, center);
22
23     //c는 임시 카메라, p은 이전 스크린좌표를 의미. 임시 카메라에서 스크린 prev 방향의 벡터입니다.
24     let CP = new THREE.Vector3();
25     CP = CP.subVectors(prev_posWorld, center);
26
27     //유닛벡터를 구합니다.
28     let normalCN = CN.normalize();
29     let normalCP = CP.normalize();
30
31     //rot_axis => 외적을 통해 구할 수 있다.
32     let rot_axis = new THREE.Vector3();
33     rot_axis.crossVectors(normalCN, normalCP); //회전축!
34     rot_axis.normalize(); // 노말라이즈를 해주어야 합니다!
35
36     //between anlge => 두 벡터 사이의 각도
37     let bet_angle = normalCP.angleTo(normalCN);

```

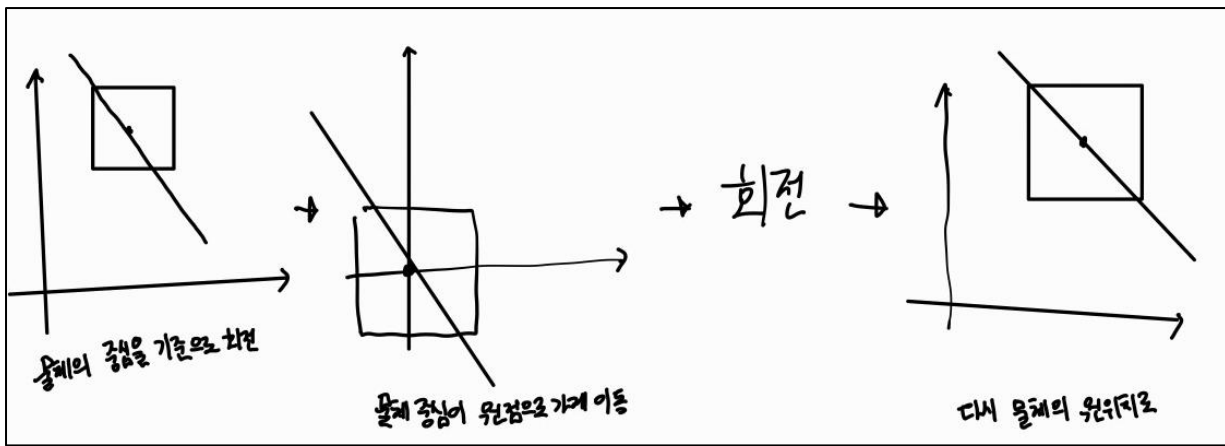
임시 카메라 좌표(Temp_Cam)에서 스크린 방향으로 가는 벡터를 구했다. 함수는 subVector를 썼다. 이 함수는 두 개의 벡터를 받으면 첫 번째 벡터에서 두 번째 벡터를 뺀 벡터를 반환해준다. CP와 CN을 구했으면 normalize()함수로 정규화를 한다. 회전축 rot_axis는 crossVector()함수를 통해 구한다. 이 함수는 두 개의 벡터를 받으면 외적을 하고 그 결과를 반환한다. rot_axis도 normalize를 꼭 해야한다! 이후에 나올 .setFromAxisAngle() 함수를 사용할 때 정규화 된 벡터를 넣어야 하기 때문이다. CP와 CN사이의 각도 bet_angle은 angleTo()라는 아주 친절한 함수를 통해 구한다. 이 함수는 두 벡터 사이의 각도를 라디안 값으로 반환한다.

```

39     let quaternion = new THREE.Quaternion();
40     quaternion.setFromAxisAngle(rot_axis, bet_angle); //이걸로 회전 행렬을 구하자.

```

r회전축과 회전각이 있으니 쿼터니언을 구한다.



물체의 중심을 지나는 회전축을 기준으로 회전하려면 그림처럼 해야한다. 그러면 우리가 큐브에 적용할 매트릭스는 “중심을 원점으로 위치시키는 이동행렬 x 회전행렬 x 원점에 있는 중심을 다시 원위치 시키는 이동행렬”이다.

```

42 //쿼터니언으로 회전행렬을 구한다.
43 const matR = new THREE.Matrix4().makeRotationFromQuaternion(quaternion);
44
45 //그 다음 큐브를 원점으로 이동시키는 행렬
46 const matT = new THREE.Matrix4().makeTranslation(
47     -myMesh.position.x,
48     -myMesh.position.y,
49     -myMesh.position.z
50 );
51
52 //다시 큐브를 원위치로 돌려놓는 행렬.
53 const matT_inv = new THREE.Matrix4().makeTranslation(
54     myMesh.position.x,
55     myMesh.position.y,
56     myMesh.position.z
57 );

```

앞서 구한 쿼터니언과 큐브의 위치를 가지고 행렬을 구했다.

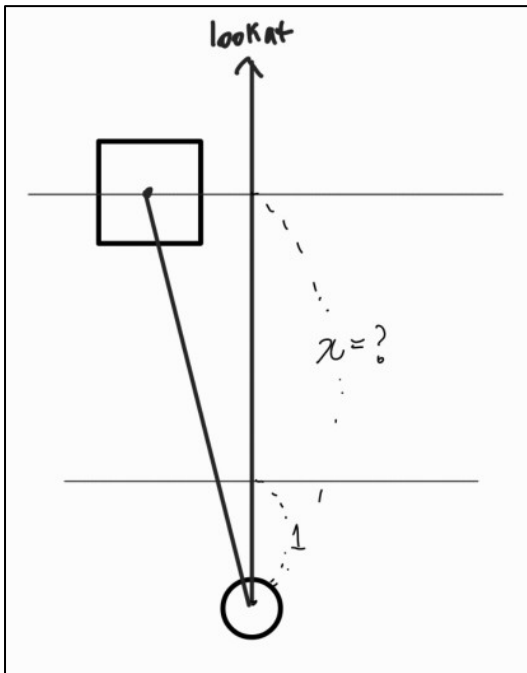
```

