

Nik Lever



THE
ThreeJS
PRIMER

Links



<https://www.facebook.com/groups/nikthreejs>



<https://x.com/NikLever>



<https://niklever.com>



<https://discord.gg/k2udGkHMEem>



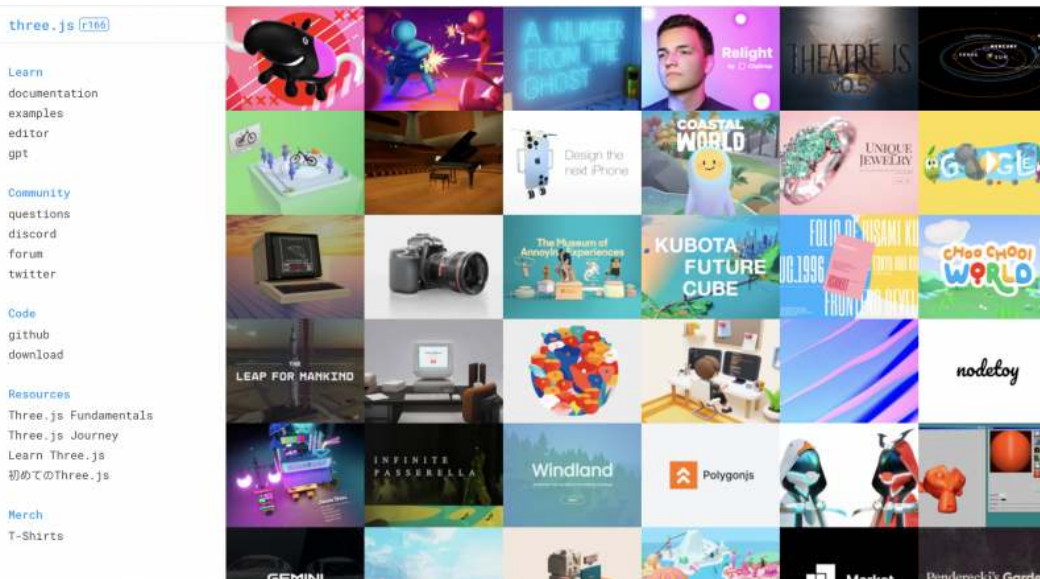
<https://youtube.com/c/NikLever>



<https://www.udemy.com/user/nicholas-lever-3/>

The ThreeJS Primer

Overview



Caption: threejs.org home page

Three.js is the most popular Open Source JavaScript library for **displaying 3D content** on the web, giving you the power to display incredible models and visualisations in your browser and even on your smartphone!

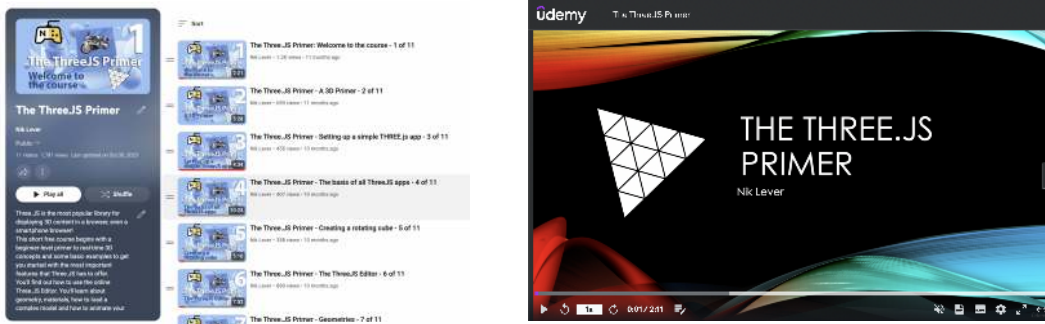
Displaying 3D content in a browser involves using the WebGL api. But the library takes care of all the complex details leaving you able to think in terms of models, cameras and lights.

This **e-book** begins with a beginner-level primer to real-time 3D concepts and some basic examples to get you started with the most important features that Three.JS has to offer.

You'll learn how to quickly create a **scene, camera, and renderer** and how to add meshes using the Geometry primitives included with the library. You'll find out how to use the **online Three.JS Editor** which will help as you learn to use the library. You'll learn about materials and how to load a complex model that you may find from an online store and how to animate your models.

This is a quick introduction to the most important features of the library. After completing the examples you will have a basic understanding of how to use Three.JS in your own Web Apps.

This course is also available as a series of videos on [YouTube](#) and a **free** course on [Udemy](#).



Caption: Course on YouTube and Udemy. Click images to view.

Table Of Contents

Introduction	3
2. A 3D Primer	7
3. Setting up a simple THREE.js app	15
4. The basis of all ThreeJS apps	19
5. Creating a rotating cube	28
6. The ThreeJS editor	33
Step 1	34
Step 2	36
Step 3	38
Step 4	40
Step 5	41
Step 6	42
Step 7	43
Step 8	44
Step 9	45
7. Geometries	46
8. Materials	53
9. Loaders	59

10. Animation

66

11. Where to from here

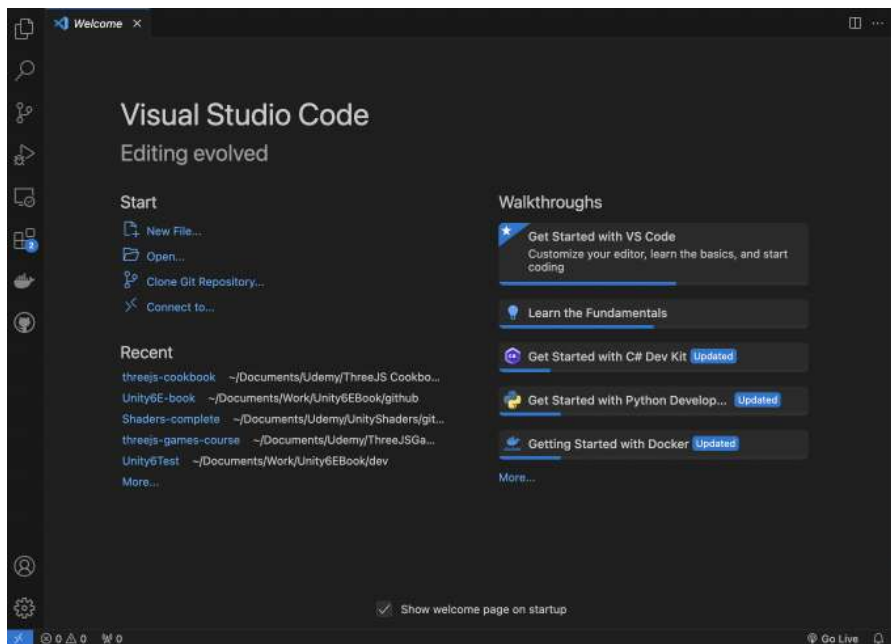
74

Introduction

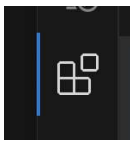


I'm Nik Lever and I've been creating real-time 3d in the browser for over 25 years.

To create **ThreeJS** apps we will be writing HTML and JavaScript code. For that we need a code editor. In this course I'll be using **VSCODE** it is available free if you haven't got a copy and want to follow my steps exactly then [download](#) and install it now (<https://code.visualstudio.com/>). If you prefer to use another code editor, then that's completely fine.

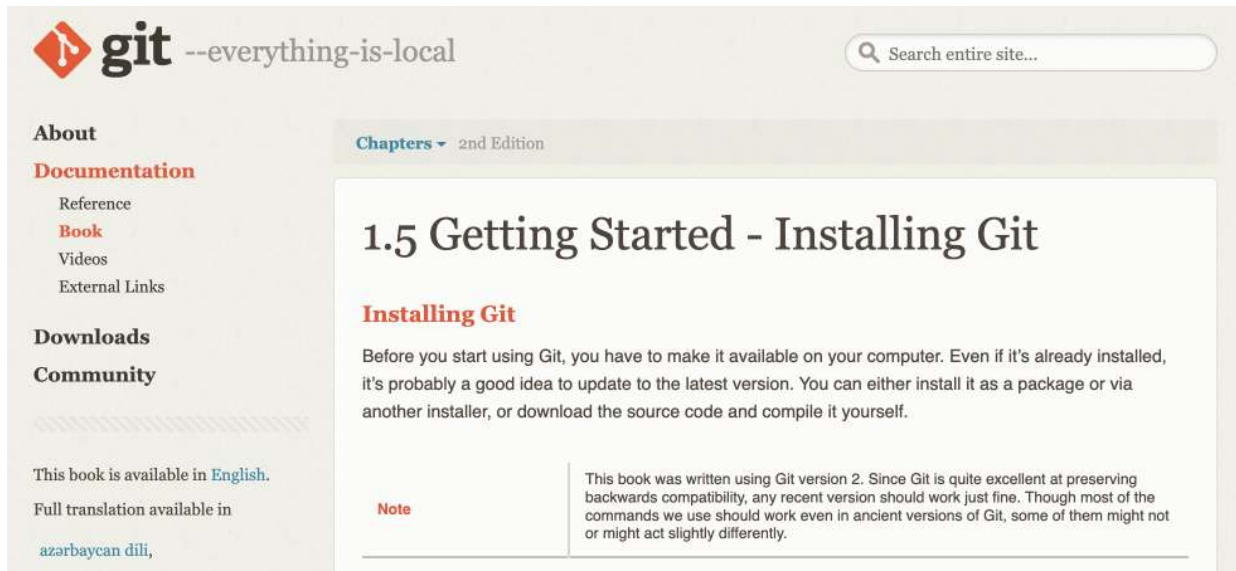


Caption: VSCode

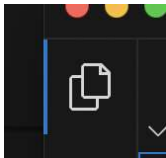


If you're using VSCode open it now, click the Extensions button, and in the search-field enter **Live Server**. Install this useful extension. It will help you run your ThreeJS apps.

If you don't have Git installed I urge you to install it. Follow this guide <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>



Caption: Installing Git



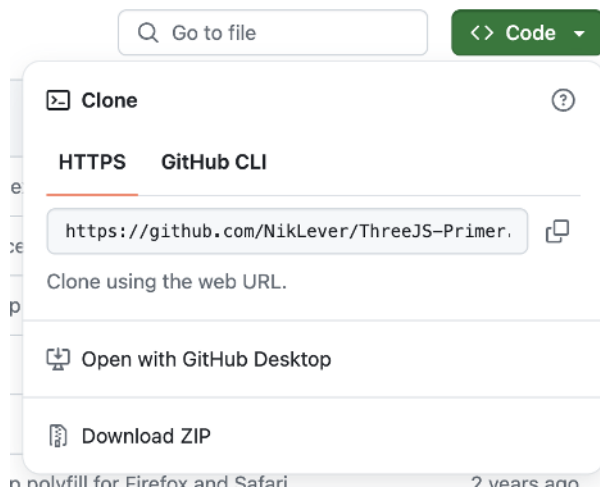
Time to get the course resources. Move to the explorer view and click the text that says clone repository enter this url:

<https://github.com/NikLever/threejs-primer>

You'll be asked to *'Select a Repository Location'*

The course resources will be transferred to a sub-folder called threejs-primer in the location you selected. Select open when asked and trust the authors.

If you prefer not to use Git then just go to the github address in your browser, there's a link in the resources, click the green code button and choose download zip.



Caption: Using the GitHub Code button

Unzip to a suitable folder then in VSCode choose **File > New Window** and select the Open option selecting the unzipped folder.

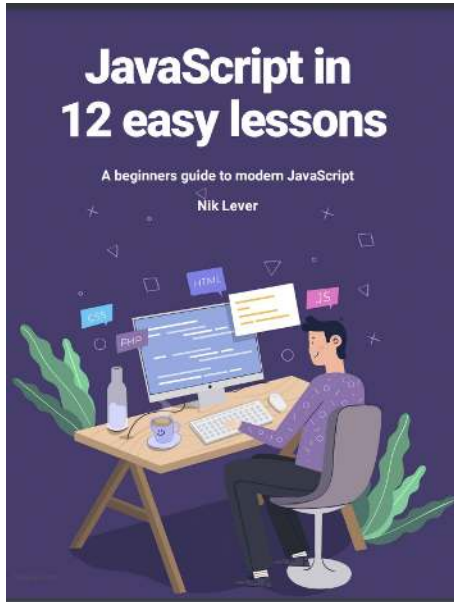
You're all ready for the course.

Let's review the steps

1. Download VSCode
2. Open VSCode and install the Live Server extension
3. Clone or download the course repository.

To share your student journey with others consider joining my Facebook Group or Discord server. Links on the first page of this book.

