Webstrom Three.js   
  
First Code – rotating cube

**Index.html:**

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="utf-8">  
 <title>My first three.js app</title>  
 <style>  
 body { margin: 0; }  
 </style>  
</head>  
<body>  
<script type="importmap">  
 {  
 "imports": {  
 "three": "https://cdn.jsdelivr.net/npm/three@0.167.1/build/three.module.js",  
 "three/addons/": "https://cdn.jsdelivr.net/npm/three@0.167.1/examples/jsm/"  
 }  
 }  
</script>  
  
<script type="module" src="./main.js"></script>  
</body>  
</html>

**Main.js**

import \* as THREE from 'three';  
  
const scene = new THREE.Scene();  
scene.background = new THREE.Color('#F0F0F0');  
  
const camera = new THREE.PerspectiveCamera(75, *window*.innerWidth / *window*.innerHeight, 0.1, 1000);  
camera.position.z = 5;  
  
const geometry = new THREE.BoxGeometry();  
const material = new THREE.MeshLambertMaterial({color: '#468585', emissive: '#468585' });  
  
const cube = new THREE.Mesh(geometry, material);  
scene.add(cube);  
  
const light = new THREE.DirectionalLight(0x9CDBA6, 10);  
light.position.set(1, 1, 1);  
scene.add(light);  
  
const renderer = new THREE.WebGLRenderer();  
renderer.setSize(*window*.innerWidth, *window*.innerHeight);  
*document*.body.appendChild(renderer.domElement);  
  
  
  
function animate() {  
 requestAnimationFrame(animate);  
 cube.rotation.y += 0.01;  
 cube.rotation.z += 0.01;  
  
 renderer.render(scene, camera);  
}  
  
animate();

**Main.js Code Explanation:**

import \* as THREE from 'three';

**// Create the Scene**  
const scene = new THREE.Scene();  
scene.background = new THREE.Color('#F0F0F0');

//creates scene and selects color of the scene

//Add Camera  
const camera = new THREE.PerspectiveCamera(75, *window*.innerWidth / *window*.innerHeight, 0.1, 1000);  
camera.position.z = 5;

/\*

FOV = 75

Aspect Ratio = width/height of window

Nearpoint = 0.1

Farpoint = 1000

\*/

//Create Object (Cube)  
const geometry = new THREE.BoxGeometry();  
const material = new THREE.MeshLambertMaterial({color: '#468585', emissive: '#468585' });

//Create and set color of box

/\*

Mesh Lambert Material: Matte Color that reacts to light, unseen without light, only diffuse and no specular.

\*/

//color: color when light shines on object. Changes depending on brightness, angle

//emissive: no color effect, static.

//final color when lighted upon: color + emissive  
  
const cube = new THREE.Mesh(geometry, material);  
scene.add(cube);

//Create Lighting  
const light = new THREE.DirectionalLight(0x9CDBA6, 10);

/color and intensity  
light.position.set(1, 1, 1);

//lighting angle, at which point the light is shining from  
scene.add(light);

//Add Renderer  
const renderer = new THREE.WebGLRenderer();  
renderer.setSize(*window*.innerWidth, *window*.innerHeight);  
*document*.body.appendChild(renderer.domElement);

//<canvas> created by webGLRenderer = renderer.domElement.

//This must be appended to the DOM in order to show it on screen

function animate() {  
 requestAnimationFrame(animate);  
 cube.rotation.y += 0.01;  
 cube.rotation.z += 0.01;  
  
 renderer.render(scene, camera);  
}  
  
animate();

**Result:**

****

**DOM:**

Document Object Model

Browser changes HTML file into **TREE** structure to control it as JS.

* Each HTML Tag – Each node in Tree
* HTML is a static file whereas DOM is dynamic
* webGLRenderer creates <canvas> HTML tag
* appendChild(renderer.domElement) will append that canvas onto the DOM tree thus having it appear on browser screen.