44 //---- myMesh(내가 만든 큐브) matrix 업데이트----
45 myMesh.matrixAutoUpdate = false;
46 let mat_prev = myMesh.matrix.clone();
47 mat_prev.premultiply(matT);
48 mat_prev.premultiply(matR);
49 mat_prev.premultiply(matT_inv);
50 myMesh.matrix = mat_prev;
51 }

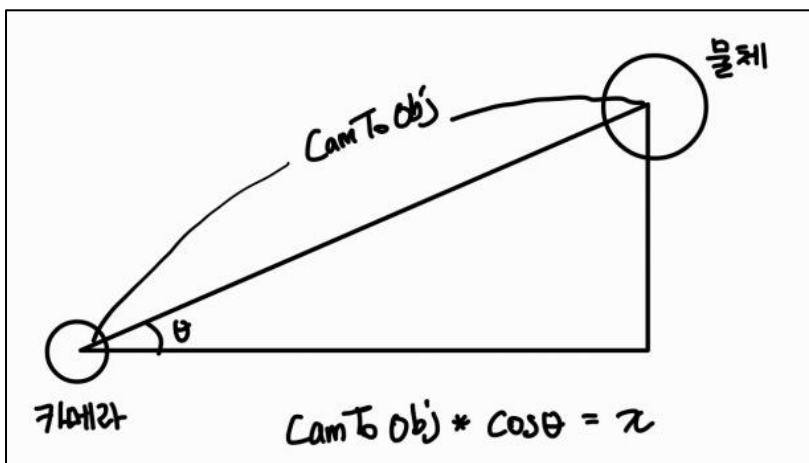
```

큐브 매트릭스를 업데이트 한다. .matrixAutoUpdate를 false로 지정하여 큐브의 position, rotation, scale 속성에 맞게 업데이트 되지 않게 한다. 내가 지정한 매트릭스로 트랜스폼 할 것이라는 의미이다. 큐브의 기존 트랜스폼에 덧 붙여서 트랜스폼을 진행해야 한다. 그래야 연속된 느낌이 들어야 하기 때문이다. mat_prev에 기존 큐브 매트릭스를 불러온다. 이 mat_prev에 앞서 구한 매트릭스를 앞에 곱한다. 곱해진 mat_prev를 큐브 매트릭스에 저장한다.

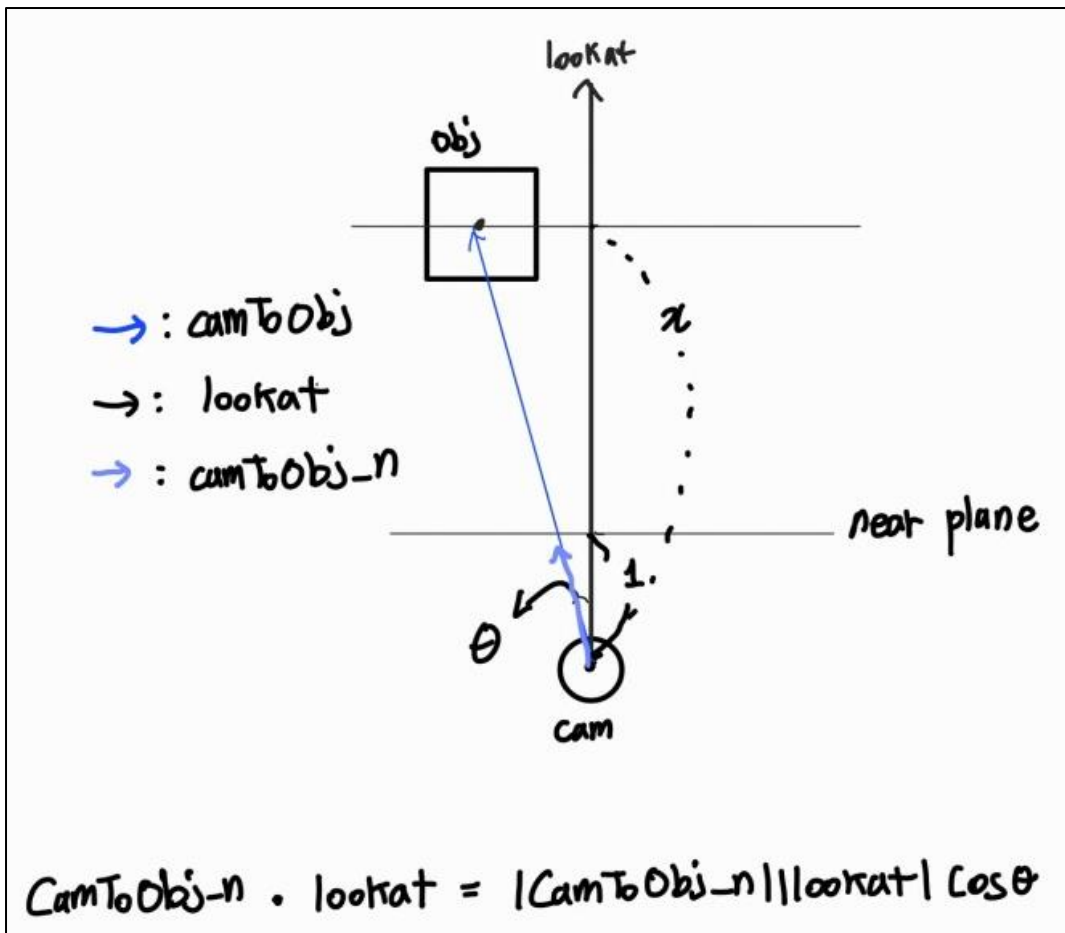
3-2) object 이동



그림에서 미지수 x 를 구하면 된다. near plane 거리와 x 와의 비율을 통해서 screen에서 움직인 게 실제로 물체를 얼마나 움직이는지 구할 수 있다.



x 는 "카메라에서 물체까지의 벡터"에서 lookat 벡터에 정사영을 내리면 된다. 그럼 $\cos\theta$ 만 구하면 된다.



camToObj를와 lookat을 내적하고 각각의 길이로 나눠주면 cos를 구할 수 있다. camToObj를 정규화 하면 어차피 길이가 1이어서 길이를 안 나눠줘도 된다.

