Ben Coleman Follow
Sep 18, 2017 · 8 min read · Listen

Save

The Beginner's Guide to Beginning Three.js

Let's start at the beginning.

What is Three.js?

Three.js is a Javascript library for creating 3D web graphics within the web browser created by Ricardo Cabello using WebGL.

We started too fast, so…

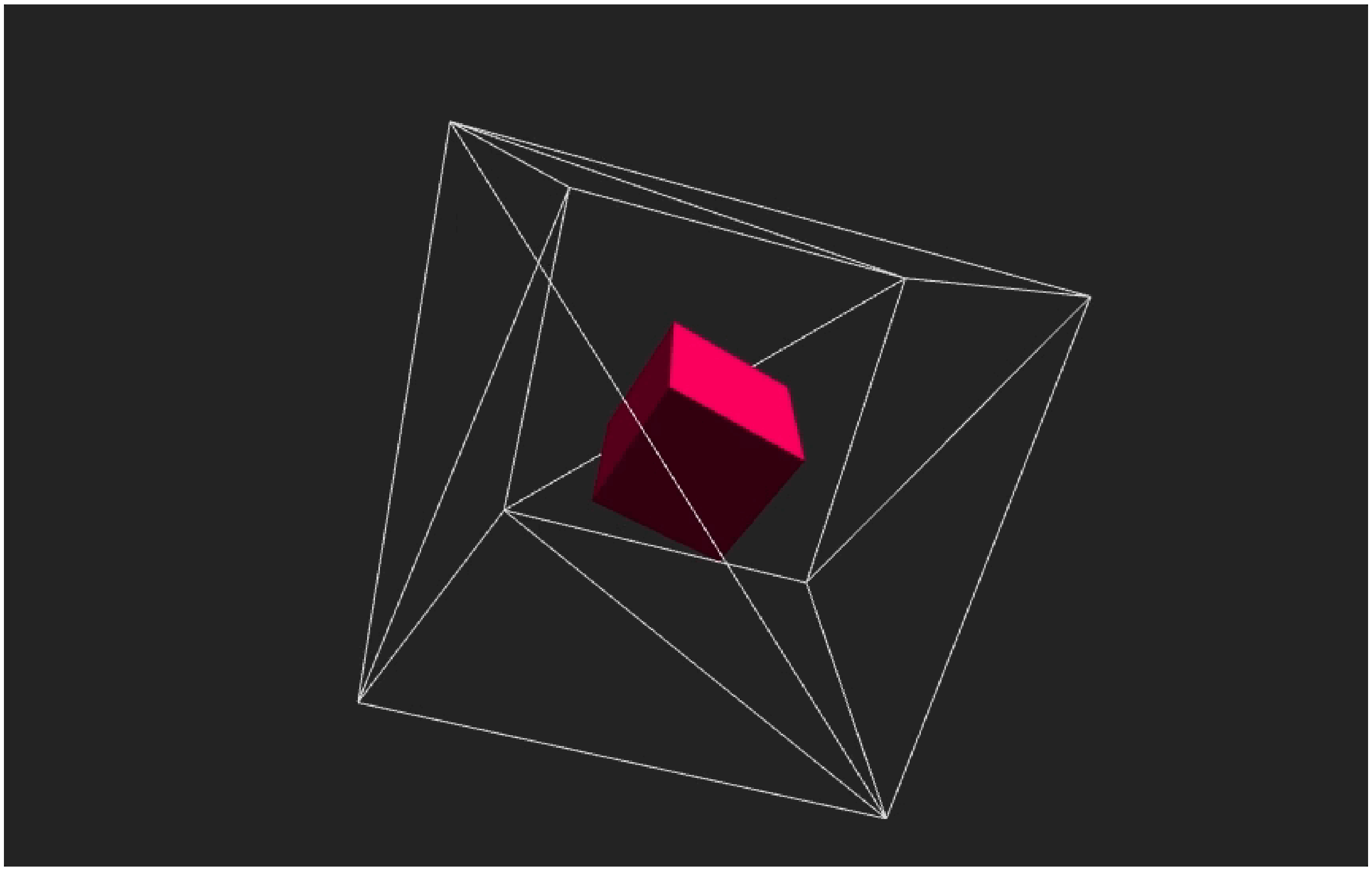
What is WebGL?