If you need to brush-up your JavaScript then download the free e-book "[JavaScript in 12 Easy Lessons](#)"

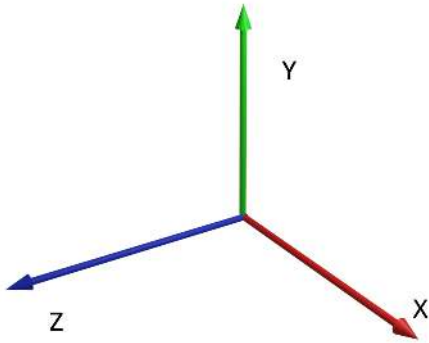


Caption: JavaScript in 12 Easy Lessons

Before you can start to create ThreeJS apps you'll need a basic understanding of real-time 3d and that's the aim of the next chapter.

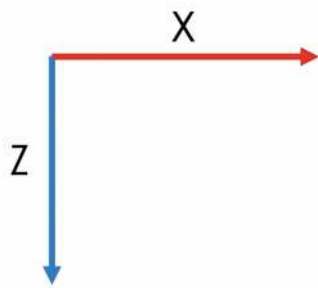
2. A 3D Primer

3D graphics is all about representing shapes in a 3D space using a coordinate system. The renderer in **ThreeJS** that we will be using is the **WebGLRender** and so we will focus, in the course, on the coordinate system used by **WebGL** which looks like this.



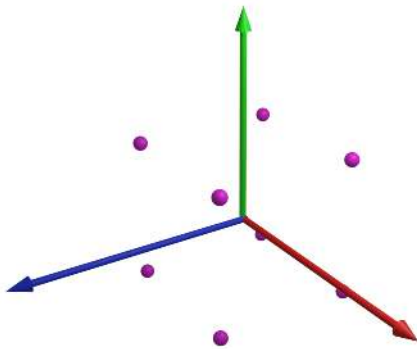
Caption: The coordinate system used by ThreeJS' WebGLRender

Other coordinate systems are common such as z pointing up but in this e-book y will always point up. Notice that assuming the viewer is looking down the negative z axis then the x axis extends to the right, the y axis extends upwards and the z axis points towards the viewer. This arrangement is sometimes referred to as **EUS**, East, Up, South. If the camera was positioned high in the y axis and looks directly down then the East and South directions are more obvious.



Caption: Looking down the y axis. X is East, Z is South, Y is Up. EUS coordinate system

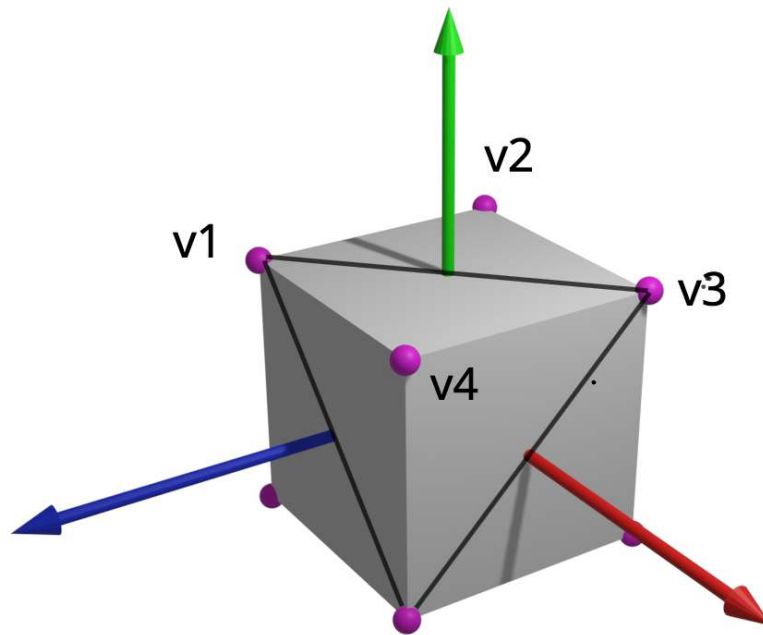
Objects are defined using vertices.



Caption: Vertices

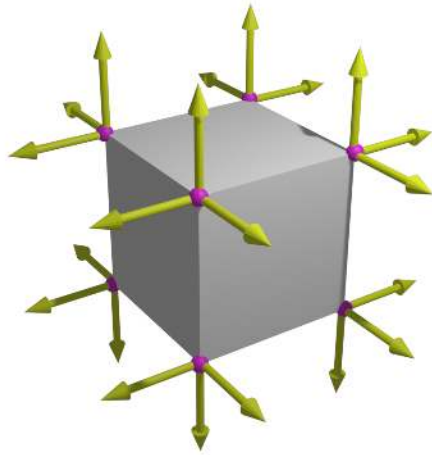
A single one is called a vertex. It is a vector value. If you are not sure what a vector is, don't worry, it's simple. A vector is simply a group of numbers. A single number is a scalar, but if we use two numbers together then this is a vector. For a vertex it needs an x, y and z value and so in ThreeJS terminology it is a Vector3. A vector with three numeric values.

An object is made out of vertices and faces. Suppose we have a cube; this has 8 vertices and 6 faces. Each face has 4 edges. Imagine the top face of this cube has the vertices v_1 , v_2 , v_3 and v_4 . But WebGL only uses triangles. So a cube actually has 12 triangles, each face being two triangles. The top face is v_1, v_2, v_3 and v_1, v_3, v_4 .



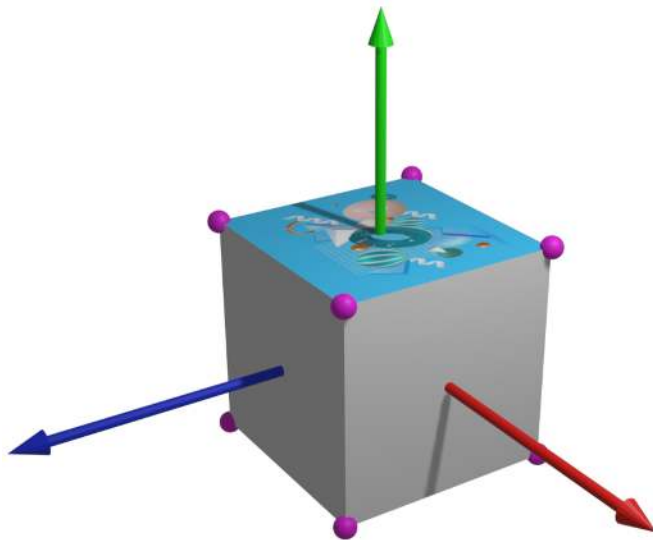
Caption: Describing geometry

A normal is a **Vector3** property that stores the direction in which a face is pointing, the length of this vector is usual of unit length. That is, it is 1 unit long.



Caption: Normals

And finally, a face will need to describe how it is coloured. Possibly as a simple colour or it could be by mapping an image onto the face.



Caption: Adding an image to a face

By carefully selecting the uv values, a 3D artist can map an image onto a complex mesh.



Caption: UV mapping

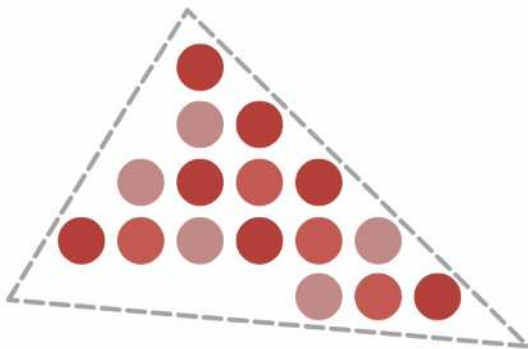
In the 3D application, we load in our 3D objects and their materials, then we can move, rotate and scale them using the ThreeJS library.

You might be relieved to know that we very rarely have to be concerned about the underlying details of the mathematics that converts this into an image on screen. But in principle WebGL uses shaders.

A shader consists of a vertex shader and a fragment shader. The vertex shader moves the vertex into the normalized clip coordinates. That is x, y and z are all converted into values between -1 and 1. ThreeJS has all the shaders you'll need for most things, so we won't be worrying about creating our own shaders in this course.

In general, ThreeJS will do the mathematical heavy lifting. But it is useful to know that the library uses matrices. A matrix is simply a 2-dimensional array of numbers. When working in 3d a matrix is usually 4 x 4. ThreeJS uses a matrix for the model that moves rotates and scales it. Another for the view which takes the cameras position into consideration. And finally, a matrix that projects this onto the 2D screen.

As well as the vertex shader we also have a fragment shader, this works at the pixel level. The fragment shader follows the vertex shader and so it already has the vertices converted into clip-space coordinates. The purpose of the fragment shader is to determine the colour for the individual pixel.



Caption: A fragment shader

The process of taking the 3D data and turning this into a 2D picture is often called the rendering pipeline and takes this form. So that's what is happening under the hood. But when creating web apps using ThreeJS unless you want to start writing your own shaders you'll be working principally at the 3D object level. So, you won't need to worry about vertices and pixels.

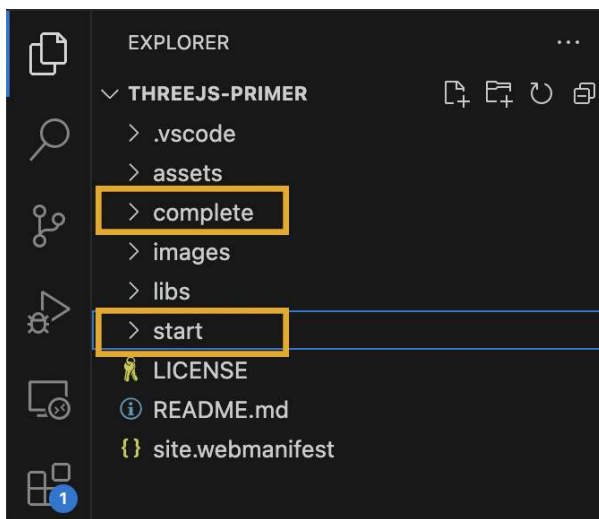
I hope this brief overview gives you an insight into what happens for all 3D apps from visualizations to games.

But it is time for us to look at creating our first ThreeJS app and in the next chapter we'll do just that.

3. Setting up a simple THREE.js app

ThreeJS comes in more than one flavour. We'll be using the modules version throughout this e-book.

If you're looking at the folder where you've downloaded or cloned the resources, you'll see that there is a **start** folder and a **complete** folder. To work along with the course, you need to use the version in the **start** folder, if you're having trouble then go ahead and look at the **complete** folder the problem is almost certainly a typo.



Caption: VSCode Explorer view

You can't view your work just by clicking on an html file. That will not work for 99% of the examples in the course.

The examples in this e-book assume you're using the Live Server extension in VSCode. It is very much worth installing.

OK let's open the file index.html in the folder start/lecture3.

```
30     <script type="module">
31         import { App } from './app.js';
32
33         document.addEventListener("DOMContentLoaded", function(){
34             const app = new App();
35             window.app = app;
36         });
37     </script>
```

This is already setup for you and each example in the resources is going to use pretty much the same approach. The index page will define a script tag, see above, that is set to type module, it will import the App class from the file app.js in the same folder as the index.html file and in the DOMContentLoaded event a new instance of this App class will be initialised. The window object will have a property called app set to this instance to facilitate debugging as will inevitably be necessary.

```
start > lecture3 > JS app.js > ...
1  import * as THREE from 'three';
2  import { OrbitControls } from '../libs/three/examples/jsm/controls/OrbitControls.js';
3
4  class App{
5      constructor(){
6          const container = document.createElement( 'div' );
7          document.body.appendChild( container );
8
9          window.addEventListener('resize', this.resize.bind(this) );
10     }
11
12     resize(){
13
14     }
15
16     render( ) {
17
18     }
19 }
20
21 export { App };
```

Now let's look at the app.js file, see above. This is an essentially empty class file. In this template I have imported the entire ThreeJS library, line 1, which is stored in the file ../../libs/three/build/three.module.js this is the same file you will find in the THREE.js repo in the build folder.

```
23     <script type="importmap">
24       {
25         "imports": {
26           "three": "../../libs/three/build/three.module.js"
27         }
28       }
29     </script>
```

We use an importmap, in the index.html file, to convert the string, *three*, into this path. I also import OrbitControls, app.js line 2, from ../../libs/three/examples/jsm/controls/OrbitControls.js.

When you use modules the import from path must be an absolute or relative path from the file to the import module. So, in app.js you need to go up 2 folders to reach the root of the resources then look in the libs/three/build folder to find the file three.module.js.

```
4 > class App{ ...
19 }
```

A JavaScript class file is defined with a name and then code inside curly braces.

The constructor method is called whenever a new instance of this class is created.

The constructor method may take any number of parameters or none.

```
5     constructor(){
6         const container = document.createElement( 'div' );
7         document.body.appendChild( container );
8
9         window.addEventListener('resize', this.resize.bind(this) );
10    }
```

Here, line 5, we have none. The template is setup to create a div element and append it to the body of the document. It is also setup to call the resize method of this class whenever there is a window resize event.

