Real time rendering of shadows need to be around 60 times per second.

Ray rendering is the slow way to do it.

Ray tracing in blender is what can take hours to compile

Three.js has a built in solution for shadows. Renders for each light supporting shadows

Renders simulate what the light would see if it was a camera. During a render those materials will be replaced by MeshDepthMaterial, this will enable depth information. You need to get the depth of the material

The renders are then stored as textures and we can use the shadow maps. Projected onto the geometry.

Once threejs has the shadow maps, it will use them to color our objects and meshes

Shadow maps are done automatically and we don’t need to understand them.

To activate shadows first you have to check your renderer

So:

renderer.shadowMap.enabled = true;

now you need to ask, can my object receive and cast shadows?

Does my plane receive and cast shadows?

sphere.castShadow = true;

plane.receiveShadow = true;

in this example we have a sphere above a plane. So the sphere can cast a shadow and the plane can receive a shadow.

Only three lights support shadows:

PointLight

DirectionalLight

SpotLight

Now we need to change the shadowmap width and height render. Base is 512 x 512

directionalLight.shadow.mapSize.width = 1024;

directionalLight.shadow.mapSize.height = 1024;

can access the property of the width and height b/c shadow is a Vector2.

Can make the shadow even more crisp by increasing resolution of the shadow map by:

directionalLight.shadow.mapSize.width = 1024 \* 3;

directionalLight.shadow.mapSize.height = 1024 \* 3;

keep in mind you have to use power of 2 b/c of Mip mapping

b/c we are doing a render we also can control the near and the far. If we make it fit our scene, it won’t save in pref as much or the result. It is mostly about precision.

How do we access shadow light camera?

const directionalLightCameraHelper = new THREE.CameraHelper(

  directionalLight.shadow.camera

);

scene.add(directionalLightCameraHelper);

this allows us to really see what our camera is capturing for the shadowing and customize the near and the far.

directionalLight.shadow.camera.near = 1;

directionalLight.shadow.camera.far = 6;

this will help the camera only capture the range that you need and avoid bug or glitches on the picture.

Next thing is about the amplitude. You do not need to render the entire scene, it is way too much.

B/c it is an orthographic camera, we have control over the left, right, top and bottom and near and far.

directionalLight.shadow.camera.top = 2;

directionalLight.shadow.camera.right = 2;

directionalLight.shadow.camera.bottom = -2;

directionalLight.shadow.camera.left = -2;

this helps us grab a small camera with a higher resolution making the edges even crisper.

We can also hide the helper if we want to keep but not use it

directionalLightCameraHelper.visible = false;

We can also control the blur of our shadow.

directionalLight.shadow.radius = 15;

this blurs a shadow the higher the value.

Shadowmap algorithm that we can change

The current one is the PC shadow map.

There is a Basic shadow map – better pref but lose qualities

* **THREE.BasicShadowMap:**Very performant but lousy quality
* **THREE.PCFShadowMap:**Less performant but smoother edges
* **THREE.PCFSoftShadowMap:**Less performant but even softer edges
* **THREE.VSMShadowMap:**Less performant, more constraints, can have unexpected results

To change it, update the **renderer.shadowMap.type** property. The default is **THREE.PCFShadowMap** but you can use **THREE.PCFSoftShadowMap** for better quality.

renderer.shadowMap.type = THREE.PCFSoftShadowMap;

change the type on the renderer

PCFSoftShadowMap is what Bruno usually uses to improve perf. Also, radius does not work with this type of shadow map.

Usually if you want a blury shadow you will play with the resolution

Add in a spotlight

const spotLight = new THREE.SpotLight(0xffffff, 0.4, 10, Math.PI \* 0.3);

spotLight.castShadow = true;

spotLight.position.set(0, 2, 2);

scene.add(spotLight);

add camera helper for this

const spotLightCameraHelper = new THREE.CameraHelper(spotLight.shadow.camera);

scene.add(spotLightCameraHelper);

spotLightCameraHelper.visible = true;

PointLight gives off light in every direction so the camera takes renders of every plane in every direction using a perspective camera. So the camera helper will only show one direction. It is a ton of renders. Basically 6 times the renders per pointLight

Baking Shadows nowwwwwwwwwwww – we basically bake the shadow into the textures.

const plane = new THREE.Mesh(

  new THREE.PlaneGeometry(5, 5),

  new THREE.MeshBasicMaterial({ map: bakeShadow })

);

So in the material, we pass in the bakeShadow as a map property in the object. And this produces a baked in material with a shadow.

Blender can bake shadows and provide materials or textures for this.

Baking shadows alternative, ones that can move.

You can map a material underneath and increase/decrease the alpha depending on how high or low an object is, and then also move it if the object is moved in order to replicate a shadow effect object.

So you put the shadow on a plane slightly above the floor and slightly below the object.

const sphereShadow = new THREE.Mesh(

  new THREE.PlaneBufferGeometry(1.5, 1.5),

  new THREE.MeshBasicMaterial({

    color: 0xff0000,

    transparent: true,

    alphaMap: simpleShadow,

  })

);

sphereShadow.rotation.x = -Math.PI \* 0.5;

sphereShadow.position.y = plane.position.y + 0.01;

scene.add(sphereShadow);

here is how this looks if we have a texure loaded into simpleShadow.

When we use an alphamap, we have to have transparent set to true.

We also need the shadowmap plane to be slightly above our normal plane.

To turn the shadow black just change 0xff0000 to 0x000000

Now lets update the sphere and move it to try and get the shadow to follow it around and change with height

In our tick function we add a update sphere function:

  sphere.position.x = Math.cos(elapsedTime) \* 1.5;

  sphere.position.z = Math.sin(elapsedTime) \* 1.5;

  sphere.position.y = Math.abs(Math.cos(elapsedTime \* 3));

this makes the ball go around in a circle and bounce almost.

Now we need to update the shadow to follow it.

 //update Shadow

  sphereShadow.position.x = sphere.position.x;

  sphereShadow.position.z = sphere.position.z;

this makes the shadow follow the ball around.

Now we need to play with opacity to make the shadow seem to go in and out as the ball goes up and down.

 sphereShadow.material.opacity = (1 - sphere.position.y) \* 0.5;

here is what he did. I tried to use natural log and it was not linear. Did look nice though.

Shadow handling is really up to need.