```

1 function ObjectMove(new_posWorld, prev_posWorld) {
2
3   // camera의 -z축을 구하자.
4   let lookAt = new THREE.Vector3();
5   //getWorldDirection이라는 친절한 함수를 사용하여 -z축을 구함.
6   CAMERA1.getWorldDirection(lookAt);
7
8   //world에서 camera의 position과 world에서 object의 position을 통해 벡터를 구함.
9   //camera에서 object로 향하는 벡터
10  let camToObj = new THREE.Vector3();
11  camToObj.subVectors(myMesh.position, CAMERA1.position);
12  //정규화
13  let camToObj_n = camToObj.clone().normalize();
14
15  //두 벡터로 cos세타 구하기
16  let cos_ = lookAt.dot(camToObj_n); //유닛벡터라서 길이를 딱히 나눠주지 않는다.
17  //벡터 CO * cos_ = CO`
18  let CO_N_length = camToObj.length() * cos_ * 1.0; //float 만들려고 1.0곱함.

```

camToObj는 카메라에서 물체로 가는 벡터다. camToObj 벡터를 정규화하고 lookat 벡터와 내적을 이용해 두 벡터의 cos세타를 구한다. 그러면 camToObj 길이에 이 cos세타를 곱해주면 x를 구할 수 있다. near가 1이니깐 스크린

에서 움직인 것에 x배를 곱해줘서 물체를 이동시키면 된다.

```
19 //이제 비율을 구했다!
20 // near : CO정사영 의 비율로 벡터 differenc를 구하면됨!
21 let diff = new THREE.Vector3();
22 diff.subVectors(new_posWorld, prev_posWorld); // Prev -> New로 가는 벡터
23
24 //벡터에 스칼라를 곱해서 물체 쪽에서는 얼마큼 움직이는지 파악.
25 let diff_s = diff.clone().multiplyScalar(CO_N_length);
26
27 //mesh에 곱해질 매트릭스 선언
28 let matLocal = new THREE.Matrix4();
29 const matT = new THREE.Matrix4().makeTranslation(
30     diff_s.x,
31     diff_s.y,
32     diff_s.z
33 );
```

diff는 prev 마우스 좌표에서 new 마우스 좌표로 가는 벡터다. 이 벡터에 앞서 구한 x(camToObj 정사영 내린 것)를 곱해준다. 스크린에서의 드래그가 물체에서는 x배 만큼 이동한다. 이 값을 통해 이동행렬을 구한다.