Scope is very important when using the keyword `this`. Inside the constructor method for the App class – `this` refers to an instance of the App. To call methods of the class we use `this` then the method name. If we do not add `bind(this)` to the method call for an event then inside the function `this` would have a different scope, it would refer, not to the App instance instead it would be the instance that had the event listener added, in this case the window. By adding `bind(this)` it ensures that `this` inside the class method is the App as expected. But you already know that from your JavaScript knowledge. This is just a reminder, because if you're a real beginner scope is confusing.

Now we're poised to enter our first code to display some ThreeJS content. Let's take a break and come back in the next chapter ready to start typing and testing. See you in a jiffy.

4. The basis of all ThreeJS apps

In the previous video we reviewed the starting template for your first ThreeJS app. We reviewed the basics of using modules and JavaScript class syntax. Not all developers have moved over to ES6 class syntax and modules, so I hope you'll forgive me for my brief reminder of this type of coding using JavaScript.

Now we're ready to create our first app. Open the file `app.js` in the folder **start/lecture4** and in the constructor method we'll create a virtual camera. When we imported the ThreeJS library we imported every class by using the wildcard character and the name, `THREE`, in caps.

```
import * as THREE from 'three';
```

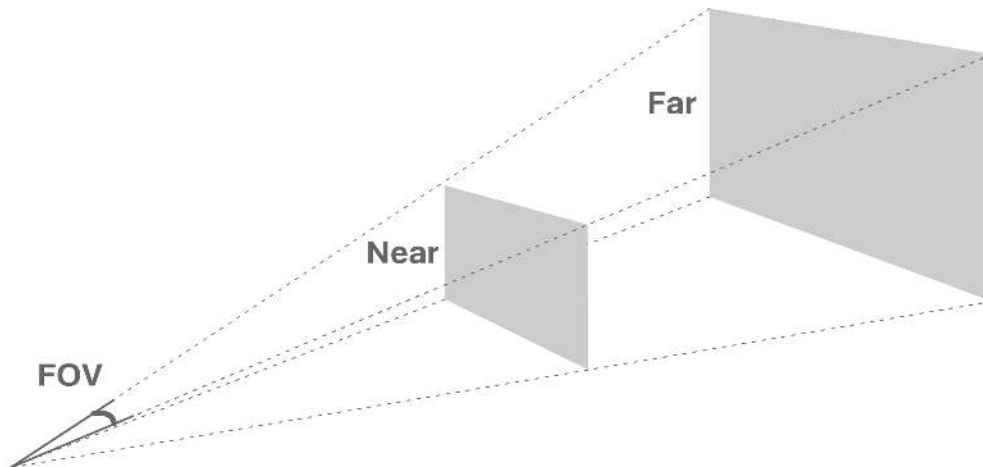
Each new instance of a class from this library needs to use the name, `THREE`, followed by a dot and then the name of the class. We're going to create a `PerspectiveCamera`. This takes four parameters. In the constructor after the `document.body.appendChild` line enter this code:

```
this.camera = new THREE.PerspectiveCamera( 60, window.innerWidth /  
window.innerHeight, 0.1, 100 );
```

Parameter one is a field of view in degrees, here we use 60 degrees. Then the aspect ratio, because we're filling the entire window, we use `innerWidth` divided by `innerHeight`. This ensures the models we create aren't stretched, whatever device layout you use.

Parameters 3 and 4 deal with the near and far values of this camera. Any attempt to render a pixel that is nearer than the near value or further away than the far value will be ignored.

A virtual camera has something called a frustum and the viewer will only see objects that are inside the field of view and the near and far planes, like this.



Caption: The Camera Frustum

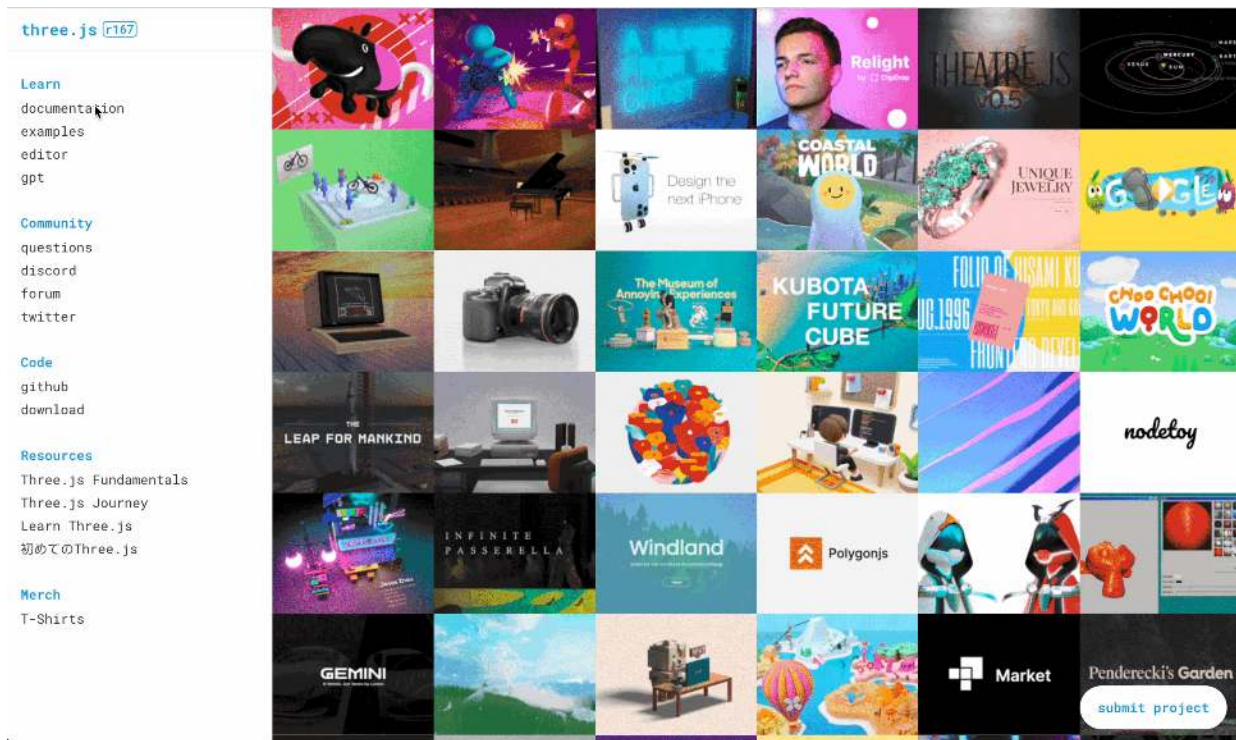
We can set the position of the camera like this.

```
this.camera.position.set( 0, 0, 4 );
```

As well as a virtual camera, a ThreeJS app needs an instance of a scene. This takes no parameters. Like a ThreeJS PerspectiveCamera the base class for a ThreeJS scene is Object3D.

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


It is a great idea to open the ThreeJS website when developing ThreeJS apps and games.



Caption: The ThreeJS website

Go to documentation and enter the class name. For a scene, notice Object3D then an arrow. Clicking Object3D we see that Object3D, is the base class for most objects in ThreeJS. Because it is a special kind of Object3D a scene will have a children property that is an Array of child objects. More about this later.

We're going to set the background colour to a mid-grey colour. The colour value is provided in hexadecimal format. Using Ox tells the JavaScript interpreter to expect a hexadecimal value. In decimal there are 10 possible numeric values in a column.

Value	100s	10s	Units
0			0
9			9
10		1	0

Once we get to 9 and increase by 1, then next value takes two columns. Each column to the left has a value ten times the one to its right. So, in this column the 1 actually represents 10.

Value	16s	Units
0		0
1		1
10		A
15		F
16	1	0

In hexadecimal there are 16 numeric values. For ten we use the letter A, for eleven B, up to 15 being F. Using this approach, a single column can represent the values 0 through to 15. Adding one to 15 we roll into the next column. A column is 16 times the column to its right. This value in hexadecimal isn't ten it is 16.

Red	Green	Blue
-----	-------	------

AA AA AA

Value	16s	Units
-------	-----	-------

170 A A

When we define a colour, the first 2 characters after 0x give the value of the red channel. AA is $10 * 16 + 10$, or 170 in decimal. Each colour channel taking a value between 0 and 255.

FF is $15 * 16 + 15$ which equals 255 and is the largest value in hexadecimal using just two columns. The middle two characters provide the value for the green channel and the last two the blue value.

Enter this code.

```
this.scene.background = new THREE.Color( 0xaaaaaa );
```

Before we can see any content from our ThreeJS scene we need a renderer. ThreeJS has a number of renderers but the one we need for our apps is the WebGLRenderer. You can optional pass a parameters object to the constructor, here we're going to set antialias to true, to reduce the jaggies along edges.

```
this.renderer = newTHREE.WebGLRenderer({ antialias:true } );
```

It's a good idea to use the renderer method set pixel ratio, this will avoid blurring on retina screens. Where css pixel density is less than actual physical pixel densities.

```
this.renderer.setPixelRatio( window.devicePixelRatio );
```

And it is essential to set the physical size of the renderer. Here we set the renderer to the full size of the window using innerWidth and innerHeight. When a WebGLRender is created, it creates a HTML5 canvas element, with the property name domElement and we need to add this to the container we created in the App constructor.

```
this.renderer.setSize( window.innerWidth, window.innerHeight );  
container.appendChild( this.renderer.domElement );
```

The renderer needs to render the scene repeatedly so that changes to the camera position or objects in the scene are constantly updated. We can set up an animation loop to do just that. It will take the render method of the app as a parameter, don't forget to use bind this to ensure the scope inside the function is the App instance. Now this method will called around 60 times a second.

```
this.renderer.setAnimationLoop(this.render.bind(this));
```

Slide down to the render method add a call to the render method of the renderer, this takes the scene as parameter one and the camera as parameter two.

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

Now if we test this in the browser, using Live Server. You should see a grey screen. It might seem like a lot of effort to get a grey screen. But there's actually a lot going on. Let's review the code.

The minimum any ThreeJS app requires is a scene, where we will add objects and lights. A camera that acts as the viewers position and orientation in the scene. And you will need a renderer, usually a WebGLRenderer instance.

```
1 import * as THREE from 'three';
2 import { OrbitControls } from '../libs/three/examples/jsm/controls/OrbitControls.js';
3
4 class App{
5   constructor(){
6     const container = document.createElement( 'div' );
7     document.body.appendChild( container );
8
9     this.camera = new THREE.PerspectiveCamera( 60, window.innerWidth / window.innerHeight, 0.1, 100 );
10    this.camera.position.set( 0, 0, 4 );
11
12    this.scene = new THREE.Scene();
13    this.scene.background = new THREE.Color( 0xaaaaaa );
14
15    this.renderer = new THREE.WebGLRenderer({ antialias: true } );
16    this.renderer.setPixelRatio( window.devicePixelRatio );
17    this.renderer.setSize( window.innerWidth, window.innerHeight );
18    container.appendChild( this.renderer.domElement );
19
20    this.renderer.setAnimationLoop(this.render.bind(this));
21
22    window.addEventListener('resize', this.resize.bind(this) );
23  }
24
25  resize(){
26
27  }
28
29  render( ) {
30    this.renderer.render( this.scene, this.camera );
31  }
32 }
33
34 export { App };
```

Code: Complete code for this chapter

In the code for this chapter we first created a camera, line 9. In this instance we use a **PerspectiveCamera**, so distant objects will appear smaller than close-up objects. When creating a **PerspectiveCamera**, you passed parameters to the constructor method. Parameter 1 is the field of view, the FOV, this is in degrees. The second parameter is the aspect ratio the width of the window divided by its height. Because we are using the whole size of the window we can use the **innerWidth** and **innerHeight** properties of the window object. Parameter 3 is the near value and parameter 4 the far value. Objects nearer than parameter 3 will be clipped and those further away than parameter 4 will be clipped.

Real-time computer graphics uses a virtual cage to define a clipping region. This cage is called a frustum and although it might be tempting to set near to a tiny value and far to a massive one. The recommendation is to keep within as small a range as your scene will require. The renderer uses a special buffer when it builds up the image to display. This buffer stores the distance away from the camera for each pixel rendered and it allows nearer objects to mask distant ones. The accuracy of this masking is determined by the range between near and far. If this range is measured in the millions when in fact your objects never exceed the range 100 then you are likely to get rendering errors.

Once the camera is created it will be at 0,0,0, looking directly down the negative z axis. Recall that WebGL has by default, x increasing to the right, the east, y increasing upwards and z increasing out of the screen, south. At line 10 we move the camera back to 4 in the z. Think of it as 4 metres from the origin.