**Shapes and Geometry:**

1. **Box Geometry:**

* Creates a cube or rectangular prism depending on dimensions
* const geometry = new THREE.BoxGeometry( 1, 1, 1 );

param(height, depth, width)

* const cube = new THREE.Mesh( geometry, material);
* material may be **basic** or **mesh lambert**

1. **Cylinder Geometry**

* Creates prisms and cylinders
* const geometry = new THREE.CylinderGeometry(

radiusTop, radiusBottom, height, radialSegments, heightSegments, openEnded);

* radiusTop, radiusBottom: **radius** of top and bottom faces. Equal for prisms
* radial Segments: **shape** of prism, 3 for triangular 8 for octagonal etc.
* To generate a cylinder, set radialSegments = 32 – 64, turn on **antialiasing: smoothening the edges**

const renderer = new THREE.WebGLRenderer({ antialias: true });

1. **Line Geometry (Edges)**

* Single width lines, could be used as edge of geometry object
* const edges = new THREE.EdgesGeometry( geometry )
* const line = new THREE.LineSegments(edges, new THREE.LineBasicMaterial( { color: ‘#273747’ } ) );
* scene.add( line );
* Create **edges** then create **line** using the edges.
* To move it along with the geometry, must add the lines’ rotation:

Line.rotation.y += 0.01

1. **More Geometries**

* Circle:

const geometry = new THREE.CircleGeometry( radius, radial segments (32 default));

* Cone:

const geometry = new THREE.ConeGeometry( radius, height, radial segments);

* Ring

const geometry = new THREE.RingGeometry( inner rad, outer rad, radial segments);