```
34 //---- myMesh(내가 만든 큐브) matrix 업데이트----
35 myMesh.matrixAutoUpdate = false;
36 let mat_prev = myMesh.matrix.clone();
37 mat_prev.premultiply(matT);
38 myMesh.matrix = mat_prev;
39
40 //---- myMesh의 position property 업데이트 ----
41 myMesh.position.set(
42     myMesh.position.x + diff_s.x,
43     myMesh.position.y + diff_s.y,
44     myMesh.position.z + diff_s.z
45 );
46 }
```

큐브 매트릭스를 업데이트 한다. .matrixAutoUpdate를 false로 지정하여 큐브의 position, rotation, scale 속성에 맞게 업데이트 되지 않게 한다. 내가 지정한 매트릭스로 트랜스폼 할 것이라는 의미이다. 큐브의 기존 트랜스폼에 덧 붙여서 트랜스폼을 진행해야 한다. 그래야 연속된 느낌이 들어야 하기 때문이다. mat_prev에 기존 큐브 매트릭스를 불러온다. 이 mat_prev에 앞서 구한 매트릭스를 앞에 곱한다. 곱해진 mat_prev를 큐브 매트릭스에 저장한다.

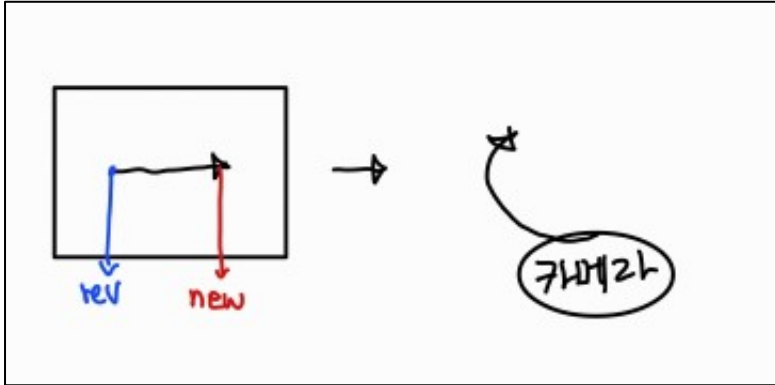
큐브의 position 속성도 업데이트 해야한다. 이유는 camToObj 벡터를 구할 때 마다 cube의 position 속성을 사용하기 때문이다. 현재 큐브의 position에 이동한 만큼 더해준다.

3-3) camera 회전

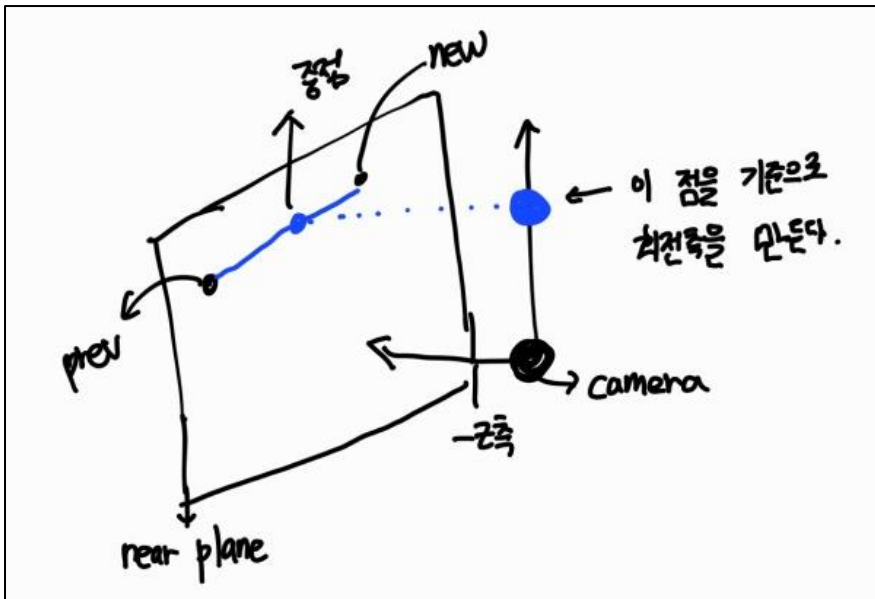


```
1 // camera의 회전축의 중심! 초기값은 원점이다.  
2 let camRotCenter = new THREE.Vector3(0, 0, 0);
```

카메라 회전축 중심. 이 점을 중심으로 회전한다. 카메라가 이동할 때 같이 이동한다.



마우스를 왼쪽에서 오른쪽으로 드래그하면 카메라는 시계 방향으로 회전하도록 설정했다.



큐브를 회전할 때와 비슷하다. 각도를 구하는 방식도 같고 이용하는 벡터도 같다. 다만 회전축의 방향이 다르다. 마우스를 왼쪽에서 오른쪽으로 드래그 하면 카메라는 시계 방향으로 회전하도록 설정해서 $\text{prev} \times \text{new}$ 로 외적을 진행한다.

```

1 function CameraRotate(new_posWorld, prev_posWorld) {
2   //교수님이 처음 회전 중심 원점을 하라 했다..
3
4   //스크린 좌표들의 중점을 구한다.
5   let center = new THREE.Vector3(
6     (new_posWorld.x + prev_posWorld.x) / 2,
7     (new_posWorld.y + prev_posWorld.y) / 2,
8     (new_posWorld.z + prev_posWorld.z) / 2
9   );
10  // camera의 -z축을 구하자.
11  let lookAt = new THREE.Vector3();
12  //getWorldDirection이라는 친절한 함수를 사용하여 -z축을 구함.
13  CAMERA1.getWorldDirection(lookAt);
14  //CENTER의 z축으로 1만큼 이동
15  center = new THREE.Vector3(
16    center.x + -lookAt.x,
17    center.y + -lookAt.y,
18    center.z + -lookAt.z
19  );
20
21  //c는 임시 카메라 좌표, n은 새로운 스크린좌표를 의미. 임시 카메라에서 스크린 new 방향의 벡터입니다.
22  let CN = new THREE.Vector3();
23  CN = CN.subVectors(new_posWorld, center);
24
25  //c는 임시 카메라 좌표, p은 이전 스크린좌표를 의미. 임시 카메라에서 스크린 prev 방향의 벡터입니다.
26  let CP = new THREE.Vector3();
27  CP = CP.subVectors(prev_posWorld, center);

```

큐브를 회전할 때와 똑같이 벡터를 구한다.