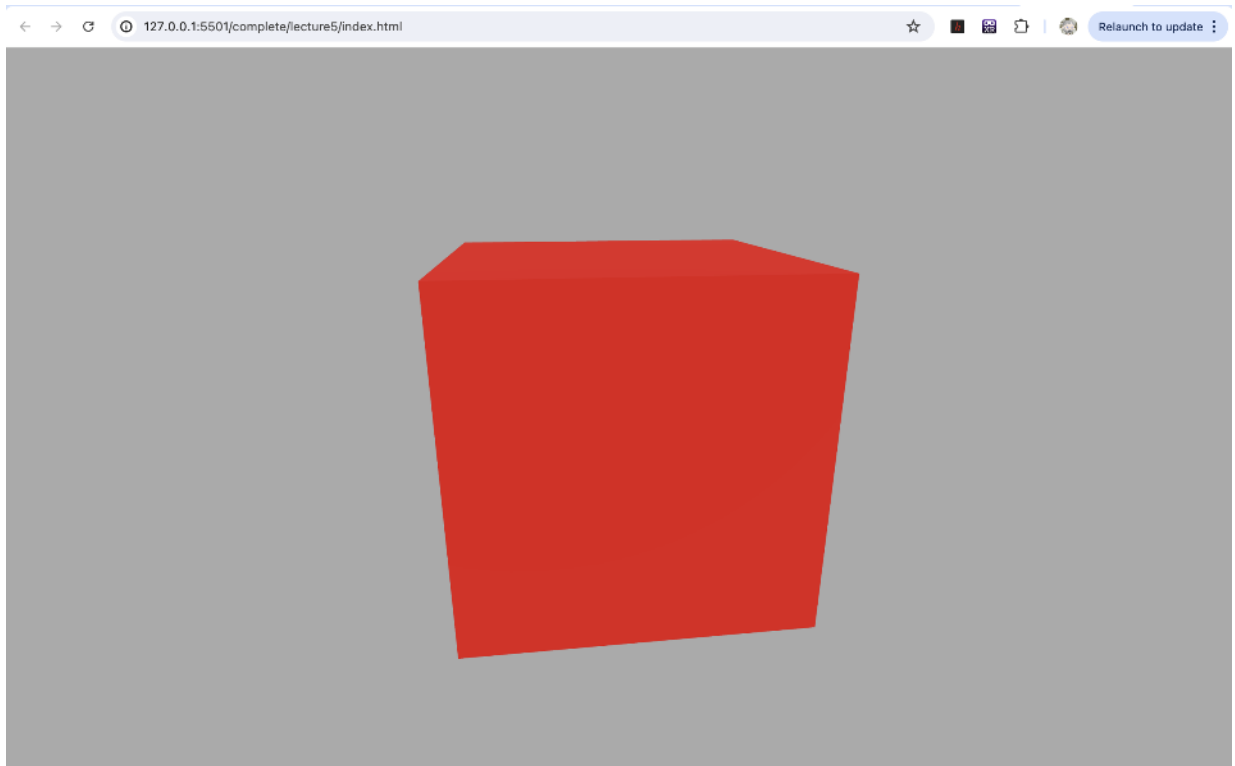
After creating the camera we create a scene, line 12. This is easily done and requires no parameters. By default the background will be white. To alter this we change the background colour to grey, line 13, using an instance of a ThreeJS **Color** class. The **Color** class can take a hex value to set its value.

Then we create the renderer, line 15. It is an instance of a **WebGLRenderer** with antialiasing set to true. Without setting antialiasing to true the app will suffer from jaggy. It is important to set the pixel ratio, otherwise there is a danger of blurring on many devices, line 16. And it is essential to set the size of the renderer, line 17. Here, because we're filling the window, we use inner Width and Height of the window object. When a renderer is created it creates a domElement and this should be added to the container to ensure it is visible, line 18.

The renderer has a method called **setAnimationLoop** that takes a callback. At line 20 we use the class method render and to ensure our scope is the App instance we add `bind(this)` to the reference. The **setAnimationLoop** is called up to 60 times a second if the browser can sustain that rate. It is an alternative to using **refreshAnimationFrame**.

So, now we have the key ingredients of a ThreeJS app, a scene, camera and renderer. And the renderer is being called many times a second. But grey is a bit boring. In the next chapter we'll start to add some content to our 3D world.

5. Creating a rotating cube



Caption: The code for this chapter running in a browser

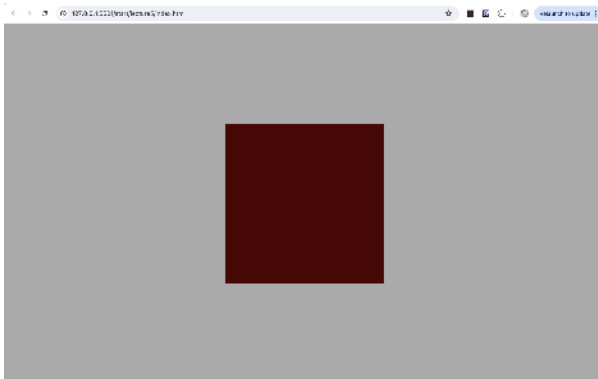
To code-along with this chapter open `app.js` in the folder `start/lecture5`

Let's add an object. Enter

```
const geometry = new THREE.BoxGeometry();
const material = new THREE.MeshStandardMaterial( { color: 0xFF0000 } );
this.mesh = new THREE.Mesh( geometry, material );
this.scene.add(this.mesh);
```


Using the ThreeJS library we create an object that we can see by creating geometry and a material. We'll look at the other geometries that we can create using the library in a later chapter. Here we create a **Box**, since we pass no parameters it is 1 unit wide, high and deep.

We create an instance of a type of material called **MeshStandardMaterial**, this is quite a sophisticated material and we'll learn more about it later in this book. For now we simply set its colour value. Using `0xFF0000`, we have the maximum value applied to the red channel, FF and minimum to both the green and blue channels, each set to 00. Consequently the material is red.

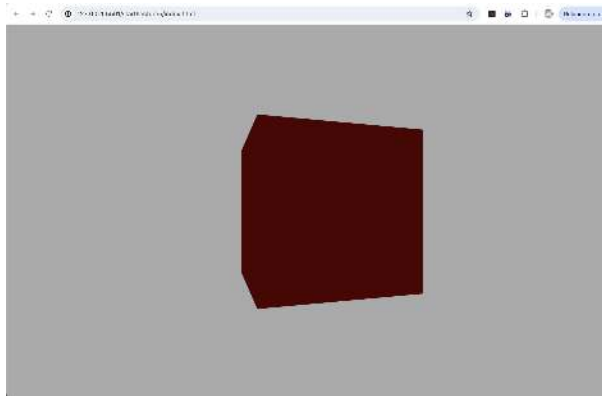


Caption: Cube appears as black square

Now we have geometry and a material we can create a **Mesh**, this takes the geometry as parameter one and the material as parameter two. If we save and use Live Server all we see is a Black square. To prove this is actually 3D. Let's add some code to the render method. Enter

```
this.mesh.rotateY( 0.01 );
```

Save and let Live Server refresh.



Caption: Cube rotating

Now you can see it looks more 3D. But it's still black, it should be red, what's gone wrong? The reason is simple. In our scene there are no lights. Let's add some, just after the scene is initialised add

```
const ambient = new THREE.HemisphereLight(0xffffff, 0xbbbbff, 0.3);
this.scene.add(ambient);
const light = new THREE.DirectionalLight(0xFFFFFF, 3);
light.position.set( 0.2, 1, 1);
this.scene.add(light);
```

Here we create an ambient light, one that illuminates not based on location or position. A **HemisphereLight** has a different color for surfaces that pointing up, they'll be hit by a white light, FFFFFFFF. But surfaces pointing down, get a bluey gray coloured light hitting them. We control the intensity of this light with the third parameter, here it is set to 0.3. For the light to affect the scene it must be added to the scene.

As well as the ambient light we'll add a directional light. This type of light points from its position to the origin, 0,0,0, or a target object if one is assigned. By moving it to 0.2, 1, 1, we can control the direction in which the light points. For the light

Now it is looking like a rotating red cube. But there are a couple more things to do before you go. At the moment the user cannot interact with the page at all. Using an **OrbitControls** instance it is very easy to allow the user with a mouse or touch event to rotate the scene. Just enter

```
const controls = new OrbitControls( this.camera, this.renderer.domElement );
```

Now dragging with the mouse rotates the view of the scene.

There is still a little problem to address. If we resize the window then the renderer does not resize. We need to add some code to the **resize** method. Enter

```
this.camera.aspect = window.innerWidth / window.innerHeight;  
this.camera.updateProjectionMatrix(); this.renderer.setSize(  
    window.innerWidth, window.innerHeight );
```

We update the camera aspect ratio and by so doing we need to update the projection matrix and we adjust the renderer size.

Now resizing the window causes the renderer to resize. Try commenting out the **updateProjectMatrix** line to see its effect.

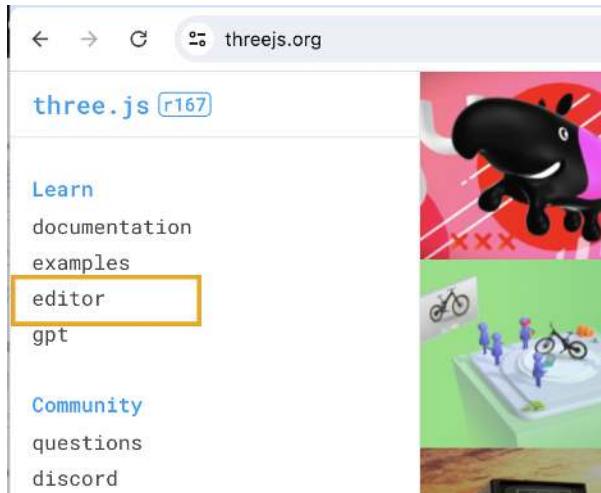
Can you change the rotation of the cube so it rotates around the x axis not the y axis?

Nice easy one. Just a case of changing rotateY to rotateX.

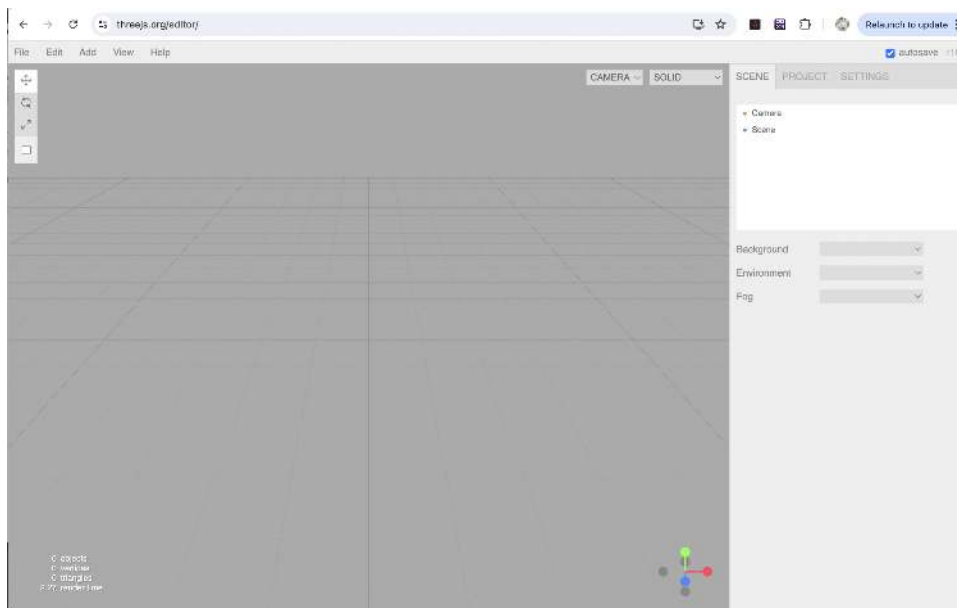
I hope you can see how easy the ThreeJS library is to use and how flexible. In the next chapter we'll look at a great online tool for you to use that will help you learn about the library.

6. The ThreeJS editor

Now you know the basics of a ThreeJS web app it's time to learn more. A really useful tool for learning is the ThreeJS editor. Open the [ThreeJS website](https://threejs.org), remember that is threejs.org. Probably time to bookmark this important resource.