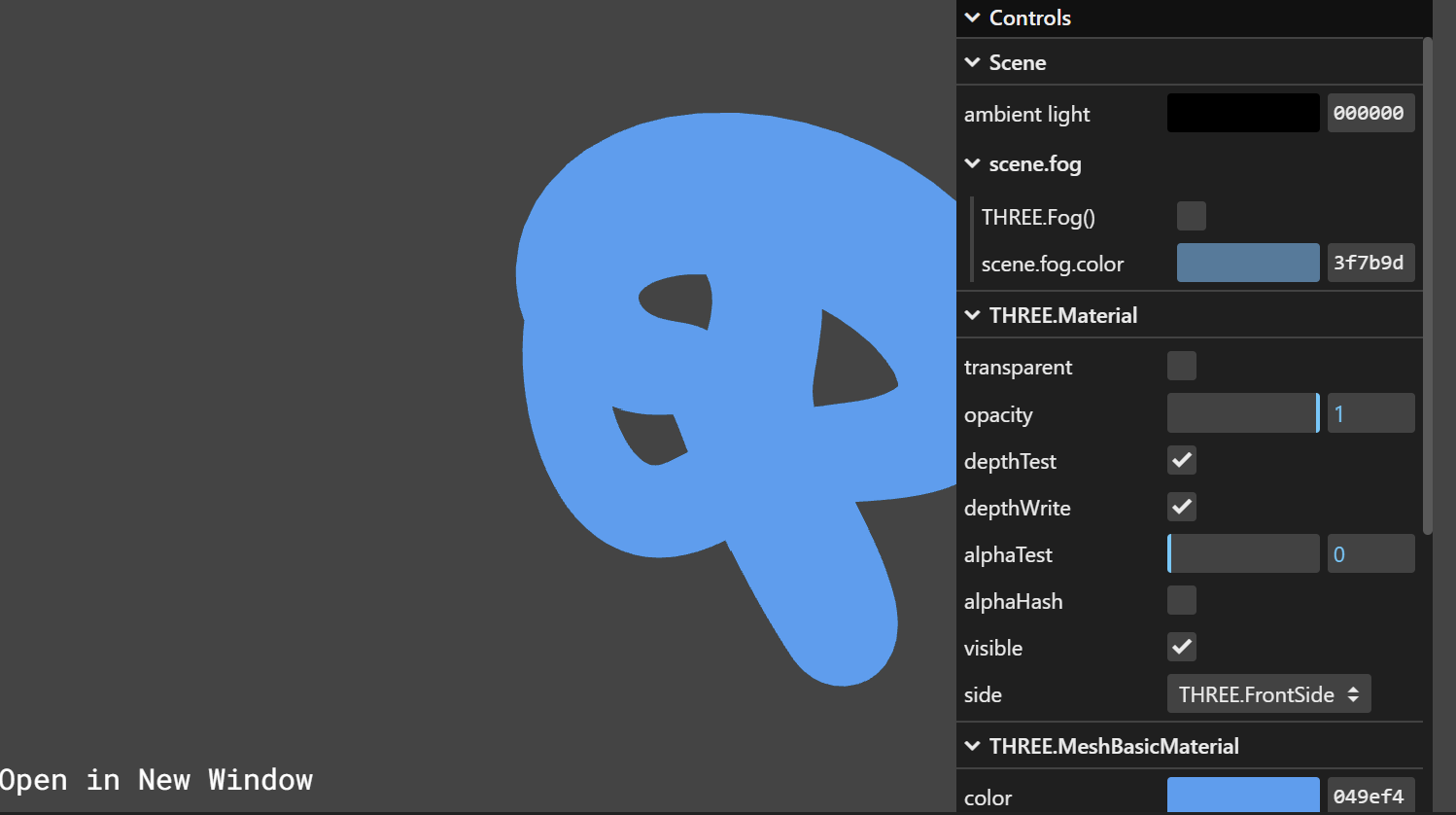
**Materials:**

1. **Material()**

* Material() to create a generic material
* Renderer Independent -> no need to rewrite for each renderer