```

29  //유닛벡터를 구합니다.
30  let normalCN = CN.normalize();
31  let normalCP = CP.normalize();
32
33  //rot_axis => 외적을 통해 구할 수 있다.
34  let rot_axis = new THREE.Vector3();
35  rot_axis.crossVectors(normalCP, normalCN); //회전축!
36  rot_axis.normalize(); // 노말라이즈를 해주어야 합니다!
37
38  //between anlge => 두 벡터 사이의 각도
39  let bet_angle = normalCP.angleTo(normalCN);

```

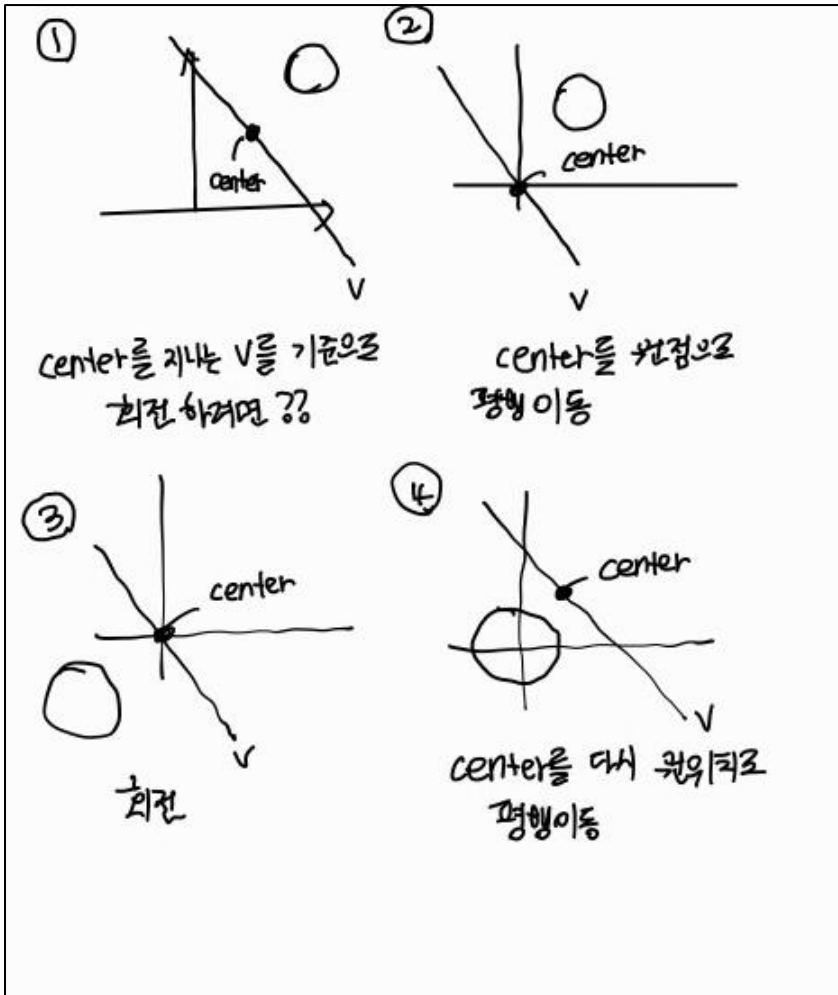
회전축을 구하는데, 큐브와 반대 방향으로 돌리고 싶어서 prev->new로 구한다.

```

41  //이걸로 회전 행렬을 구하자.
42  let quaternion = new THREE.Quaternion();
43  quaternion.setFromAxisAngle(rot_axis, bet_angle); //회전축과 각도로 쿼터니언을 구함.

```

회전축과 회전각으로 쿼터니언을 구했다.



특정 지점을 지나는 회전축을 기준으로 회전하려면 그림처럼 해야한다. 그러면 우리가 카메라에 적용할 매트릭스는 "회전 중심을 원점으로 위치시키는 이동행렬 x 회전행렬 x 원점에 있는 회전 중심을 다시 원위치 시키는 이동행렬"이다.