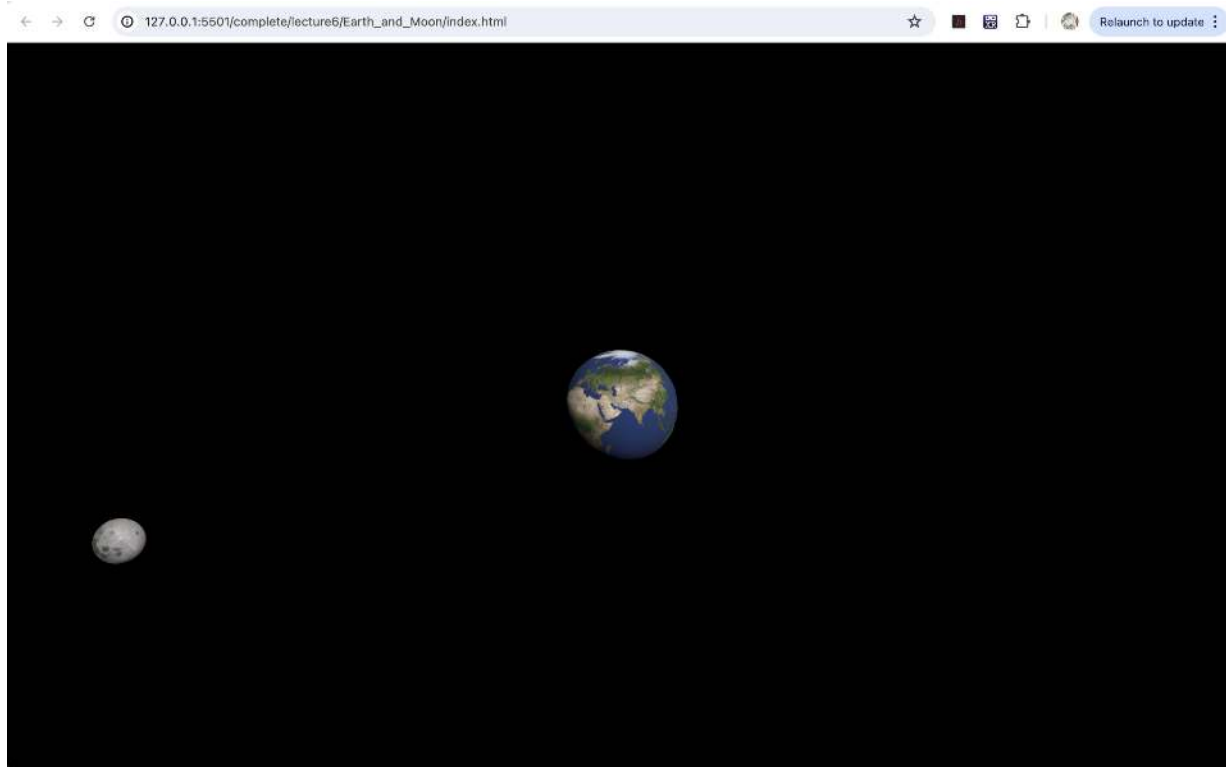
In the group of links at the top left is the editor link. This opens an online tool that lets you play around with nearly the entire ThreeJS library without writing a single line of code.



Caption: The ThreeJS Editor

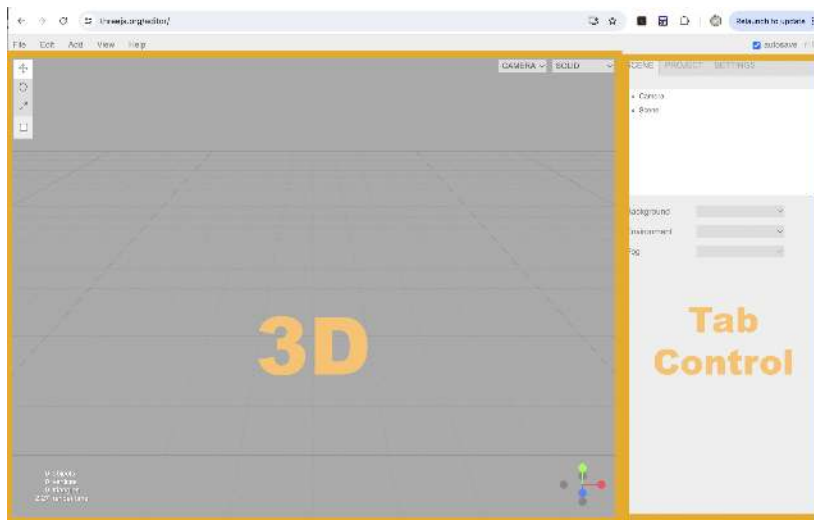
Step 1

The best way to learn is to play along as I describe the steps. So hopefully you now have the editor open in your favourite browser and you can switch to the editor and do each step. OK, here goes



Caption: The completed project

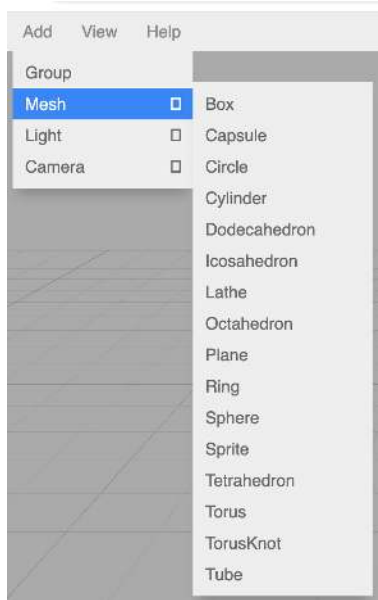
We're going to create a simple scene with the Earth rotating and add a Moon that is a child of the Earth. That means as the Earth rotates the Moon moves with it.



Caption: Editor layout

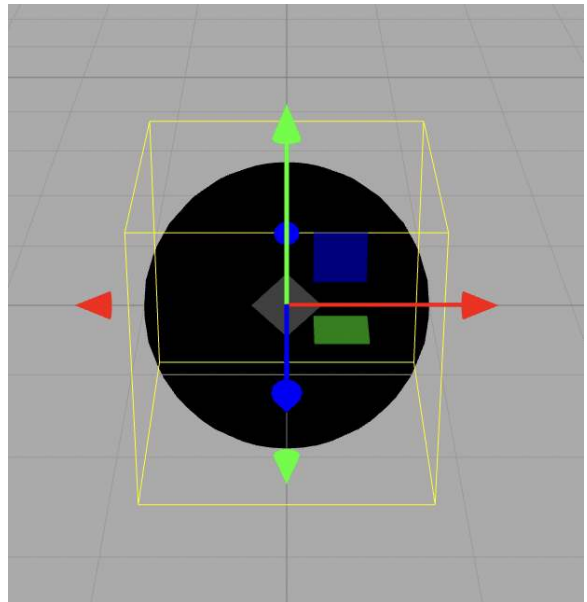
But before we do that lets just take a look at the interface. The main area is the 3D view. This will show you the camera view. At the top is a standard tool bar with various options. At the side is a tabbed control which shows the various inspectors for Scene, Project and Settings. We'll mainly be using the Scene tab. But for now, click the Project tab and enter 'Earth and Moon' as the project name. Leave the other details at the default.

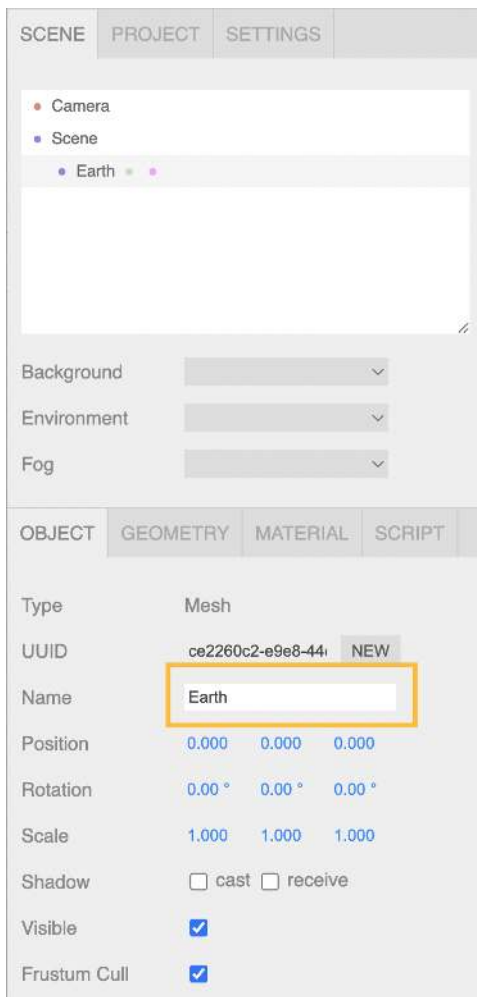
Step 2



Now click Add in the top tool bar. Notice the different object types you can add. We'll be looking in detail at these in a later chapter. For now, choose Sphere.

This will add a Sphere. You'll see that it is black.



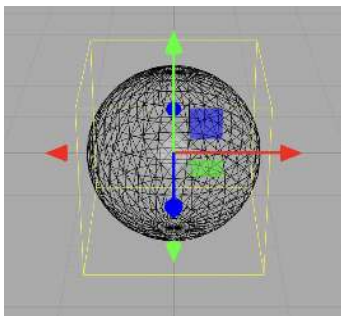


Find the Sphere in the Scene objects list. Using the Object tab, click the name 'Sphere' and change it to 'Earth'.

Notice that the object is positioned at 0 in the x, y and z and has rotation 0 in the x, y and z axes.

Can you remember which direction is which? If the camera is not rotated then x is left and right, y is up and down, and z is forward and backward.

Now switch to the Material tab. Scroll down to the bottom and click the Wireframe checkbox. The Sphere switches from solid black to a wireframe

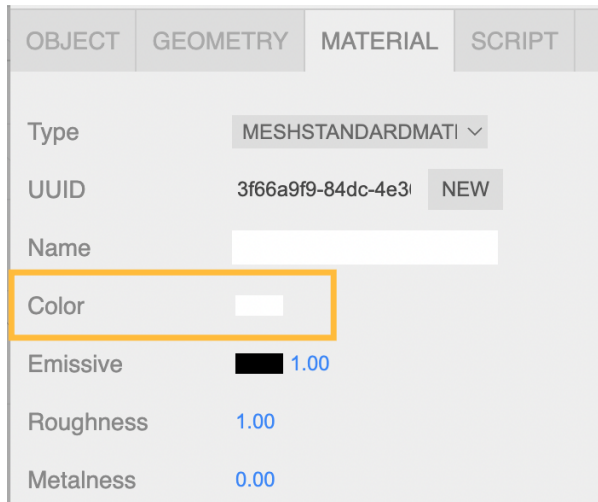


render. Now you can

clearly see the number of triangles being used to create this shape. Unclick the checkbox to go back to a solid black render.

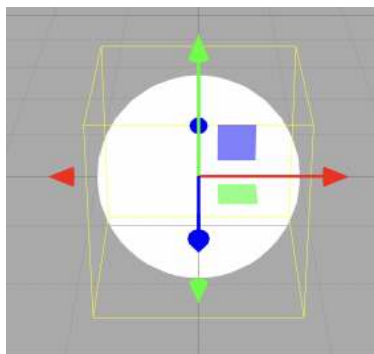
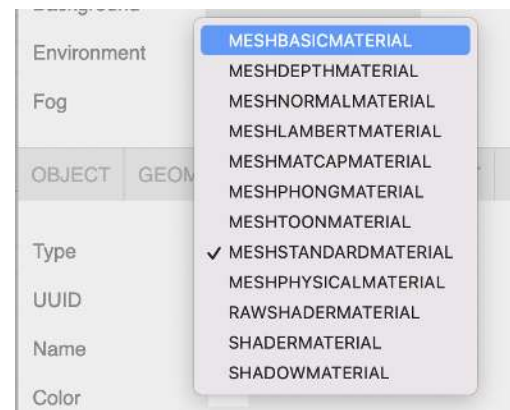


Step 3

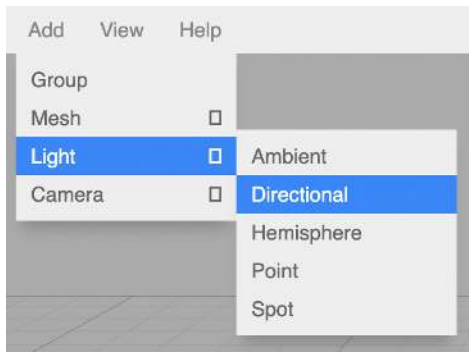


You might think that solid black is a bit boring. But if you have been looking carefully you'll see from the Material tab that Color is set to white.

So why is it showing as black? The answer is you are using a type of material that uses lighting, but you have no lights in your scene. Before we add a light try changing the material type to MeshBasicMaterial.