1. **MeshBasicMaterial**

* Unaffected by lights

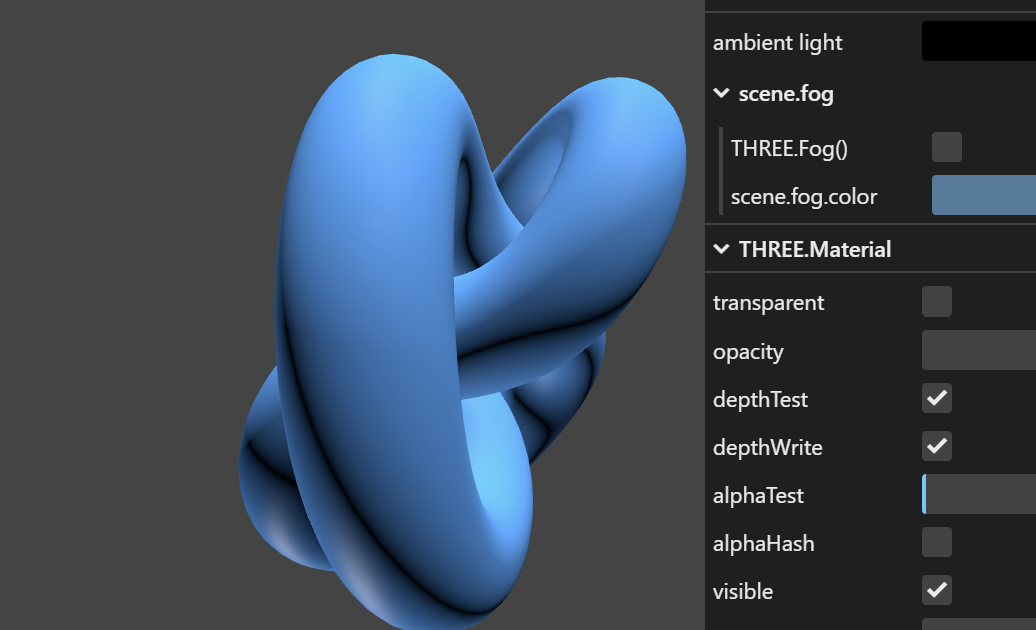
****

* .color: the color of this material

1. **MeshLambertMaterial**

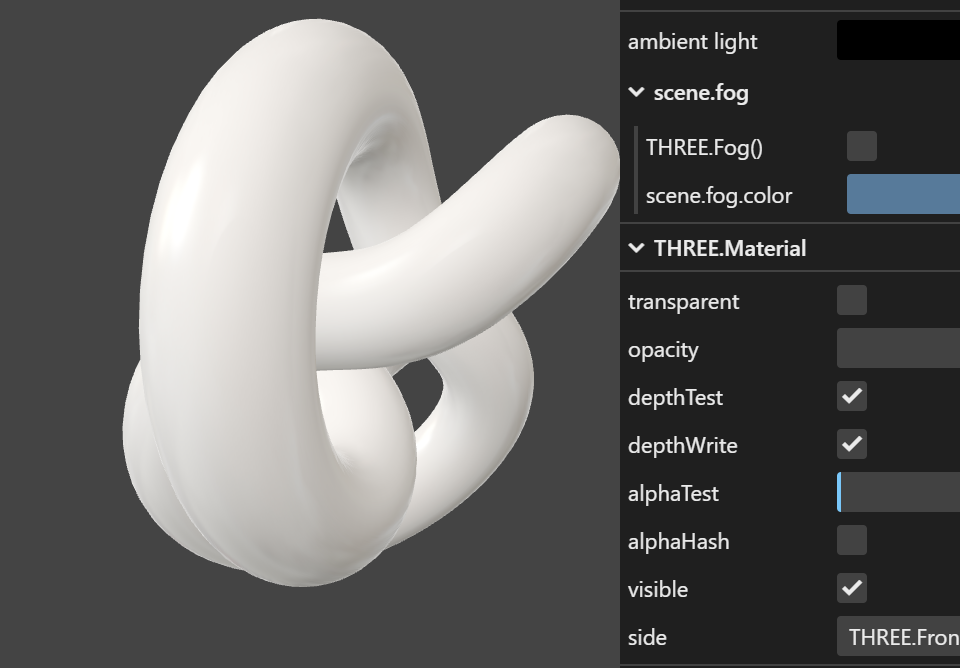
* Non-shiny obj
* Reacts to light but shows no reflection
* Emmisive exists

1. **MeshPhongMaterial**

****

* Shiny surfaces with specular highlights

1. **MeshMatCap:**



* No reaction to lights, no shadows but does cast shadows

1. **PointsMaterial:**

* Draws a point

const vertices = []; for ( let i = 0; i < 10000; i ++ ) { const x = THREE.MathUtils.randFloatSpread( 2000 ); const y = THREE.MathUtils.randFloatSpread( 2000 ); const z = THREE.MathUtils.randFloatSpread( 2000 ); vertices.push( x, y, z ); } const geometry = new THREE.BufferGeometry(); geometry.setAttribute( 'position', new THREE.Float32BufferAttribute( vertices, 3 ) ); const material = new THREE.PointsMaterial( { color: 0x888888 } ); const points = new THREE.Points( geometry, material ); scene.add( points );

1. **Common Properties:**

* Color: default color of the material
* Emmisive: a solid color that appears when shone upon
* Alpha Map: grayscale texture that controls the opacity of surface. White: not transparent, Black: fully transparent
* Side: which sides are shown. Frontside, Backside, Doubleside
* Normal Map: Tricks the normal vectors using RGB values so that the surface “seems” 3D. Low cost 3d effect

R: X axis, G: Y axis, B: Z axis creates shadow highlights

const normalMap = textureLoader.load('./textures/character\_normal.png');

**\*\***Only applies on non-matte objs

* Bump map: Simple normal map only using grayscale. If normal map is defined, it will be ignored
* Combine: Integer that defines how the surface color is mixed with the env map.
* Env map: map that reflects the environment like a mirror
* Emissive map: Uses image to set the glow pattern. Never set emissive color as black when using emissive map

**Lights**

1. **Directional Light:**

* Shines to set direction, behave as though it has rays coming from an infinitely far point.

const directionalLight = new THREE.DirectionalLight( color, intensity); scene.add( directionalLight );

1. **Ambient Light:**

* No direction, entire scene lighted

const light = new THREE.AmbientLight( 0x404040 ); // soft white light scene.add( light );

1. **Point Light:**

* Light that is emitted from a single point and points to all directions

const light = new THREE.PointLight( 0xff0000, 1, 100 ); light.position.set( 50, 50, 50 ); scene.add( light );

1. **RectAreaLight:**

* Emits light uniformly across the face of a rectangular surface

const rectLight = new THREE.RectAreaLight( 0xffffff, intensity, width, height ); rectLight.position.set( 5, 5, 0 ); rectLight.lookAt( 0, 0, 0 ); scene.add( rectLight )

1. **Spotlight:**

* Emits a cone shape light from a single point to another circular point. Increases in size as object gets farther from light

1. **Hemispherical Light:**

* Light that is positioned directly above the scene, with color fading from the sky color to the ground color. Can not cast shadows
* const light = new THREE.HemisphereLight( sky color, ground color, intensity ); scene.add( light );

1. **Properties, methods:**

* .copy(source light): Copies the value of color and intensity from source light to this light
* isLight(): Boolean to check if it is of type light