```

44  const matR = new THREE.Matrix4().makeRotationFromQuaternion(quaternion); //쿼터.
45
46  //그 다음 카메라 회전축 중심이 원점으로 이동하는 행렬을 구한다.
47  const matT_rotCenter = new THREE.Matrix4().makeTranslation(
48    -camRotCenter.x,
49    -camRotCenter.y,
50    -camRotCenter.z
51  );
52
53  // 카메라 회전축 중심이 다시 원 위치로 돌아가는 이동 행렬.
54  const matT_rotCenter_back = new THREE.Matrix4().makeTranslation(
55    camRotCenter.x,
56    camRotCenter.y,
57    camRotCenter.z
58  );

```

회전 행렬과 이동행렬들을 구했다.

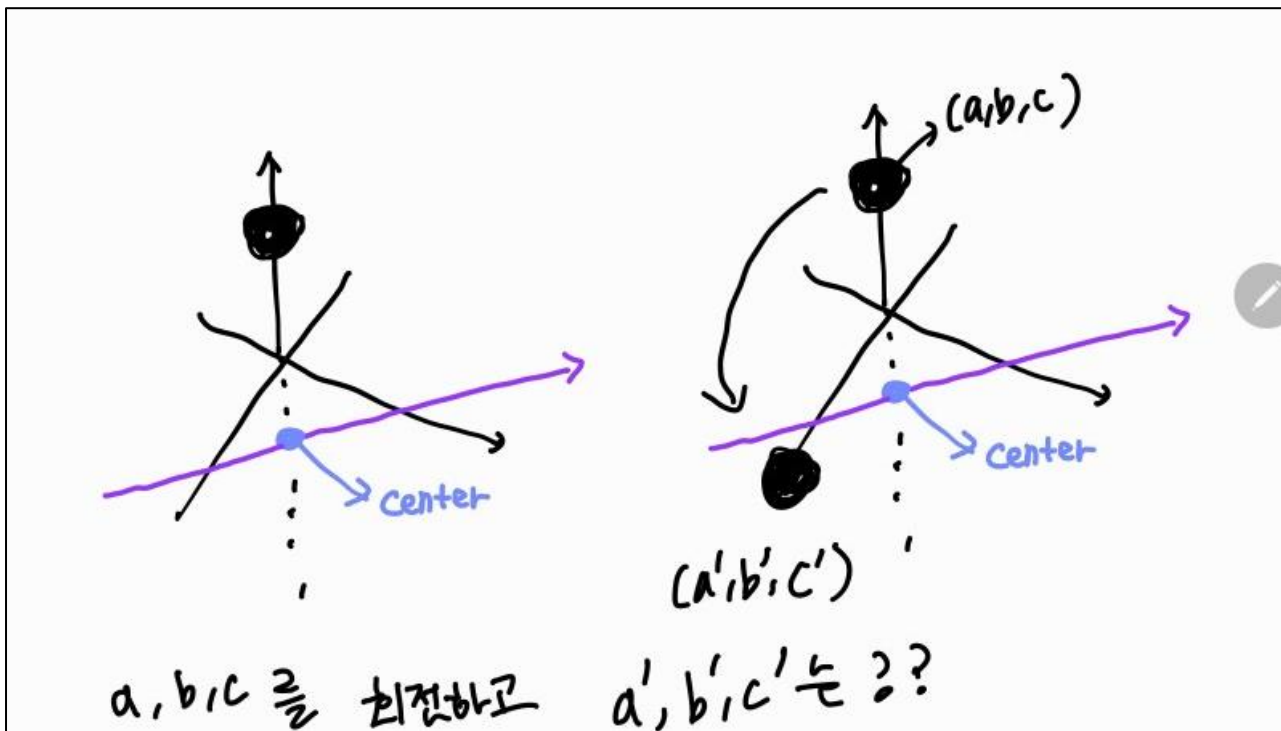
```

44  //---- Camera matrix 업데이트----
45  CAMERA1.matrixWorldNeedsUpdate = true;
46  CAMERA1.matrixAutoUpdate = false;
47  let mat_prev = CAMERA1.matrix.clone();
48  mat_prev.premultiply(matT_rotCenter);
49  mat_prev.premultiply(matR);
50  mat_prev.premultiply(matT_rotCenter_back);
51  CAMERA1.matrix = mat_prev;

```

카메라 매트릭스를 업데이트 한다. `.matrixWorldNeedsUpdate`를 `true`로 설정하여 ViewMatrix가 카메라가 변한 것에 맞춰 업데이트 되도록 한다. `.matrixAutoUpdate`를 `false`로 지정하여 카메라의 position, rotation, scale 속성에 맞게 업데이트 되지 않게 한다. 내가 지정한 매트릭스로 트랜스폼 할 것이라는 의미이다. 카메라의 기존 트랜스폼에 덧 붙여서 트랜스폼을 진행해야 한다. 그래야 연속된 느낌이 들어야 하기 때문이다. `mat_prev`에 기존 카메라 매트릭스를 불러온다. 이 `mat_prev`에 앞서 구한 매트릭스를 앞에 곱한다. 곱해진 `mat_prev`를 카메라 매트릭스에 저장한다.

카메라 position 속성도 업데이트 해야 한다. 그 이유는 다른 함수에서 카메라의 position 속성으로 벡터를 구하기 때문이다.



a', b', c' 를 구하는 방법 :

1. 회전 중심점이 원점으로 가는 만큼 점을 평행 이동한다.
2. 월드 상의 회전축으로 회전한다.
3. 회전 중심점이 원 위치로 가는 만큼 점을 평행 이동한다.

```

69 //----- Camera의 position property 업데이트 -----
70 // 회전축 중심이 원점으로 이동한 만큼 카메라 위치도 이동
71 let position = new THREE.Vector4(
72     CAMERA1.position.x - camRotCenter.x,
73     CAMERA1.position.y - camRotCenter.y,
74     CAMERA1.position.z - camRotCenter.z,
75     1
76 );
77 //이동한 점에 회전 행렬 곱해
78 position.applyMatrix4(matR);
79
80 //카메라 회전 중심을 다시 원 위치로 이동한 만큼 카메라도 이동
81 position.x = position.x + camRotCenter.x;
82 position.y = position.y + camRotCenter.y;
83 position.z = position.z + camRotCenter.z;
84 //이 위치를 카메라 포지션에 저장
85 CAMERA1.position.set(position.x, position.y, position.z);
86 }

```

벡터 position 변수에는 카메라의 새로운 위치가 담길 것이다. 카메라의 기존 위치를 회전 중심점이 원점으로 이동한 만큼 이동해준다. 그 위치 값이 position 변수에 저장된다. 그리고 이 position 변수에 회전행렬을 곱하여 회전한 다음 위치를 얻는다. 이 위치에 다시 회전 중심점이 원 위치로 이동하는 만큼 이동해준다. 그럼 position 값이 카메라의 새로운 position이다. 그래서 CAMERA1.position에 이 position을 넣어준다.

3-4) camera 이동