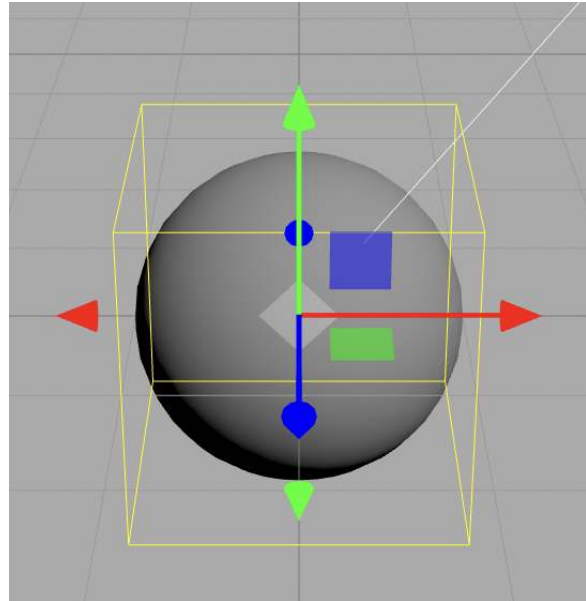


Instantly the sphere is shown as white. But it doesn't look very 3D. That is because MeshBasicMaterial doesn't use lighting. Switch it back to MeshStandardMaterial.



Now go up to the tool bar and choose **Add>DirectionalLight**.

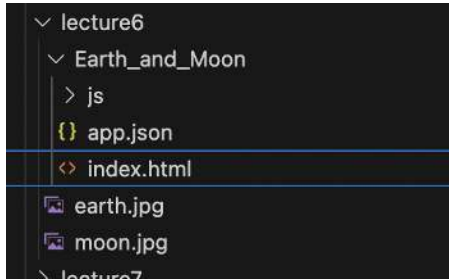
That's much more 3D. In the Object tab you can see the Position of the light. Given that the only thing that is important about a **DirectionalLight** is the direction, why have a position. That is because ThreeJS uses two things to decide the direction for a **DirectionalLight**, its position AND its target. By default, the target will be the location [0,0,0]. But you could assign an object to be the target.



Finally set intensity to 3.

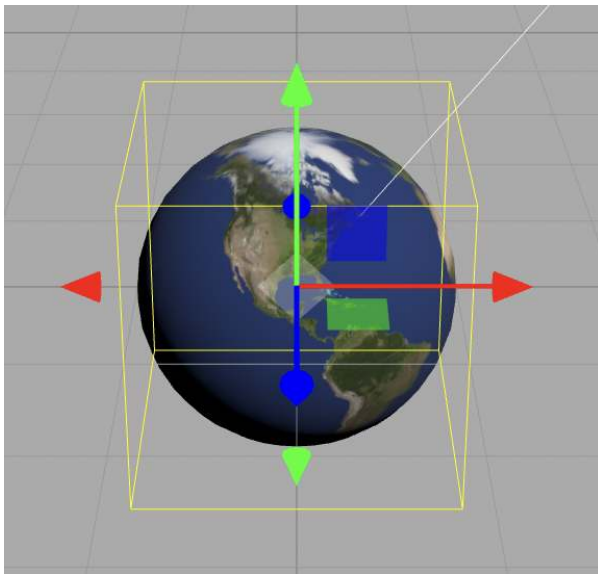
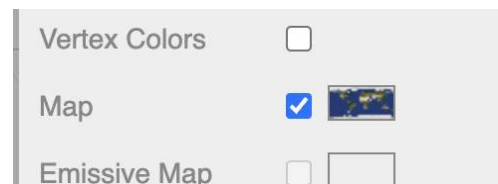
Step 4

Click on the sphere and choose the *Material* tab. We are now going to assign a texture to the sphere. Click on rectangle to the right of the *Map* checkbox. You will see a file

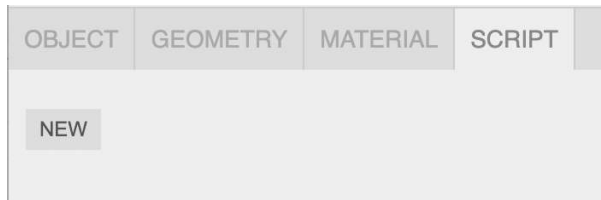


selector. Now we need the resources that you downloaded at the start of this book. Navigate to the folder `complete/lecture6`.

In there, select the `earth.jpg` file. At first you will see no change to the 3D view. But if you click the checkbox for the *Map* you will see the grey sphere become a reasonable representation of the earth.

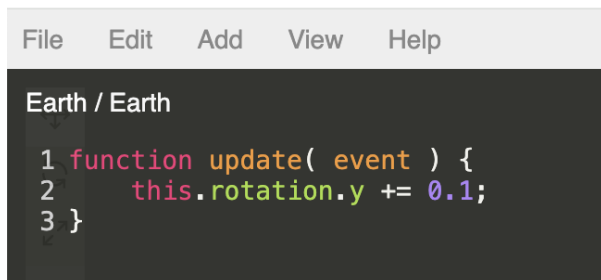
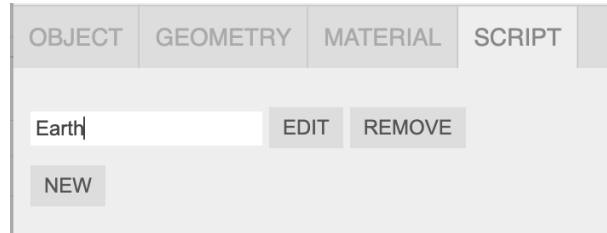


Step 5



Make sure the Earth object is in the Scene pane. Switch to the SCRIPT tab. Click the NEW button.

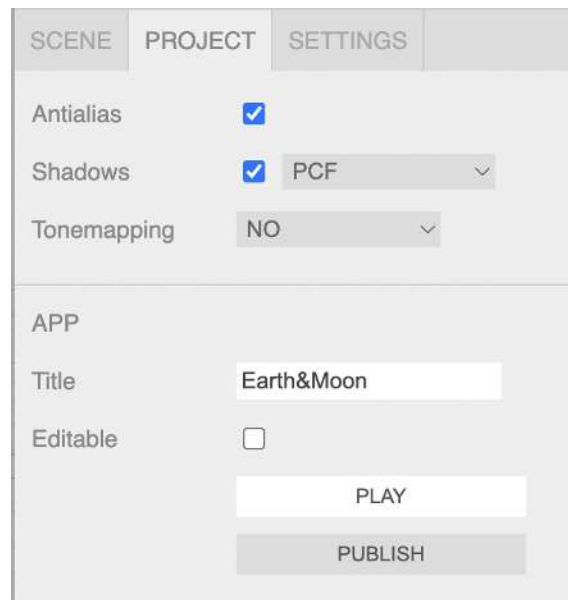
Give it the name Earth and click the EDIT button.



Between the curly braces, enter `this.rotation.y += 0.1;` Because we have the earth selected the keyword `this` refers to the earth object. All ThreeJS objects have a rotation property. Here we

are adding 0.1 to the y component of this property. This will cause the Earth to spin around its y axis.

Switch to the PROJECT tab. Press the PLAY button. You'll see the Earth object is spinning very fast. Click the STOP button. Changing the 0.1 to 0.01 makes for a more suitable rotation.

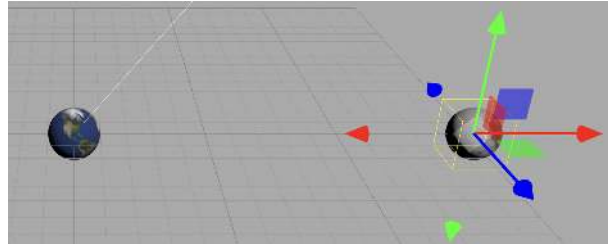




Step 6

UUID	cbc184c5-4ad3-4aaf	NEW	
Name	Moon		
Position	0.000	0.000	0.000

Add another Sphere, select it in the Object list and using the Object tab change its name to 'moon'.

Then using the red arrow, drag the moon to the right.



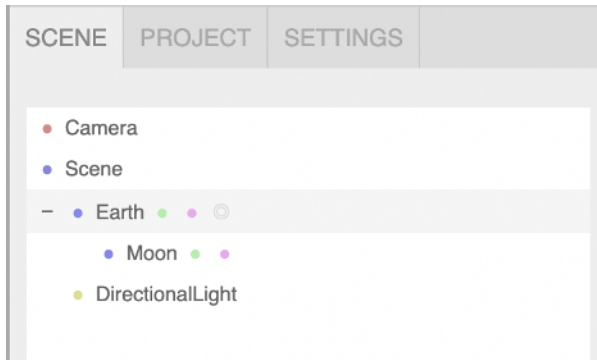
Vertex Colors	<input type="checkbox"/>
Map	<input checked="" type="checkbox"/> 
Emissive Map	<input type="checkbox"/> 

Now switch to the Material tab and set the Map to moon.jpg from the resources folder we used earlier.

Finally the moon should be smaller than the earth under the Geometry tab change the radius to 0.3.

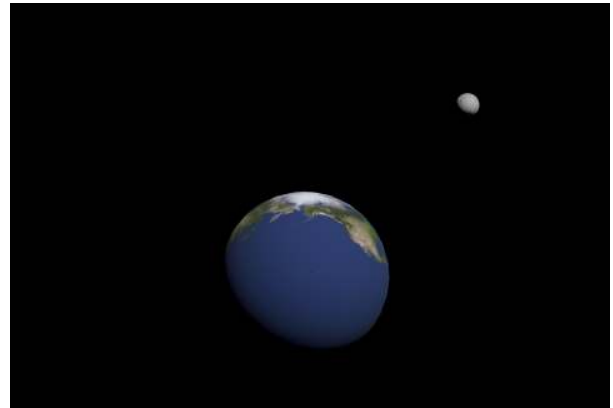
Name	
Radius	0.30
Width segments	32

Step 7

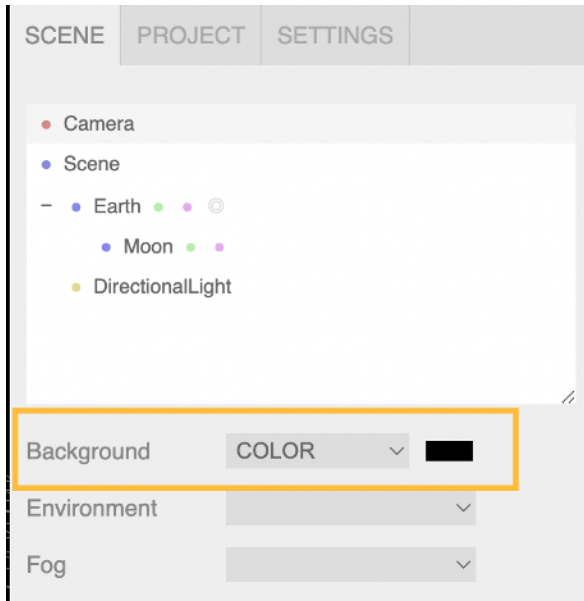


If you drag the Moon over the Earth in the Scene object list, the moon becomes a child of the earth and will inherit its motion. Try dragging the earth now.

Notice that the moon moves with it. When we press PLAY the moon appears to orbit the earth as the earth spins on its y axis. This is the effect of the child-parent relationship.

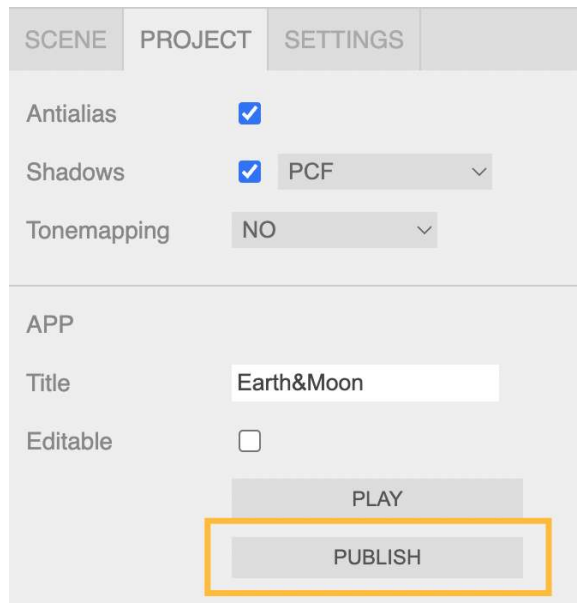


Step 8



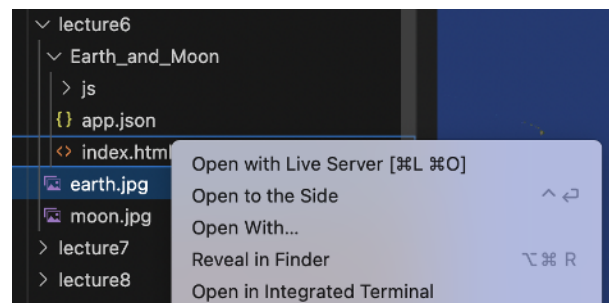
Finally change the background colour to black by clicking the colour bar to the right of the word Background.

Step 9



Now we're going to Publish our work. Select the PROJECT tab and press the PUBLISH button. The ThreeJS editor bundles the project into a zip file.