**Shadows:**

1. **Properties :**

* .camera: The light's view of the world. This is used to generate a depth map of the scene; objects behind other objects from the light's perspective will be in shadow.
* .intensity: Intensity of the shadow. Default is 1, values range from 0 to 1
* .map: Auto gen. WebGLRenderTarget. Used mostly for debugging
* .mapSize: pair of number that determine the quality of the shadow. Numbers powers of 2, (512, 512) (512 1024)

\*\* Important to first set a small camera frustum before increasing shadow quality to actually increase quality.

light.shadow.camera.left = -5; light.shadow.camera.right = 5;

light.shadow.camera.top = 5; light.shadow.camera.bottom = -5;

* .radius: Setting this to something greater than 1 will blur the edges of the shadow
* .bias: increase to fix acnes (dots or lines created by light shadow) on surfaces
* .normalBias = increase to fix acnes on tilted surfaces (surfaces that receive light at an angle)

\*\*Acnes happen frequently for double sided objects or very thin objects

1. **Methods:**

* .dispose(): Deallocate GPU memory regarding shadow. Used when toggling shadows. Does not actually get rid of shadows in the next frame. That is done by light.castShadow = false; or renderer.shadowMap.enabled = false;
* .copy(source): Copy shadow settings (bias/normalbias/mapSize/camera near, far) for multiple lights.

spot2.shadow.copy(spot1.shadow);

1. **PointLightShadow**

* Not called directly, created when light.castShadow = true for a point light

1. **DirectionalLightShadow:**

* Not called directly, created when light.castShadow = true for a directional light. Orthographic Camera used to view shadow map

1. **SpotlightShadow:**

* Not called directly, created when light.castShadow = true for a spotlight. Uses perspective camera

**Lights and Shadows Helpers**

1. **Import:**

Import three/addons/helpers/\*

1. **DirectionalLightHelper**

* Visualizes the DirectionalLight’s effect on the scene using plane and a line representing the light’s position and direction   
  const helper = new THREE.DirectionalLightHelper( light, plane size, color ); scene.add( helper );
* Must use helper.update() after the position or targer of the directional light has been modified.

1. **SpotLightHelper**

* Visualizes the spotlight’s effect on the scene using a cone.

const spotLightHelper = new THREE.SpotLightHelper( spotLight ); scene.add( spotLightHelper );

* Must use helper.update() if a feature of the spotlight is modified

1. **PointLightHelper**

* Visualizes the spotlight’s effect on the scene using a sphere.

const pointHelper = new PointLightHelper(point, sphere size);

scene.add(pointHelper);

* Must use helper.update() if color is modified

1. **HemisphereLightHelper**

* Visualizes the hemispherical light’s effect on the scene using a spherical mesh that consists of both sky and ground color

const helper = new THREE.HemisphereLightHelper( light, size ); scene.add( helper );

1. **Camera Helper**

* Visualizes what a camera contains in its frustum, what the shadow camera (internal) covers. Check to see if an object is within the frustum of the camera or not. Decrease frustum size and updateProjectionMatrix() -> helper.update()

const helper = new THREE.CameraHelper( camera ); scene.add( helper );

**Renderer:**

1. **Properties:**

* Antialias: On to reduce “stair effects” where a single line that is meant to be circular or curved is broken to lines that look like stairs
* Canvas: Set particular canvas
* setSize: Match the size of the display monitor/screen

1. **Color Related Options**

* Renderer.setClearColor(): Background color by default

1. **Shadow and Light Related:**

* Renderer.shadowmap.enabled = true: Shadow on for scene
* Renderer.shadowMap.type = THREE.PCFSoftShadowMap: Shadow quality control, most frequently used
* physicallyCorrectLights = true: apply physical angles and distances to display shadows (receive and cast)

1. **Code snippet: resize according to window**

window.addEventListener('resize', () => {

camera.aspect = window.innerWidth / window.innerHeight;

camera.updateProjectionMatrix();

renderer.setSize(window.innerWidth, window.innerHeight);

});j

**Camera:**

1. **ArrayCamera:**

* A array that is used to store predefined cameras to render a scene
* Used for VR

1. **Camera():**

* Not called directly
* isCamera: Boolean – straight forward
* .layers: layer that the camera is a part of

1. **Methods:**

* .clone(): Return a new camera of the same properties
* .copy(source): Copy properties of camera 1 to camera 2