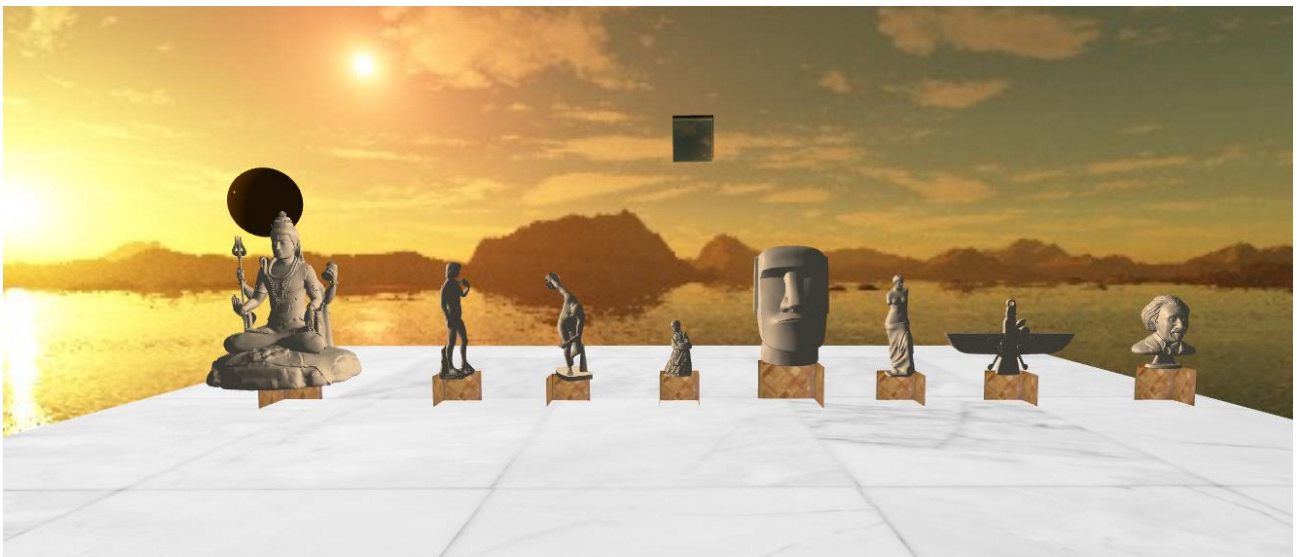
Part 1

```
// Title:Three.js
// Author: Eren Çağlaroğlu,      Deniz Caner Akdeniz
// ID: 11500168978,              14021504096
// Section: 1,3
// Project: 9
// Description:The goal of the homework is to create an
// art gallery museum scene using Three.js.
```

My friends and I completed all of our tasks together via the messaging platform Discord. We found it to be a convenient and efficient way to collaborate and communicate with each other throughout the project.



In our project, we made use of various geometric shapes including cubes, a plane, and a sphere. If you take a closer look, you can spot the sphere in the left back corner, whereas the boxes are positioned beneath the objects.

Setting Scene

1. Create the scene:

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

Driver Link for video: https://drive.google.com/file/d/1W-gD1tYr6pCYPWE6AOoWYd_RIT6ClugQ/view?usp=sharing

- Instantiate a THREE.Scene() object to hold the 3D objects.

2. Create the camera:

```
camera = new THREE.PerspectiveCamera(55, window.innerWidth / window.innerHeight, 0.1, 10000);
camera.position.set(0, 10, 50);
```

- Instantiate a THREE.PerspectiveCamera() with specified field of view, aspect ratio, near and far clipping planes.
- Position the camera using camera.position.set(0, 10, 50).

3. Create the renderer:

```
renderer = new THREE.WebGLRenderer({ antialias: true });
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

- Instantiate a THREE.WebGLRenderer() with antialiasing enabled.
- Set the renderer's size to match the window dimensions.
- Append the renderer's canvas element to the HTML document body.

4. Add camera controls:

```
let controls = new THREE.OrbitControls(camera, renderer.domElement);
controls.minDistance = 500;
controls.maxDistance = 2000;
```

- Instantiate a THREE.OrbitControls() to allow user camera rotation and zooming.
- Set minimum and maximum distances for the camera controls.

5. Create and add ambient light:

```
// Create and add ambient light
let ambientLight = new THREE.AmbientLight(0x7d776a, 0.5);
scene.add(ambientLight);
```

- Instantiate a THREE.AmbientLight() with specified color and intensity.
- Add the ambient light to the scene.

6. Create and add directional light:

```
// Create and add directional light
let directionalLight = new THREE.DirectionalLight(0xf7ce9e, 0.7);
directionalLight.position.set(-1500, 500, 0);
scene.add(directionalLight);
```

- Instantiate a THREE.DirectionalLight() with specified color and intensity.
- Position the directional light using directionalLight.position.set(-1500, 500, 0).
- Add the directional light to the scene.

Driver Link for video: https://drive.google.com/file/d/1W-gD1tYr6pCYPWE6AOoWYd_RIT6ClugQ/view?usp=sharing

SkyBox

1. Create an empty array for materials:

```
let materialArray = [];
```

- **Call let materialArray = [] to store materials for the skybox.**

2. Load skybox textures:

```
let texture_ft = new THREE.TextureLoader().load( 'SKYBOX/sun_ft.jpg');
let texture_bk = new THREE.TextureLoader().load( 'SKYBOX/sun_bk.jpg');
let texture_up = new THREE.TextureLoader().load( 'SKYBOX/sun_up.jpg');
let texture_dn = new THREE.TextureLoader().load( 'SKYBOX/sun_dn.jpg');
let texture_rt = new THREE.TextureLoader().load( 'SKYBOX/sun_rt.jpg');
let texture_lf = new THREE.TextureLoader().load( 'SKYBOX/sun_lf.jpg');
```

- **Load six textures for different faces of the skybox using a THREE.TextureLoader().**

3. Create materials and add to array:

```
materialArray.push(new THREE.MeshBasicMaterial( { map: texture_ft }));
materialArray.push(new THREE.MeshBasicMaterial( { map: texture_bk }));
materialArray.push(new THREE.MeshBasicMaterial( { map: texture_up }));
materialArray.push(new THREE.MeshBasicMaterial( { map: texture_dn }));
materialArray.push(new THREE.MeshBasicMaterial( { map: texture_rt }));
materialArray.push(new THREE.MeshBasicMaterial( { map: texture_lf }));
```

- **For each texture, create a THREE.MeshBasicMaterial with the texture as its map.**

- **Add each material to the materialArray.**

4. Set materials to render on back side:

```
for (let i = 0; i < 6; i++)
    materialArray[i].side = THREE.BackSide;
```

- **Iterate through the materials and set their side property to THREE.BackSide.**

5. Create skybox geometry:

```
let skyboxGeo = new THREE.BoxGeometry( 10000, 10000, 10000);
```

- **Create a large cube using THREE.BoxGeometry() with dimensions of 10000, 10000, 10000.**

6. Create skybox mesh:

```
let skybox = new THREE.Mesh( skyboxGeo, materialArray );
```

- **Create a THREE.Mesh() using the skybox geometry and the material array.**

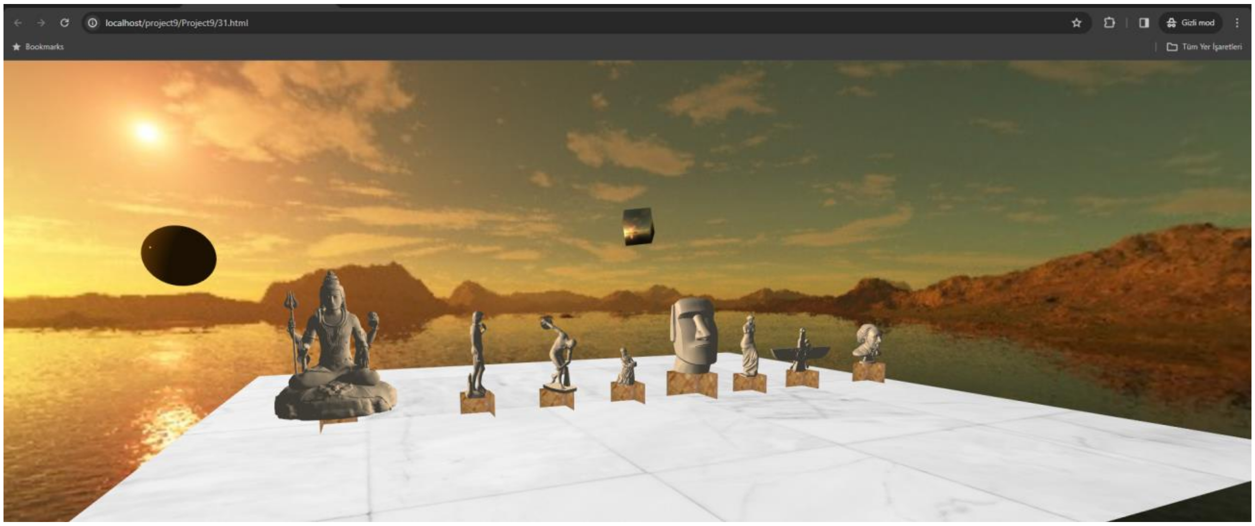
7. Add skybox to scene:

```
scene.add( skybox );
```

- **Call scene.add(skybox) to add the skybox to the scene.**

Driver Link for video: https://drive.google.com/file/d/1W-gD1tYr6pCYPWE6AOoWYd_RIT6ClugQ/view?usp=sharing

Object Loading (.obj)



We have eight objects, each with a box underneath them. Additionally, we have a sphere in the same position as our direct light.

1. Create loaders:

```
        let loader = new THREE.OBJLoader();
        let textureLoader = new THREE.TextureLoader();
```

- Instantiate a THREE.OBJLoader() for loading OBJ files.
- Instantiate a THREE.TextureLoader() for loading textures (not used in this specific code snippet).

2. Load the model:

```
        loader.load('OBJ/statue.obj', function (object) {
            name= 'OBJ/Statue.obj';
            scene.add(object);
            // set position
            object.position.y -= 2;object.position.x -= 2; object.position.z -= 2;
            // set scale
            object.scale.x = 10; object.scale.y = 10; object.scale.z = 10;
        });
```

- Call loader.load('OBJ/statue.obj', function (object) { ... }) to load the model.
- Inside the callback function:
    1. Store model name (optional):
        - Set a variable name to 'OBJ/Statue.obj' for reference.
    2. Add model to scene:
        - Call scene.add(object) to add the loaded model to the scene.

Driver Link for video: https://drive.google.com/file/d/1W-gD1tYr6pCYPWE6AOoWYd_RIT6ClugQ/view?usp=sharing

3. Set model position:

- Adjust object.position.y, object.position.x, and object.position.z to position the model within the scene.

4. Set model scale:

- Set object.scale.x, object.scale.y, and object.scale.z to 10 to scale the model up by a factor of 10 in all directions.

Cube Rendering , Specifying

Cube Rotation:

```
// Rotate the cube
cube8.rotation.x += 0.01;
cube8.rotation.y += 0.01;
```

- cube8.rotation.x += 0.01: Increments the cube's rotation around the X-axis by 0.01 on each frame.
- cube8.rotation.y += 0.01: Increments the cube's rotation around the Y-axis by 0.01 on each frame.

Create cube geometry:

```
let cubeGeometry = new THREE.BoxGeometry();
let cube = new THREE.Mesh(cubeGeometry, materialArray1);
// Set the scale factors along the x, y, and z axes
cube.scale.set(70, 70, 70); // This will scale the cube byx
// Set the position of each cube in a row
cube.position.set(0, -100, 2); // Adjust the spacing between cubes by multiplying
scene.add(cube);
```

- Call THREE.BoxGeometry() to create a cube shape 2. Create cube mesh:
- Call THREE.Mesh() with the cube geometry and a material (assumed to be defined elsewhere) 3. Set cube scale:
- Use cube.scale.set(70, 70, 70) to scale the cube by a factor of 70 in all directions 4. Set cube position:
- Use cube.position.set(0, -100, 2) to position the cube at coordinates (0, -100, 2) 5. Add cube to scene:
- Call scene.add(cube) to add the cube to the scene (assumed to be set up elsewhere)

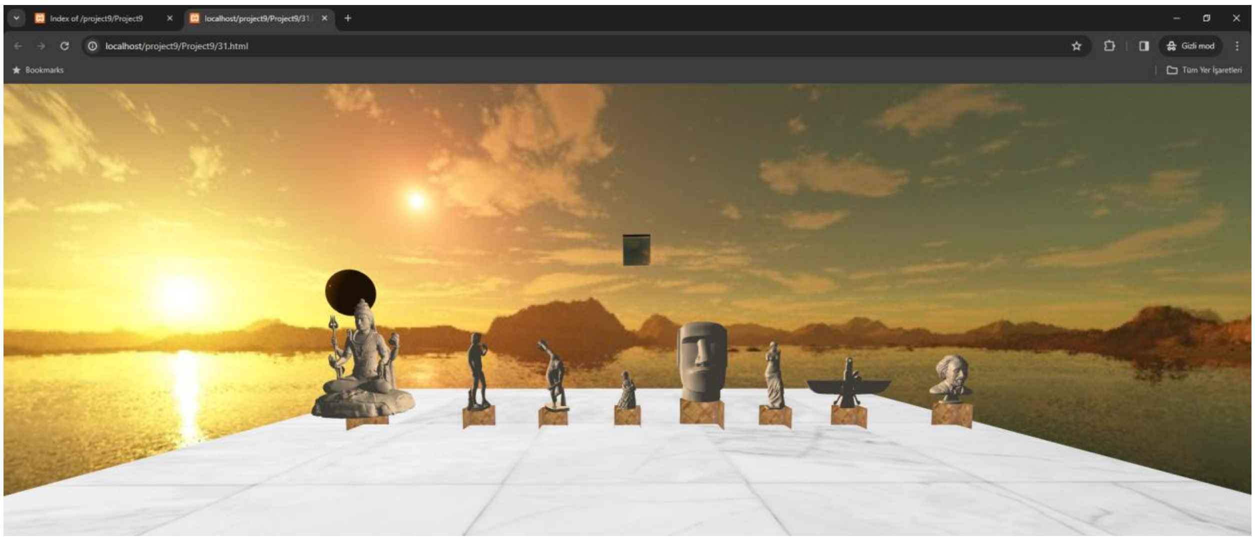
Here we use same wrapping technique of skybox we explained this already in skybox

```
let materialArray1 = [];
let texture_ft1 = new THREE.TextureLoader().load( 'SKYBOX/wood.png');
let texture_bk1 = new THREE.TextureLoader().load( 'SKYBOX/wood.png');
let texture_up1 = new THREE.TextureLoader().load( 'SKYBOX/wood.png');
let texture_dn1 = new THREE.TextureLoader().load( 'SKYBOX/wood.png');
let texture_rt1 = new THREE.TextureLoader().load( 'SKYBOX/wood.png');
let texture_lf1 = new THREE.TextureLoader().load( 'SKYBOX/wood.png');

materialArray1.push(new THREE.MeshBasicMaterial({ map: texture_ft1 +Backside }));
materialArray1.push(new THREE.MeshBasicMaterial({ map: texture_bk1 +Backside }));
```

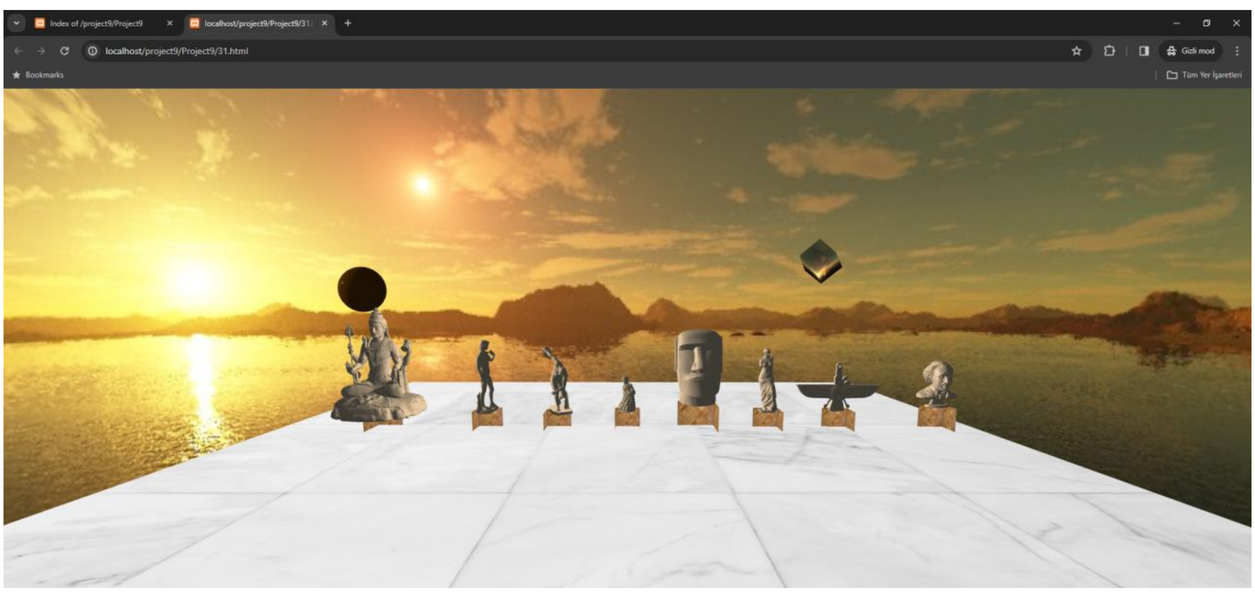
Driver Link for video: https://drive.google.com/file/d/1W-gD1tYr6pCYPWE6AOoWYd_RIT6ClugQ/view?usp=sharing