Find the file in the Downloads folder and unzip it to your development folder and launch the **index.html** file with **Live Server**. And there you have your earth and moon scene animating in all its glory.



This is a great way to get started with ThreeJS and you can get a scene ready for the web with an absolute minimum of code.

The ThreeJS editor really helps when you are learning the library. You can see what happens when you adjust the properties of objects in your scene. In the next chapter we are going to look at creating various different geometries in code.

7. Geometries

We found in the third chapter of this book that to see an object via the renderer it is usually a **Mesh** object, there are exceptions to this, such as Line segments and Particles. But the main visible object you will see in a ThreeJS scene will be a **Mesh**. We also learnt that when a new **Mesh** instance is created it takes a geometry parameter and a material parameter. We'll look at materials in the next chapter, but in this chapter, we will focus on the different geometries you can generate just using the library. Later in the book we'll look at using external programs to create meshes, but at this stage we'll look at the ones you can create with just the library code.

The geometry classes included with the library are the primitives **Box, Circle, Cone, Cylinder, Dodecahedron, Icosahedron, Octahedron, Plane, Sphere, Tetrahedron, Torus, TorusKnot** and there are several classes that will construct geometry such as **Edges, Extrude, Lathe, Parametric, Shape, Text** and **Tube**. All the details for using the primitives are on the [website](#) in the documentation section. So rather than going through them all let's look at just a couple. Then move to the construct options which are more difficult to understand and use.

Take a look at start/lecture7, for the starting template for this chapter. It is essentially the same as lecture5.

```
26
27 //Replace Box with Circle, Cone, Cylinder, Dodecahedron,
28 //Icosahedron, Octahedron, Plane, Sphere, Tetrahedron,
29 //Torus or TorusKnot
30 const geometry = new THREE.BoxGeometry();
31
```

The only thing we will change is the geometry used when creating the **Mesh**. Just replace **Box** with **Circle**, **Icosahedron** or **TorusKnot** or any of the list of geometries to see a default example rotating.

```
26
27 //Replace Box with Circle, Cone, Cylinder, Dodecahedron,
28 //Icosahedron, Octahedron, Plane, Sphere, Tetrahedron,
29 //Torus const geometry: THREE.CircleGeometry
30 const geometry = new THREE.CircleGeometry();
31
```

If you feel that the circle is too faceted then just take a look at the documentation to see that a **CircleGeometry** instance can take up to 4 parameters.

Constructor

CircleGeometry(radius : Float, segments : Integer, thetaStart : Float, thetaLength : Float)

radius — Radius of the circle, default = 1.

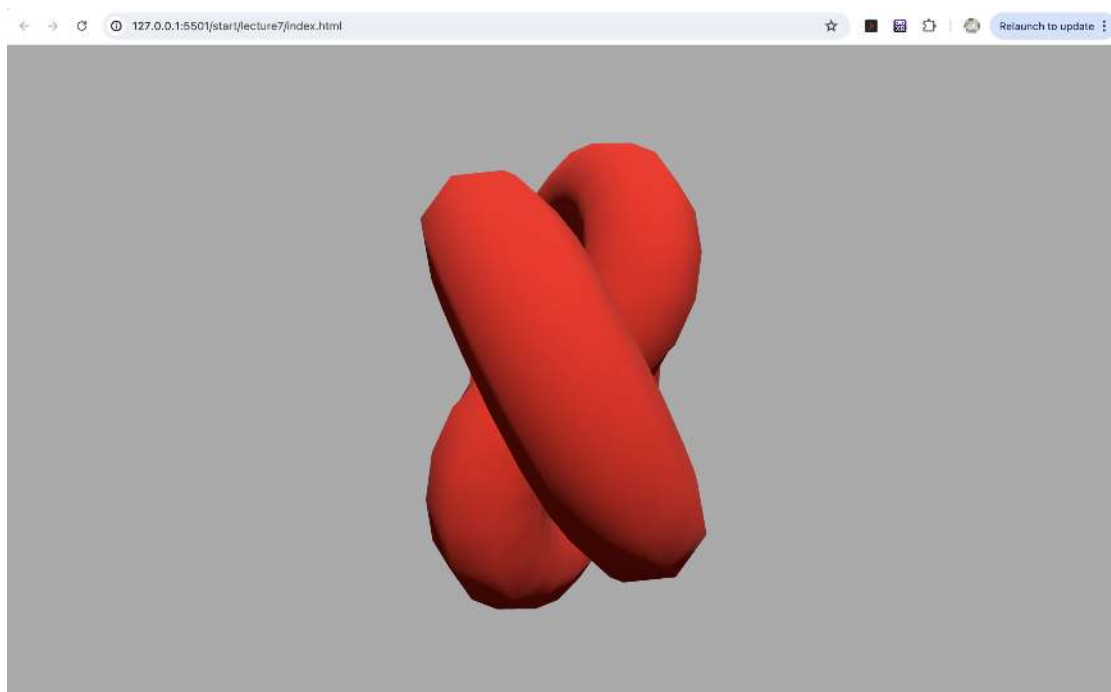
segments — Number of segments (triangles), minimum = 3, default = 32.

thetaStart — Start angle for first segment, default = 0 (three o'clock position).

thetaLength — The central angle, often called theta, of the circular sector. The default is 2 * Pi, which makes for a complete circle.

The first is the radius which defaults to 1, then the number of segments. So, entering 1, 64 will give a smoother circle. If you want a half circle, then enter 0 as the third parameter and `Math.PI` as the fourth. Parameters 3 and 4 give the starting and ending angles when generating a circle. By default, the starting angle is 0, which is the equivalent of the x axis and the default end angle is 2π which is again the x axis. 2π is a full revolution in radians. By setting parameter 4 as just π we get a half circle.

You might be wondering why the circle disappears as it rotates. By default, only front facing triangles are rendered by the renderer, so as the circle rotates and shows it's rear to the camera the renderer ignores these faces. Eventually it rotates around enough to start showing the front facing triangle again.



Caption: TorusKnot

Play around with the different geometries and check the documentation to customise them.

Once you've had a play, it's time to think about constructive geometries and we'll look at the **ExtrudeGeometry** class. This takes a **Shape** instance and extrudes it along the z axis. The class requires two parameters a **Shape** and a settings object.

Comment out the first geometry line and uncomment the other. Notice the **createStarGeometry** method is currently empty. We need to add some code. Enter:

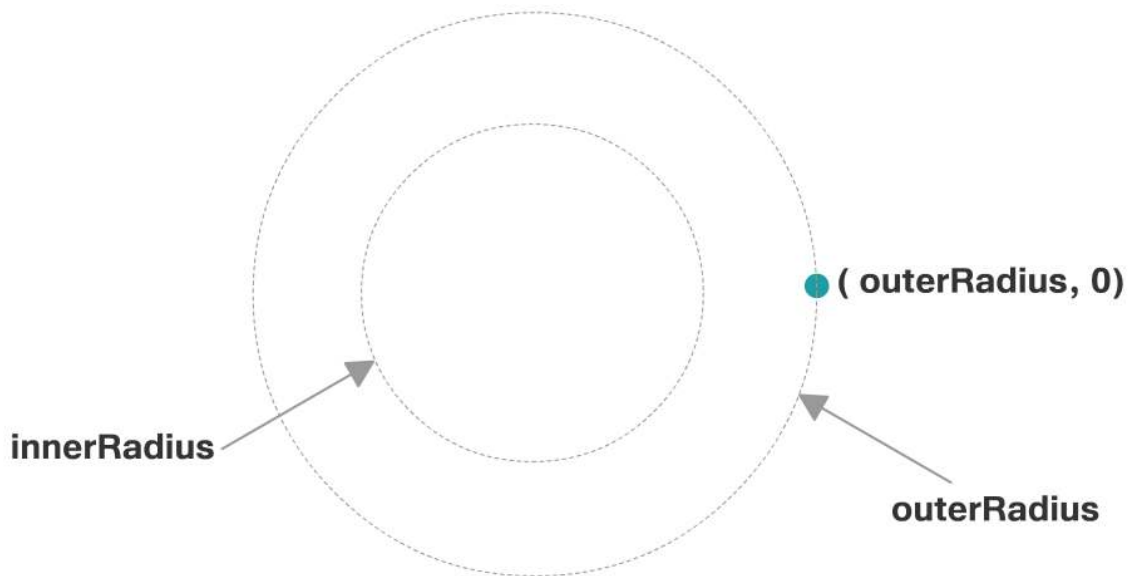
```
createStarGeometry(innerRadius=0.4, outerRadius=0.8, points=5){
  const shape = new THREE.Shape();
  const PI2 = Math.PI * 2;
  const inc = PI2/(points*2);
  shape.moveTo( outerRadius, 0 );
  let inner = true;

  for( let theta=inc; theta<PI2; theta += inc){
    const radius = (inner) ? innerRadius : outerRadius;
    shape.lineTo( Math.cos(theta)*radius, Math.sin(theta)*radius);
    inner = !inner;
  }

  const extrudeSettings = { steps: 1, depth: 1, bevelEnabled: false };
  return new THREE.ExtrudeGeometry( shape, extrudeSettings );
}
```

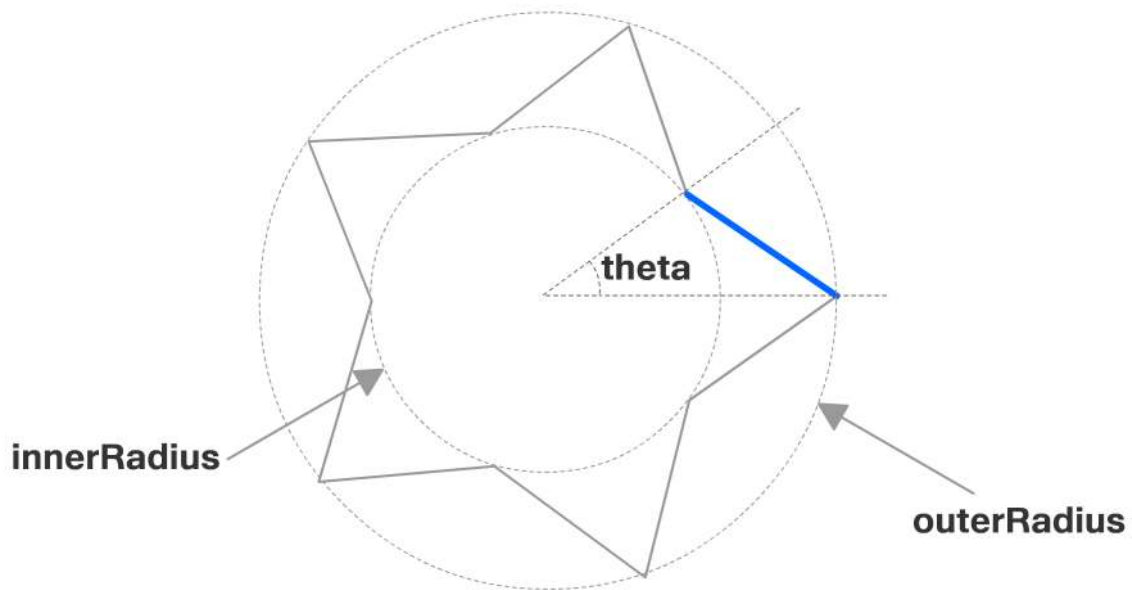
That will take some unpicking. First the **Shape**. A **Shape** instance has **moveTo** and **lineTo** methods. The idea is to draw a shape in the XY plane. Here we draw a five pointed star.

We define an **innerRadius** and **outerRadius** for the star and then set an **inc** value which is the angle to move by when drawing each line. Then we move the current point of the shape to (**outerRadius**, 0).



Caption: Creating a star shape 1

In a **for** loop we increment an angle **theta** until it exceeds 2π . In other words from the angle **inc** up to a complete revolution. We set a **radius** property alternating between **innerRadius** and **outerRadius** then draw a line to ($\cos \varnothing$, $\sin \varnothing$) times **radius**.