```

1 function CameraMove(new_posWorld, prev_posWorld) {
2   let lookAt = new THREE.Vector3();
3   //getWorldDirection이라는 아주 친절한 함수로 camera의 lookat을 구한다.
4   CAMERA1.getWorldDirection(lookAt);
5
6   //카메라도 물체와의 거리를 이용해서 움직인다.
7   //world에서 camera의 position과 world에서 object의 position을 통해 벡터를 구함.
8   //camera에서 object로 향하는 벡터
9   let camToObj = new THREE.Vector3();
10  camToObj.subVectors(myMesh.position, CAMERA1.position);
11  //정규화
12  let camToObj_n = camToObj.clone().normalize();
13  //두 벡터로 cos세타 구하기
14  let cos_ = lookAt.dot(camToObj_n); // 둘 다 유닛벡터라서 길이를 딱히 나눠주지 않는다.
15  //벡터 CO * cos_ = CO`
16  let CO_N_length = camToObj.length() * cos_ * 1.0; //float 만들려고 1.0곱함.
17
18  //이제 비율을 구했다!
19  // near : CO 정사영 의 비율로 벡터 differenc를 구하면됨!
20  let diff = new THREE.Vector3();
21  diff.subVectors(new_posWorld, prev_posWorld); // Prev -> New로 가는 벡터
22  //벡터에 스칼라 곱해서 near plane에 비해 얼마나 움직이는지 알아냄.
23  let diff_s = diff.multiplyScalar(CO_N_length);
24
25  const matT = new THREE.Matrix4().makeTranslation(
26    diff_s.x,
27    diff_s.y,
28    diff_s.z
29  );

```

카메라 이동 벡터를 구하는 것은 큐브의 이동과 동일하다.

```

31  //---- Camera matrix 업데이트----
32  CAMERA1.matrixWorldNeedsUpdate = true; // viewMatrix를 위해 업뎃!!
33  CAMERA1.matrixAutoUpdate = false; //내가 지정한 matrix로 셋팅하기 위해서
34  let mat_prev = CAMERA1.matrix.clone();
35  mat_prev.premultiply(matT); // 기존 matrix에 이동행렬을 곱해줌.
36  CAMERA1.matrix = mat_prev;
37
38  //---- Camera의 position property 업데이트 ----
39  CAMERA1.position.set(
40    CAMERA1.position.x + diff_s.x,
41    CAMERA1.position.y + diff_s.y,
42    CAMERA1.position.z + diff_s.z
43  );

```

카메라의 매트릭스와 포지션 속성을 업데이트 한다.

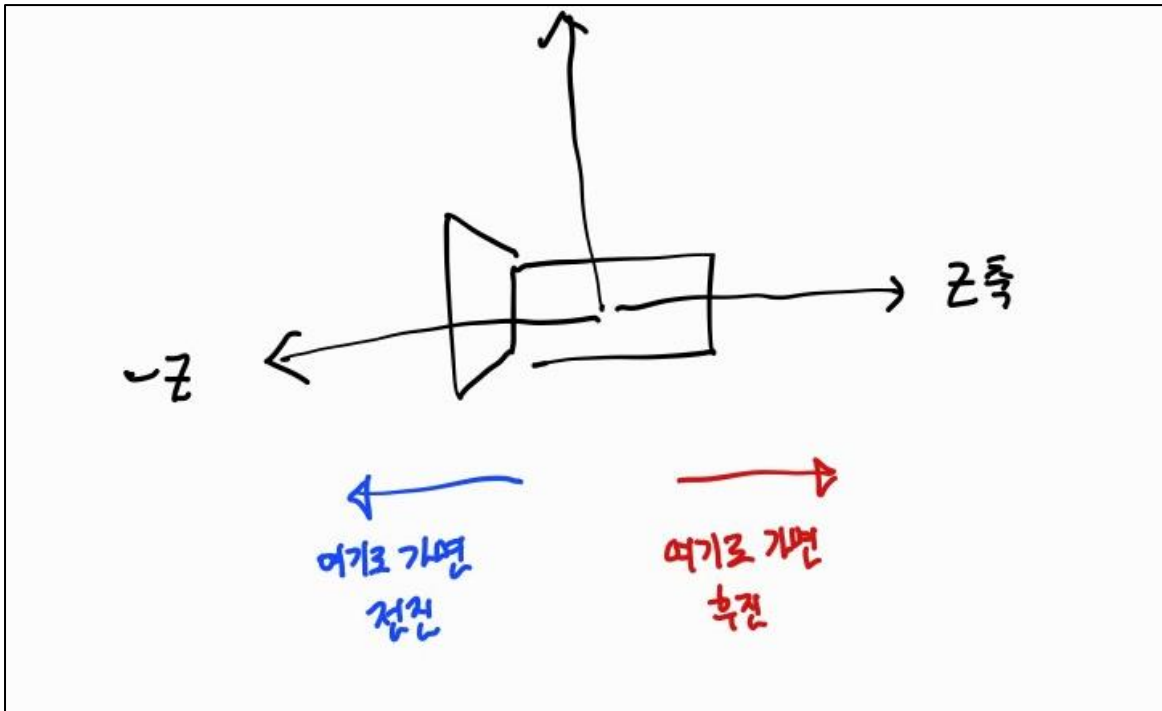
```

45  //카메라 이동에 따라, 카메라 회전축의 중심점도 바꿔줌.
46  camRotCenter.add(diff_s);
47  }

```

카메라의 이동에 따라 카메라 회전 중심점도 이동해준다.

3-5) camera 전진, 후진



```
1 function mouseWheelHandler(e) {  
2   if (mode !== MODE.CAMERA) return;
```

mouseWheelHandler() 함수에 정의되어 있음.