WebGL is a Javascript API that allows the browser to render 3D graphics without the use of any plug-ins. WebGL allows for GPU accelerated rendering within an HTML5 canvas element on a webpage. The first stable release 1.0, was released in March of 2011 and now version 2.0 has been released in January of 2017.

Our Goals

- Get started with the Three.js library
- Create our first scene
- Put something in the scene
- Make it move
- Give the scene some light

Ultimately we will complete our goals and this lovely render to show for it.



Setup

This is the beginners guide so we're starting simple. We're only going to use two files

- index.html
- index.js (or whatever you want to name it)

Starting with the HTML file, we really just need to make a basic HTML page and import the Three.js library. For the purpose of simplicity we'll use a CDN so we don't need to download and reference the full library, but if you're hardcore feel free to download. If you're using a Webpack/Babel setup and are importing from a package be sure to import * as THREE in order to keep with the tutorial. The only CSS we'll use for the example is inline to normalize the margin to 0 and to display our canvas on the entire page.

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset=utf-8>
                <meta name="viewport" content="width=device-width, user-scalable=no, minim
                <title>Beginning my Three.js Journey</title>
                <style>
                        body { margin: 0; }
                        canvas { width: 100%; height: 100% }
                </style>
        </head>
        <body>
    <!-- CDN Link to Three.js -->
                <script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/87/three.js">
                <!--reference your JS file here. Mine looks like below-->
    <script src="./index.js"></script>
        </body>
</html>
```

index.html hosted with ❤ by GitHub view raw

Now let's setup our JS file. Anytime we're using Three.js, the fundamentals of our setup are the **scene**, **camera**, and **renderer**. These will be important throughout, so remember this. Think of it like we're making a movie. We need something to shoot our movie (camera), we need somewhere to shoot it (scene), and then we need a movie theater to put it on the screen (renderer).

```
const scene = new THREE.Scene()
const camera = new THREE.PerspectiveCamera( 75, window.innerWidth /
window.innerHeight, 0.1, 1000 )
const renderer = new THREE.WebGLRenderer({ antialias: true})
```

We start with a scene, we'll be adding to that later, but for now we'll create a new scene element with no arguments.

Our camera we'll use the 'PerspectiveCamera' option. There are four default cameras to be used but the PerspectiveCamera is a good starting point that mimics the human eye. We'll need to give this function four arguments:

- Field of View — How wide the camera perspective will be.
- Aspect Ratio — Think like a TV screen. If we do width / height we'll be fine in most all situations. Everything else will squish our scene.
- Near clipping plane — Elements closer than this won't be render on the screen. If we were shooting our movie anything behind the camera wouldn't show up. We're basically using 0 for this.
- Far clipping plane — Elements further away than this won't render. Too high of a value might cause performance issues.

Our renderer will take an object of options, we're only going to use {antialiasing: true} in order to have smoother edges on our cube to be created. Entirely unnecessary but something that will look a little nicer.

Next we need to tell our movie theater where to go, and how big it should be.

```
renderer.setSize( window.innerWidth, window.innerHeight )
document.body.appendChild( renderer.domElement )
```

We set the size of the renderer to the size of the entire window, and we tell it to append itself to the body of our HTML page.

Let's see our beautiful work.

Okay so it might not be amazing yet, but the stage is set.

Next is creating our cube. Just like how we need a scene, camera, and renderer to get started, we need three core elements to every object we want to render:

- **Geometry** — The shape of our object, made up of vertices (points), and face (think the plane of an object). Three also gives us lots of premade geometries where we'll only need to give it some arguments and it'll handle the details from there. We'll be using BoxGeometry, but be sure to check out the docs.
- **Material** — What our object is made of. This will include things like the color and it'll determine how light interacts with our object. If we had a pebble and a metal ball bearing, they would have a similar geometry, but their material is very different and it would change how it looks to the viewer.
- **Mesh** — The combination of geometry and material.

So now for our cube:

```
var geometry = new THREE.BoxGeometry( 1, 1, 1)
var material = new THREE.MeshBasicMaterial( { color: 0xff0051 })
var cube = new THREE.Mesh ( geometry, material )
scene.add( cube )
renderer.render( scene, camera )

camera.position.z = 5
```

Our BoxGeometry method will take in constructor arguments that are the width, height, and depth of the box respectively.

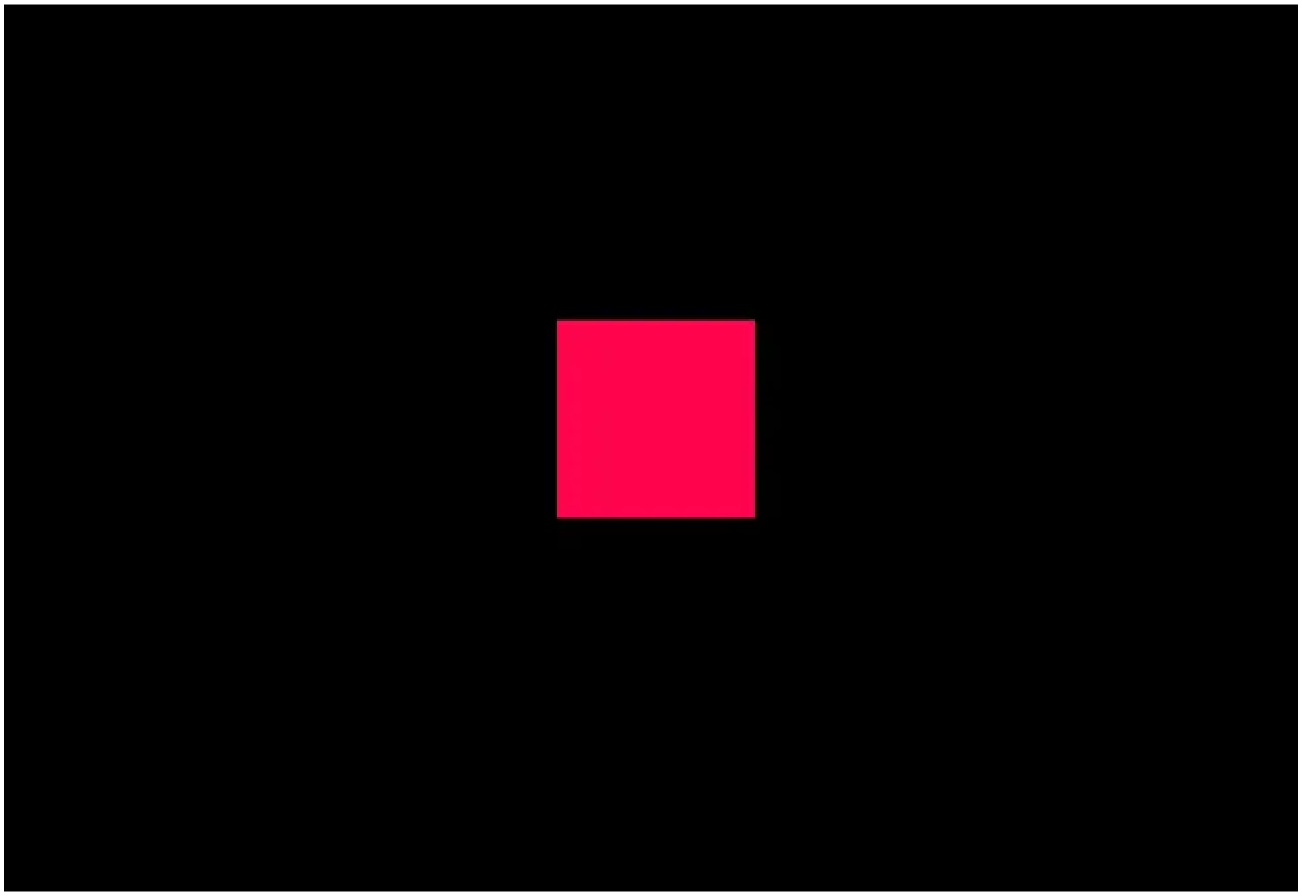
For the material we'll be using the most basic provided material that is not affected by light sources (we'll cover later), and we give it a color option. If you're not cool with writing the hex color with the 0x before the standard hex, feel free to write it as color: "#ff0051" (including the quotation marks), but the standard is the 0xff0051 to make our cube red.

Next we combine the geometry and material variables we just created with mesh, and add it to the scene. By default this will place our newly created cube at the 0,0,0 position.