```
        materialArray1.push(new THREE.MeshBasicMaterial({ map: texture_up1 +Backside }));
        materialArray1.push(new THREE.MeshBasicMaterial({ map: texture_dn1 +Backside }));
        materialArray1.push(new THREE.MeshBasicMaterial({ map: texture_rt1 +Backside }));
        materialArray1.push(new THREE.MeshBasicMaterial({ map: texture_lf1 +Backside }));

        for (let i = 0; i < 6; i++)
            materialArray1[i].side = THREE.BackSide;
```

Animation



After while



Animation Setup:

Driver Link for video: https://drive.google.com/file/d/1W-gD1tYr6pCYPWE6AOoWYd_RIT6ClugQ/view?usp=sharing

```
var direction = 1;
var speed = 0.05; // Adjust speed as needed
var check = true;
```

- direction variable: Stores a value of 1 or -1, controlling the cube's movement direction (left or right).
- speed variable: Sets the movement speed of the cube (0.05 in this case).
- check variable: Acts as a flag to ensure the cube reverses direction only once when reaching a boundary.

Animation Function:

```
requestAnimationFrame(animate);

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

- animate() function:
  - requestAnimationFrame(animate): Creates a continuous animation loop by calling itself recursively.
  - renderer.render(scene, camera): Renders the scene, updating the visuals on each frame.

Cube Movement:

```
cube8.position.x += speed * direction;
```

```
// If the cube has moved too far in either direction, reverse it
if (cube8.position.x >= 200&&check) {
    direction *= -5;
} else if (cube8.position.x <= -200&&!check) {
    direction *= -5;
}
```

- cube8.position.x += speed * direction: Updates the cube's position along the X-axis, creating left-right movement based on speed and direction.

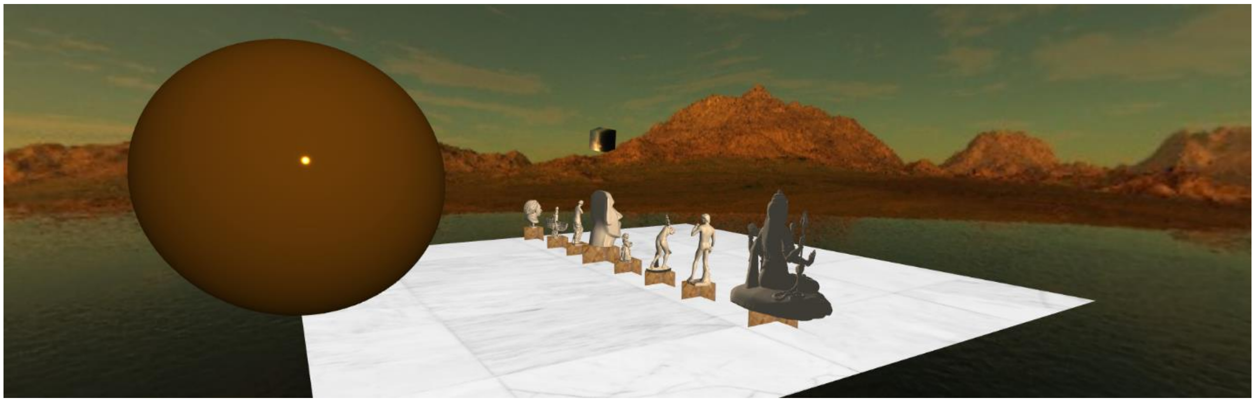
Boundary Check and Reversal:

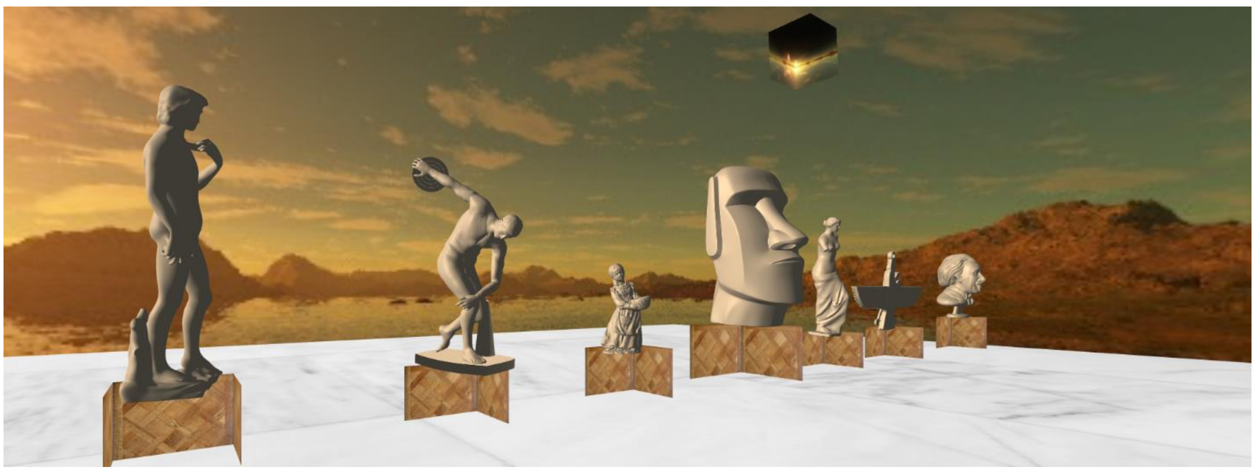
```
if(cube8.position.x >= 200)
    check = false;
if(cube8.position.x <= -200)
    check = true;
```

- if (cube8.position.x >= 200 && check): Checks if the cube has reached the right boundary (200) and if check is true.
  - direction *= -5: Reverses the direction by multiplying it by -5, causing the cube to move left.
  - check = false: Sets check to false to prevent multiple reversals at the same boundary.
- else if (cube8.position.x <= -200 && !check): Similar check for the left boundary (-200), reversing direction if needed and setting check to true.

Driver Link for video: https://drive.google.com/file/d/1W-gD1tYr6pCYPWE6AOoWYd_RIT6ClugQ/view?usp=sharing

Different Angles





Shaders

Vertex Shader:

```
precision highp float;
precision highp int;
```

- Precision: Sets high precision for floats and integers to ensure accurate calculations.

```
uniform mat4 modelMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
attribute vec3 position;
```

- Uniform variables: Declares three matrices for transformations:
    - modelMatrix: Transforms object's local coordinates to world space.
    - modelViewMatrix: Combines model and view matrices for combined transformations.
    - projectionMatrix: Projects 3D scene onto a 2D screen.
- Attribute variable: Declares position to receive vertex positions from the model geometry.

Driver Link for video: https://drive.google.com/file/d/1W-gD1tYr6pCYPWE6AOoWYd_RIT6ClugQ/view?usp=sharing

- Main function:

```
void main() { gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);}
```

1. Calculates the final vertex position by multiplying position with the combined transformation matrices (projection, modelView, and model).

2. Assigns the result to gl_Position, which is used by the GPU for rasterization.

Fragment Shader:

```
precision highp float;
precision highp int;
```

- Precision: Sets high precision for floats and integers.

- Main function:

```
    void main() { gl_FragColor = vec4(0.5, 0.5, 0.5, 1.0);  }
```

1. Sets gl_FragColor (output color of the fragment) to a fixed gray color (0.5, 0.5, 0.5, 1.0). This means all objects using this shader will appear gray.

Key Points:

- Vertex shader transforms and projects vertices.

- Fragment shader determines the color of each pixel.

- These shaders work together to render 3D objects.

Driver Link for video:

https://drive.google.com/file/d/1W-gD1tYr6pCYPWE6AOoWYd_RIT6ClugQ/view?usp=sharing

Driver Link for video: https://drive.google.com/file/d/1W-gD1tYr6pCYPWE6AOoWYd_RIT6ClugQ/view?usp=sharing