Camera in three.js is an abstract class – do not use it directly.

ArrayCamera is use to render the scene with different POVs

StereoCamera is used to render the same through two cameras. Used to mimic the eyes. Create a depth effect for special devices like VR.

CubeCamera – does 6 renders. Forward, backwards, left, right, top and bottom. Creates render of the surroundings. Three.js uses this to make environment maps, shadows, reflections. Etc

OrthoGraphicCamera – creates render of scene without perspective. Basically ignores how far things are and just renders all items like they are really close to camera.

PerspectiveCamera – what we’ve been using, but had additional parameters.

We are going to play with perspectivecamera and orthographiccamera.

Parameters of perspective camera:

First is POV field of view and vertical degrees.

Making field of view larger and smaller can distort things and make them look or seem bigger. It is also due to the scene being a rectangle.

Basically, the more you can “see” the more that will be squeezed into that window.

Typically, 45-75 is good, but 75 is a lot still. Need to decide this at beginning of project.

The second parameter is the aspect ratio which is basically the width of the render divided by the height of the render. You need to provide this because it could change in weird situations.

The other two parameters are near and far. Basically tells you what range you can see. Anything closer than 1 or further than 1000 will not show up.

Anything between the visible range will only show part if not in range.

So if we find the position length with

const camera = new THREE.PerspectiveCamera(

  75,

  sizes.width / sizes.height,

  1,

  3.4641016151377544

);

camera.position.x = 2;

camera.position.y = 2;

camera.position.z = 2;

console.log(camera.position.length());

and then pass that console.log into the “far” parameter (which we already did above) we will only see part of the cube as it spins.

But do not make the near and far values extremely small or large. Something like .0001 and 100000 because then you can have “z” fighting.

Basically, if there are two objects right next to each other your GPU will have trouble figuring out which one is in front of the other because your range is so crazy. So, keep your range something sensible like 1-1000 I think. Ultimately it depends on your project.

Bruno is using 0.1 and 100. But he says 100 might be too much ☹

Now OrthographicCamera time

The difference is that that orthographiccamera will lack perspective. Basically, objects have the same size regardless of their distance to the camera.

First parameter, we need to provide what how far the camera can see in each direction

Left, right, top and bottom. Then we provide the near and far.

Simple terms, it renders a box of a scene. And you tell it how big you want the box to be. Imagine taking a shoebox and only seeing what is inside of it. But inside is what you decide to add from your scene.

const camera = new THREE.OrthographicCamera(-1, 1, 1, -1, 0.1, 100);

camera.position.x = 2;

camera.position.y = 2;

camera.position.z = 2;

although, our cube will look a bit squished because we are rendering a square from the camera, but our render is not a square because of our selected sizes

const sizes = {

  width: 800,

  height: 600,

};

This makes the cude seem a bit squished, so we have to add the aspect ratio to the render.

So now we add an aspect ratio to our camera like so:

const aspectRatio = sizes.width / sizes.height;

const camera = new THREE.OrthographicCamera(-1 \* aspectRatio, 1 \* aspectRatio, 1, -1, 0.1, 100);

this helps our render not become distorted.

We want to add an event listener to use our mouse to change the camera.

window.addEventListener("mousemove", (event) => {

  console.log(event.clientX);

});

Now we want to store this value but we don’t want to use pixel values because screen sizes can change right?

So we are going to start with a stored screen value.

const cursor = {

  x: 0,

  y: 0,

};

window.addEventListener("mousemove", (event) => {

  cursor.x = event.clientX / sizes.width;

  console.log(event.clientX);

});

Later sizes will have actual viewport size

Now fix the variable with a -.5 for the camera ideally

  x: 0,

  y: 0,

};

window.addEventListener("mousemove", (event) => {

  cursor.x = event.clientX / sizes.width - 0.5;

  console.log(cursor.x);

});

Now tie it all together and add

  camera.position.x = cursor.x \* 3;

  camera.position.y = cursor.y \* 3;

  //   camera.lookAt(mesh);

To the tick function used to show the animation.

One odd thing is the y DOM and the animation y are going opposite, so to make everything inverse like the x value we add:

  cursor.y = -(event.clientY / sizes.height - 0.5);

Now make us look at the cube by adding:

  camera.lookAt(new THREE.Vector3());

Now lets make the cube do a full revolution when we move the mouse across the x axes

  camera.position.x = Math.sin(cursor.x \* Math.PI \* 2) \* 3;

  camera.position.z = Math.cos(cursor.x \* Math.PI \* 2) \* 3;

  camera.position.y = cursor.y \* 5;

  camera.lookAt(mesh.position);

and be able to see the top and bottom of the cube.

Now we can stop and use three.js built in controls.

We are going to play around with OrbitControls

import { OrbitContols } from "three/examples/jsm/controls/OrbitControls.js";

now we want to add this after the camera.

const controls = new OrbitContols(camera, canvas);

need to provide it the camera and the DOM element we want to it focus on

controls.enableDamping = true;

lets add damping so that the movement is not so rigid.

But we also need to update the controls on each frame in order for them to control to work without the mouse moving.