In order to see anything we'll also need to call the render function and pass in our scene and camera we made in the first step.

Lastly our camera is currently at the same position as our cube, so lets move it back a bit with our final line 'camera.position.z = 5' to move the z-index of the camera back a bit.

We have created a beautiful cube.



3D rendering at it's finest

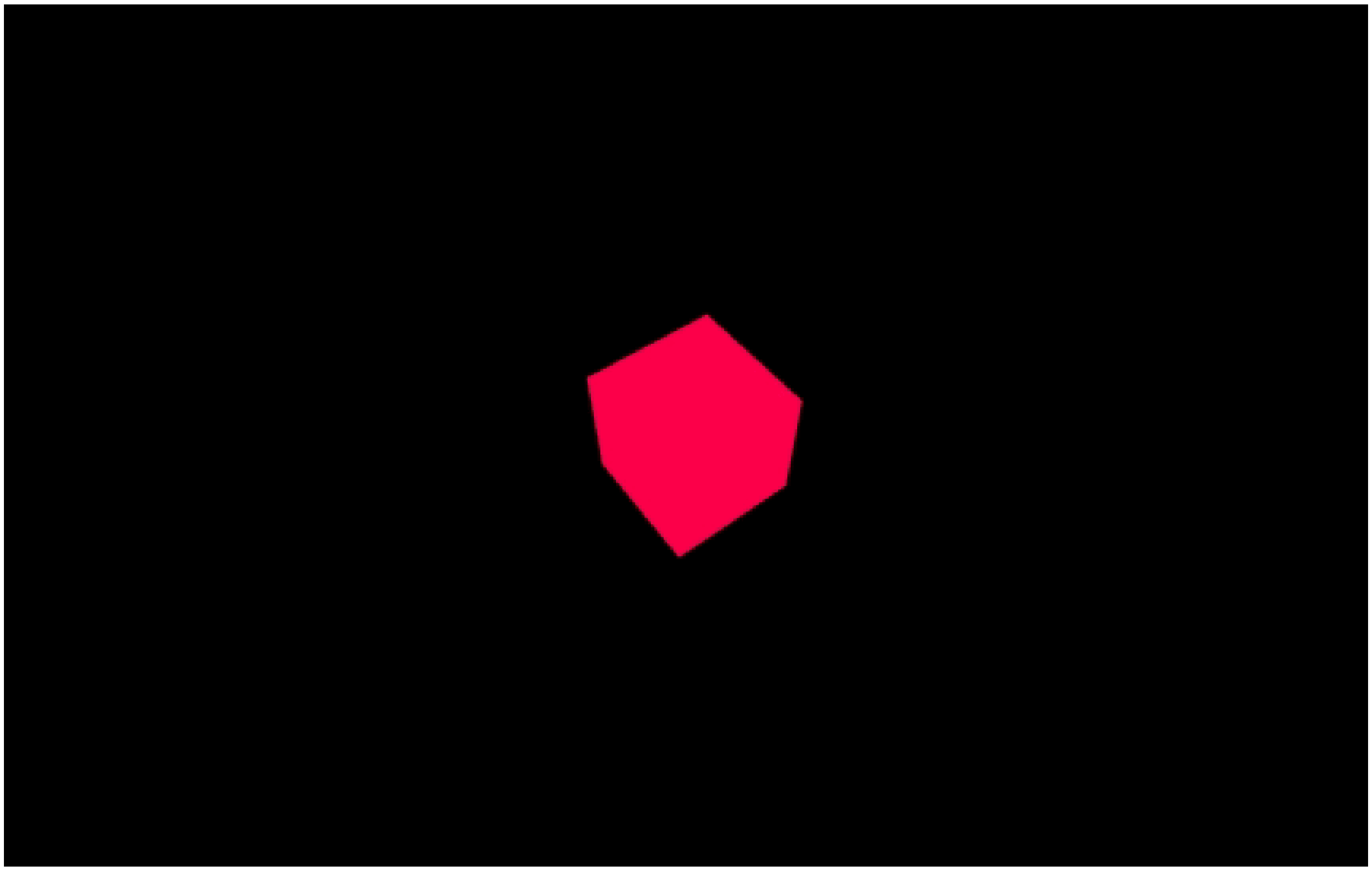
Okay so it may look like a box at the moment, but we expected a 3D cube. In fact, we have made a cube, but we're looking at it straight on. So lets move our cube around the get a better look.

Animation Time

Writing our Three.js animation won't be too different to making any Javascript animation, except for the fact that we'll be calling our renderer on every frame. Here's our basic spin animation:

```
function animate() {
 requestAnimationFrame( animate )
 cube.rotation.x += 0.04;
 cube.rotation.y += 0.04;
 renderer.render( scene, camera )
}
animate()
```

Like all requestAnimationFrame functions, this will be called 60 times every second, creating our smooth 60 FPS 3D render animation. Every frame we'll have the cube rotate on both the x and y axis by .04 of a radian. It might make sense to use 'Math.PI / some number' for units here but sticking with a basic number is fine for now. After the rotation is applied we have to remember to call our renderer.render function to display the change.



Cool but we're missing something. We have no light.

Let There Be Light

The real power of 3D rendering comes from how it is able to deal with light sources to create realistic looking scenes. How reflective our materials are, shadows, how colors appear, ect. all rely on light.

Let's start with the most basic form of light in Three.js with new Ambient Light.

```
var ambientLight = new THREE.AmbientLight ( 0xffffff, 0.5)
```

```
scene.add( ambientLight )
```

Create a light source, tell it the color (white in our case), and how intense is should be. Ambient light is omnipresent and applied to everything equally. It cannot cast shadows because it has no direction. It will just change how our colors appear.

Next let's create a Point Light. Think of this as our lightbulb. Light from this will spread in all directions equally from the point of origin.

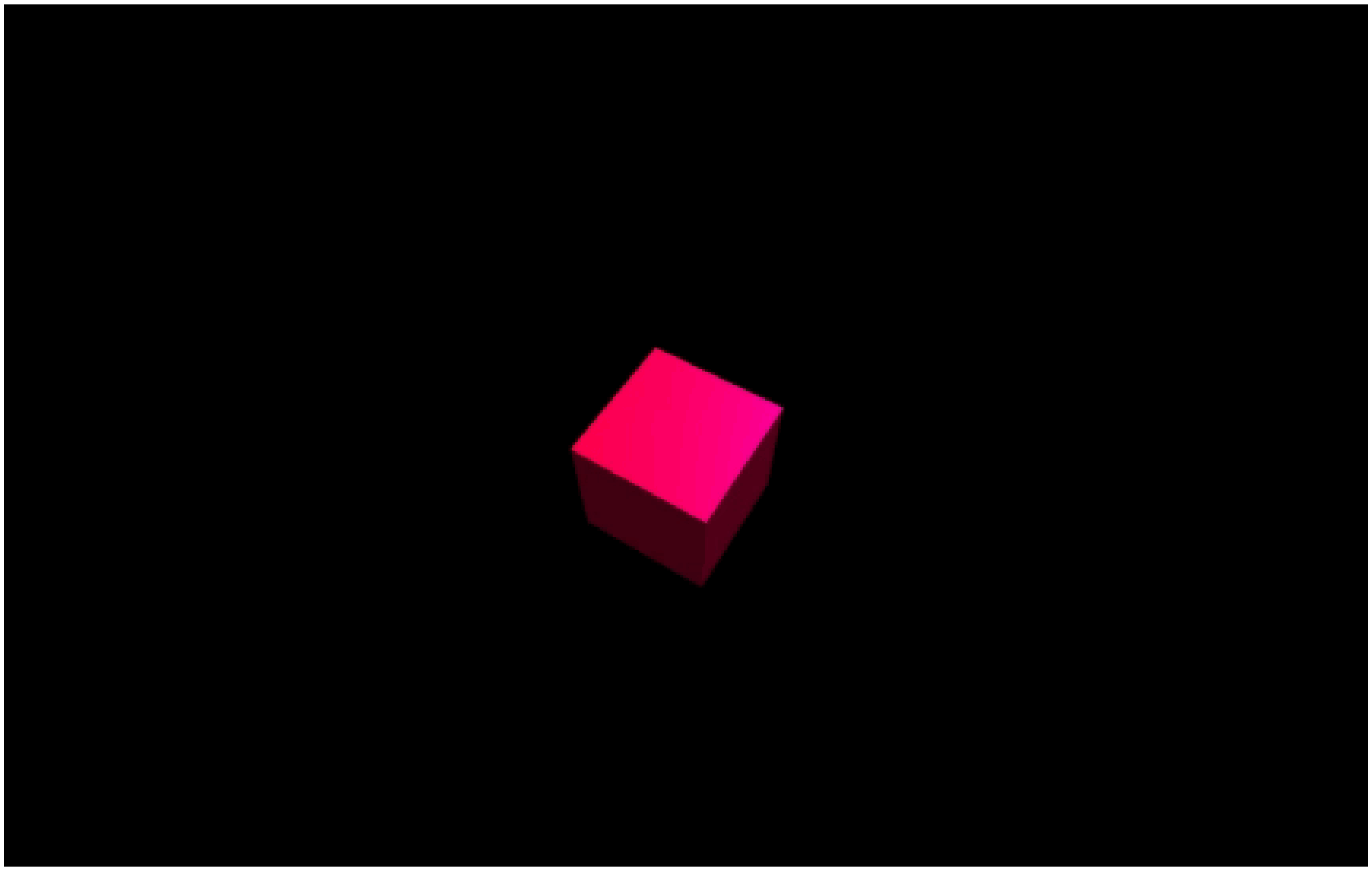
```
var pointLight = new THREE.PointLight( 0xffffff, 1 );
pointLight.position.set( 25, 50, 25 );
scene.add( pointLight );
```

We'll make this light white, but more intense. Also since there is a direct to PointLight, we'll need to give it a position. We'll put it to the upper right, and behind our cube.

The last piece of the puzzle is changing our cube material. Remember we picked the most basic material that didn't interact with light, but now we have light sources to work with. Let's edit our material code from before to:

```
var material = new THREE.MeshStandardMaterial( { color: 0xff0051 })
```

All we changed was instead of using THREE.MeshBasicMaterial, we're now using MeshStandardMaterial which is the standard physically based material.



Now let’s finish up by practicing adding another element to the scene. We’ll make another box, but this time bigger, and a different material that allows us to make it transparent and use the built in ‘wireframe’ property.

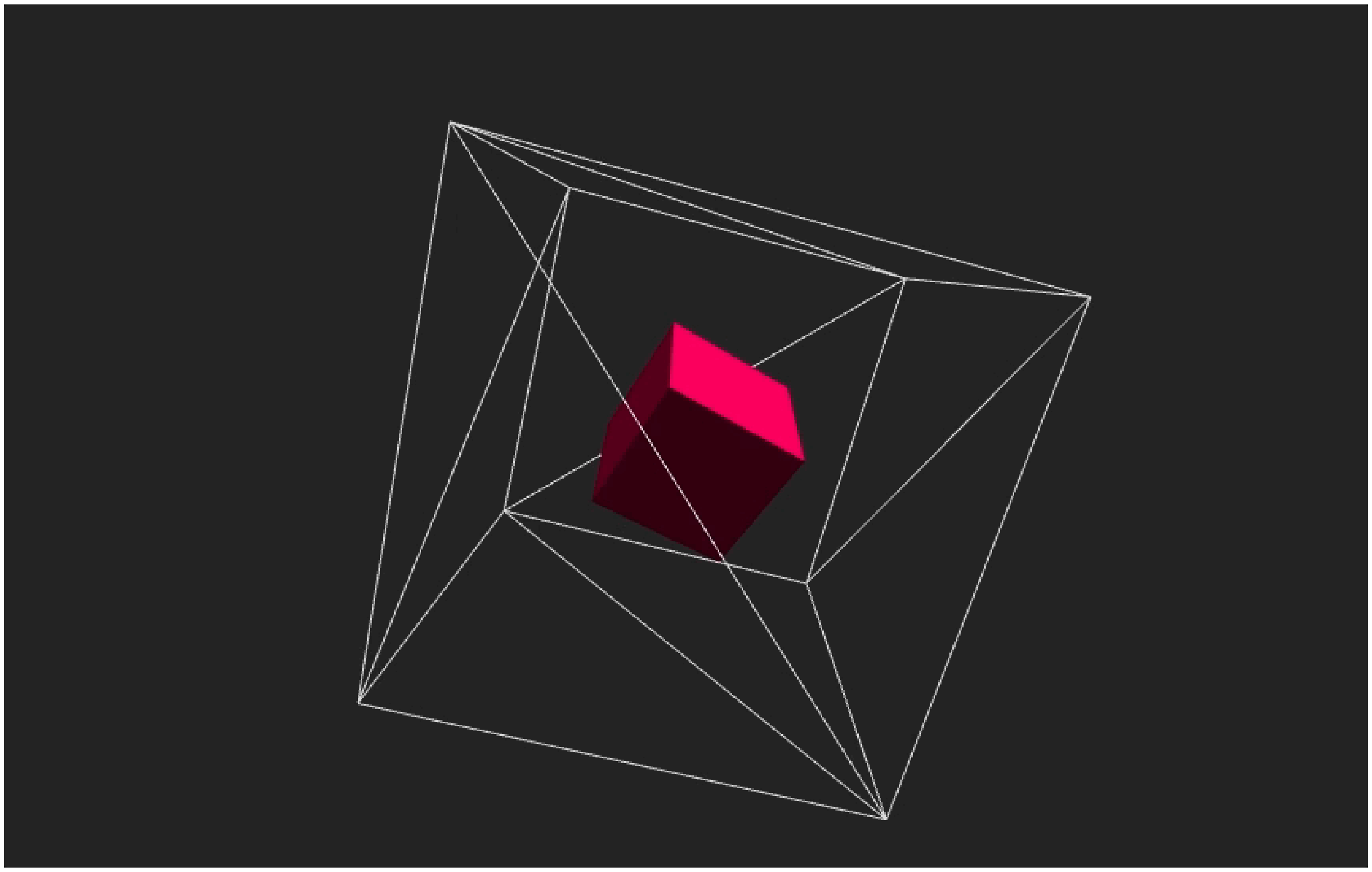
```
var geometry = new THREE.BoxGeometry( 3, 3, 3)
var material = new THREE.MeshBasicMaterial( {
 color: "#dadada", wireframe: true, transparent: true
})
var wireframeCube = new THREE.Mesh ( geometry, material )
scene.add( wireframeCube )
```

Next let’s add another animation to make the new cube move.