```
4   // 카메라의 -z축을 구하자.  
5   let lookAt = new THREE.Vector3(); // create once and reuse it!  
6   CAMERA1.getWorldDirection(lookAt);  
7   let move = new THREE.Vector3(); // 이동할 벡터를 담는 변수
```

mode가 CAMERA 모드일 때는 일단 카메라의 -z축을 구한다. 카메라의 전진 후진은 카메라의 z축에서 앞 뒤로 왔다 갔다 하는 것이기 때문이다. 친절하게도 `getWorldDirection()` 함수를 통해 카메라의 `lookAt` 벡터를 구할 수 있다. 그리고 `move` 변수에는 카메라를 z축에서 어느 방향, 얼마나 움직일 건지 담을 것이다.

```
9   if (e.deltaY > 0) {  
10    //아래로 휠! 카메라를 뒤로 후진하겠습니다.  
11    //backward move는 카메라의 z축 방향과 평행합니다.  
12    move = lookAt.clone().multiplyScalar(-1);  
13  } else if (e.deltaY < 0) {  
14    //위로 휠! 카메라를 앞으로 전진하겠습니다.  
15    //forward move는 카메라의 -z축 방향과 평행합니다.  
16    move = lookAt.clone();  
17  }
```

`e.deltaY` 변수를 통해 카메라 휠을 얼마나 움직였는지 알 수 있다. `e.deltaY`가 0보다 크면 아래로 휠 한 것이다. 아래로 휠하면 카메라를 후진한다. 이 때에는 카메라 좌표계에서 z축 방향으로 이동할 것이다. `e.deltaY`가 0보다 작으면 위로 휠한다. 이 때에는 카메라 좌표계에서 -z축 방향으로 이동할 것이다.

```

19 // 벡터에 상수배를 하여 움직일 양을 정한다. 여기선 5만큼 했다.
20 move = move.multiplyScalar(5);
21
22 //카메라 매트릭스에 곱해질 회전행렬을 구한다.
23 let matT = new THREE.Matrix4().makeTranslation(move.x, move.y, move.z);
24
25 //camera 회전축의 중심점도 move만큼 변동합니다.
26 camRotCenter.add(move);
27
28 //---- Camera matrix 업데이트----
29 CAMERA1.matrixAutoUpdate = false;
30 CAMERA1.matrixWorldNeedsUpdate = true;
31 let mat_prev = CAMERA1.matrix.clone();
32 mat_prev.premultiply(matT);
33 CAMERA1.matrix.copy(mat_prev);
34
35 //---- Camera의 position property 업데이트 ----
36 CAMERA1.position.x = CAMERA1.position.x + move.x;
37 CAMERA1.position.y = CAMERA1.position.y + move.y;
38 CAMERA1.position.z = CAMERA1.position.z + move.z;
39 }

```

move에 방향을 정했으니 이제 얼마나 갈 것인지 multiplyScalar를 통해 정한다. 여기서는 5만큼 이동할 것이다. 그리고 이 이동을 Camera에 적용시켜야 한다. Camera의 Matrix와 Camera의 position property를 업데이트 한다.

8. 부록

*Object3D position, rotation, scale, matrix 주의

position, rotation, scale, matrix 속성은 오브젝트의 로컬 스페이스 상에서의 위치, 회전, 크기, 로컬 스페이스로의 변환이다. 지금은 카메라나 큐브가 부모가 없어서 월드가 부모가 되었다. 그래서 월드가 로컬스페이스가 되어서 월드상의 위치를 구할 때 문제가 없었다. 하지만 만약에 카메라가 캐릭터에 종속되어서 캐릭터 스페이스가 카메라의 로컬 스페이스가 된다면 카메라 위치를 월드상에서 구할 때 .position으로 구하면 안된다!

* compute_pos_ss2ws 설명

```

1 //screen space to projection space
2 function compute_pos_ss2ws(x_ss, y_ss) {
3   return new THREE.Vector3(
4     (x_ss / RENDER_WIDTH) * 2 - 1,
5     (-y_ss / RENDER_HEIGHT) * 2 + 1,
6     -1 // projection space에서 -1에...
7   ).unproject(CAMERA1); // unproject는 projection space에서 world로 변환해준다.
8 }

```

스크린 x좌표를 RENDER_WIDTH로 나누면 그 값은 0~1로 맵핑된다. projection space는 -1~1인 길이가 2인 곳이

기 때문에 맵핑 된 것을 $\times 2$ 해준다. 그러면 $0 \sim 2$ 다. 이것 다시 $-1 \sim 1$ 로 맵핑해야 하니 -1 을 해준다.

스크린 y좌표에 음수를 곱해준 이유는 스크린에서 y좌표는 아래로 갈수록 커지지만 projection space에서 y는 위로 갈수록 커지기 때문이다. 그래서 반대로 해야하기 때문에 -1 을 곱해준다. 이것도 `RENDER_HEIGHT`로 나누면 $-1 \sim 0$ 으로 맵핑 되고 길이가 2가 되게 2를 곱한다. $-2 \sim 0$ 으로 된 것을 $-1 \sim 1$ 로 맵핑해야하니 $+1$ 을 해준다.