```
function animate() {
 requestAnimationFrame( animate )
 cube.rotation.x += 0.04;
 cube.rotation.y += 0.04;
 wireframeCube.rotation.x -= 0.01;
 wireframeCube.rotation.y -= 0.01;
 renderer.render( scene, camera )
```

```
  }
  animate()
```

We'll use a similar rotation, but slower and now you have it!



My final index.js file is here with small additions. Mess around with the settings, change things up, try adding new elements and reading through the docs to get a better feel.

```
// We need 3 things everytime we use Three.js
 // Scene + Camera + Renderer
const scene = new THREE.Scene()
const camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.innerHeight, 0.
const renderer = new THREE.WebGLRenderer({ antialias: true})

renderer.setSize( window.innerWidth, window.innerHeight )
// sets renderer background color
renderer.setClearColor("#222222")
document.body.appendChild( renderer.domElement )
camera.position.z = 5

// resize canvas on resize window
window.addEventListener( 'resize', () => {
        let width = window.innerWidth
        let height = window.innerHeight
        renderer.setSize( width, height )
        camera.aspect = width / height
        camera.updateProjectionMatrix()
})

// basic cube
var geometry = new THREE.BoxGeometry( 1, 1, 1)
var material = new THREE.MeshStandardMaterial( { color: 0xff0051, flatShading: true, metal
var cube = new THREE.Mesh ( geometry, material )
scene.add( cube )

// wireframe cube
var geometry = new THREE.BoxGeometry( 3, 3, 3)
var material = new THREE.MeshBasicMaterial( {
        color: "#dadada", wireframe: true, transparent: true
})
var wireframeCube = new THREE.Mesh ( geometry, material )
scene.add( wireframeCube )

// ambient light
var ambientLight = new THREE.AmbientLight ( 0xffffff, 0.2)
scene.add( ambientLight )

// point light
var pointLight = new THREE.PointLight( 0xffffff, 1 );
pointLight.position.set( 25, 50, 25 );
scene.add( pointLight );

```

```
function animate() {
        requestAnimationFrame( animate )
        cube.rotation.x += 0.04;
        cube.rotation.y += 0.04;
        wireframeCube.rotation.x -= 0.01;
        wireframeCube.rotation.y -= 0.01;
        renderer.render( scene, camera )
}
animate()
```

index.js hosted with ❤ by GitHub view raw

- Official Three.js Docs