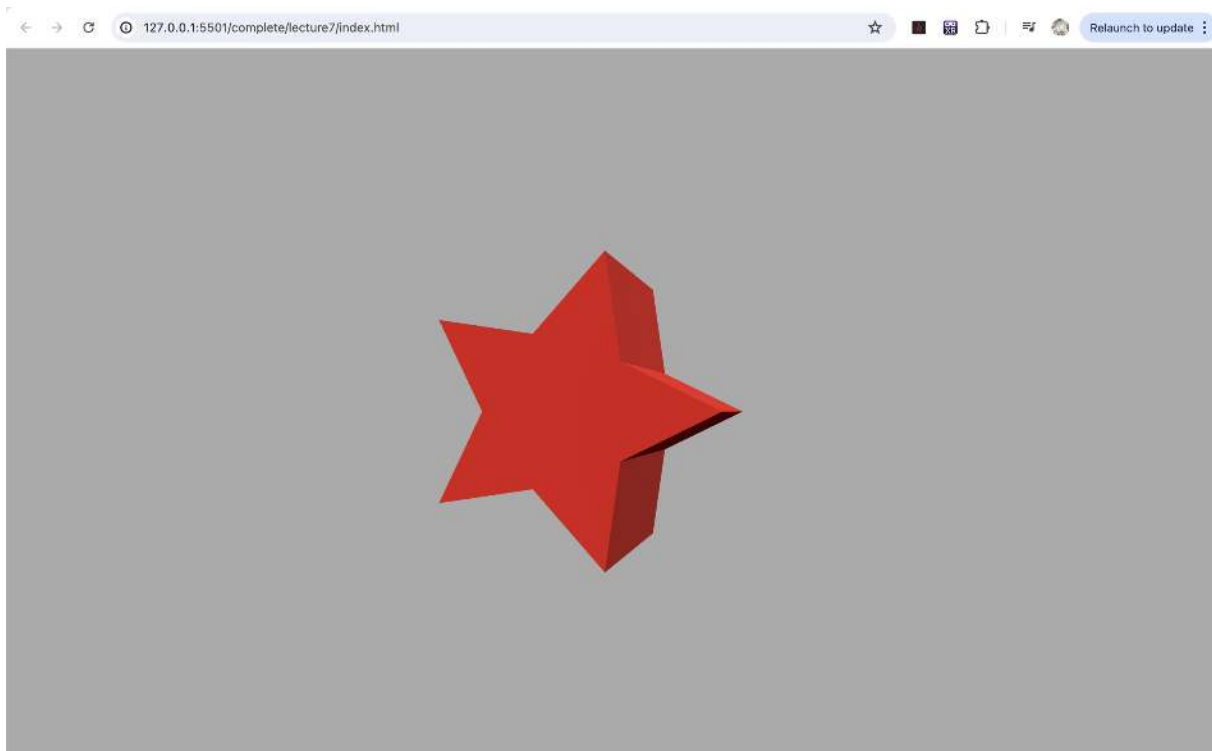


Caption: Creating a star shape 2

$(\cos \varnothing, \sin \varnothing)$ will be on the perimeter of a circle of radius 1, multiplying these values by the **innerRadius** or **outerRadius**, we first draw a line from the outer to the inner radius having rotated by **inc**.

Then we draw a line to outer radius having again rotated. Repeating this 10 times we have a star shape.

Then we define a simple **extrudeSettings** object, saying we want to span the entire length in a single step and that the depth of the extrusion is 1 unit. We set **bevel** to false. If you look at the [documentation](#) you can see how **bevel** would work. Now we have a rotating object that has a star shaped cross section.



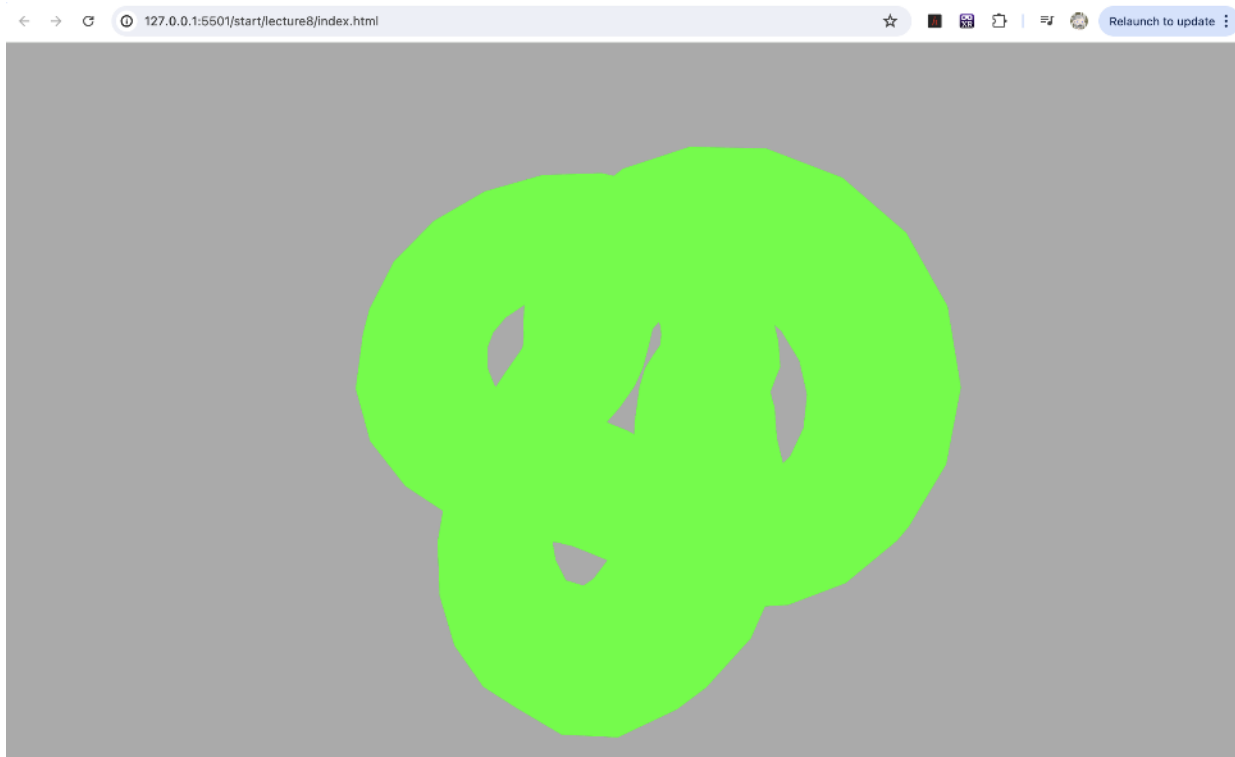
Caption: The extruded star shape

Go ahead and play with the other ways you can construct geometry using ThreeJS classes.

When you're ready it's time to play around with materials.

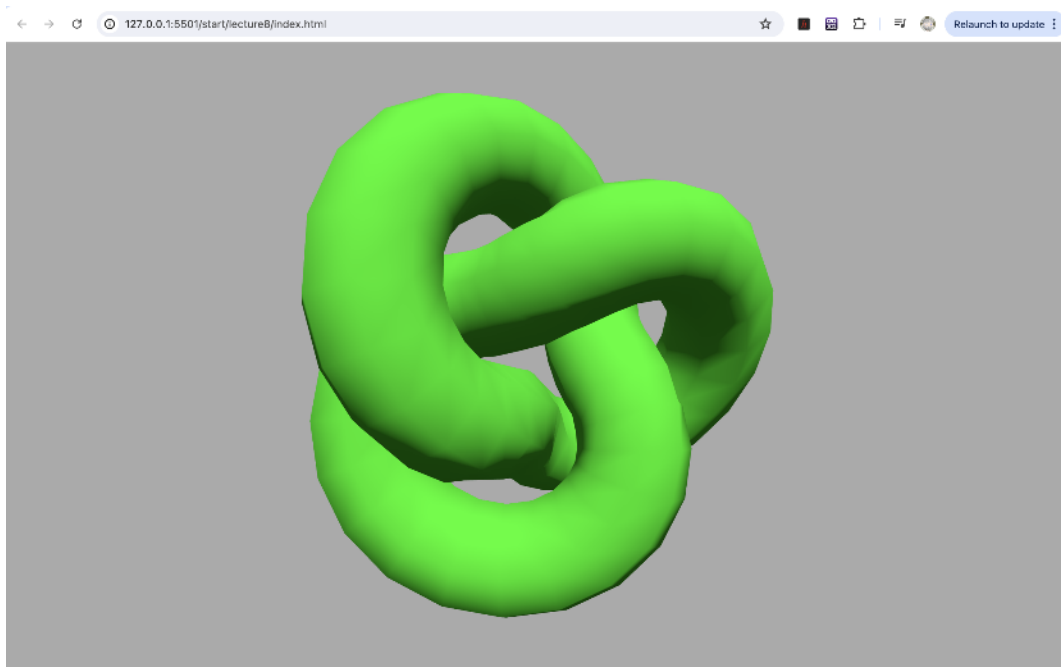
8. Materials

Each **Mesh** you create can have a different **Material** or share materials with other meshes. To code along with this video go to the folder `start/lecture8` and open the file `app.js`. The first material we'll use is **MeshBasicMaterial**, this type of material takes a colour value and paints every pixel where the mesh shows this colour. It takes no notice of lights in the scene. Try changing the colour value and reviewing the results. Remember colour values are a hexadecimal value. Where A represents 10, B – 11 up to F – 15. The first 2 digits in the value give the red component, the second 2 the green and the last 2 the blue. A value of FF is actually $15 \times 16 + 15$. `0xFF0000` means maximum red and minimum green and blue.



Caption: An example of using MeshBasicMaterial

The simplest material that supports lighting is **MeshLambertMaterial**. The Lambert model of lighting calculates the lighting at the vertex and then interpolates this value across the triangle that is being rendered. Regardless of the complexity of a mesh, it is actually made up of lots of triangles and a renderer works on each triangle when building up the final image. Because lighting calculations are at the vertex level, Lambert lighting cannot show specular, shiny bright points of light. To show the TorusKnot mesh using Lambert lighting simply change **MeshBasicMaterial** to **MeshLambertMaterial**. You don't need to do anything else.

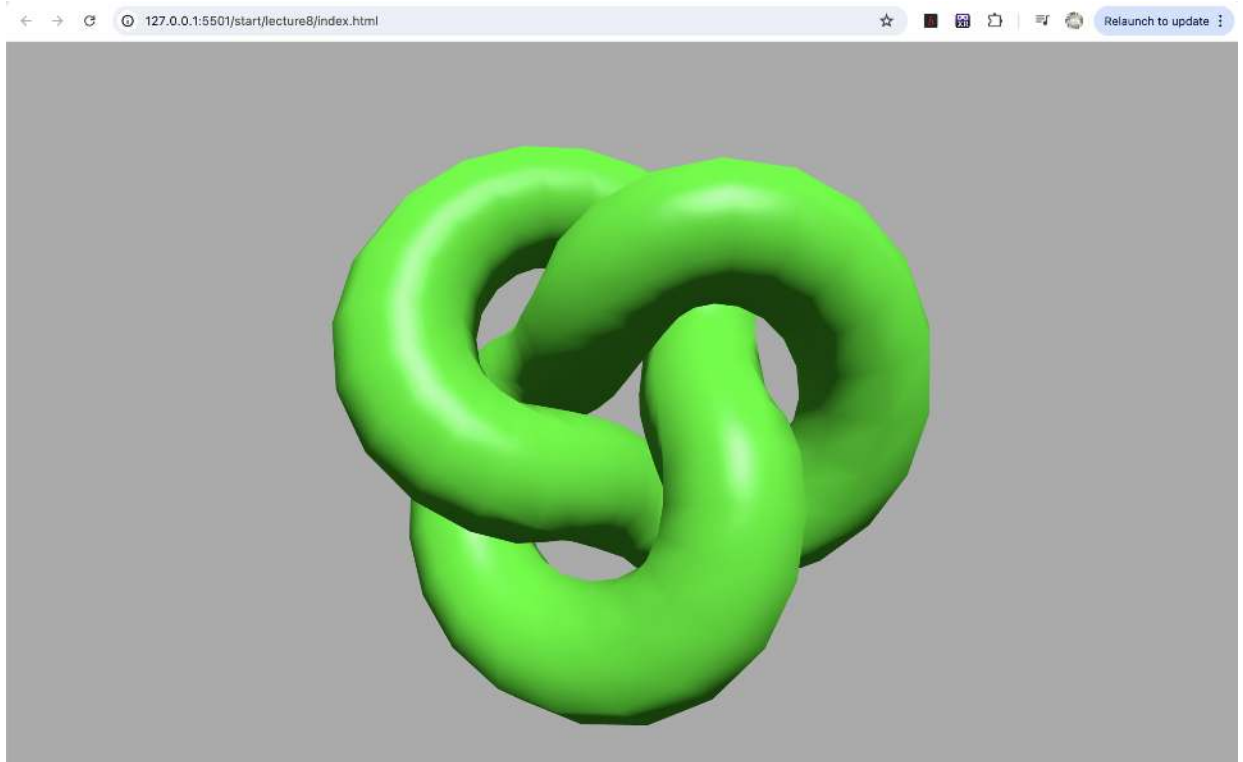


Caption: An example of using MeshLambertMaterial

If you want to see specular highlights, then it is time to use the **MeshPhongMaterial** class. Change Lambert to Phong. When using Phong shading, if you want a shiny appearance then you will need to add some options to the object passed to the constructor.

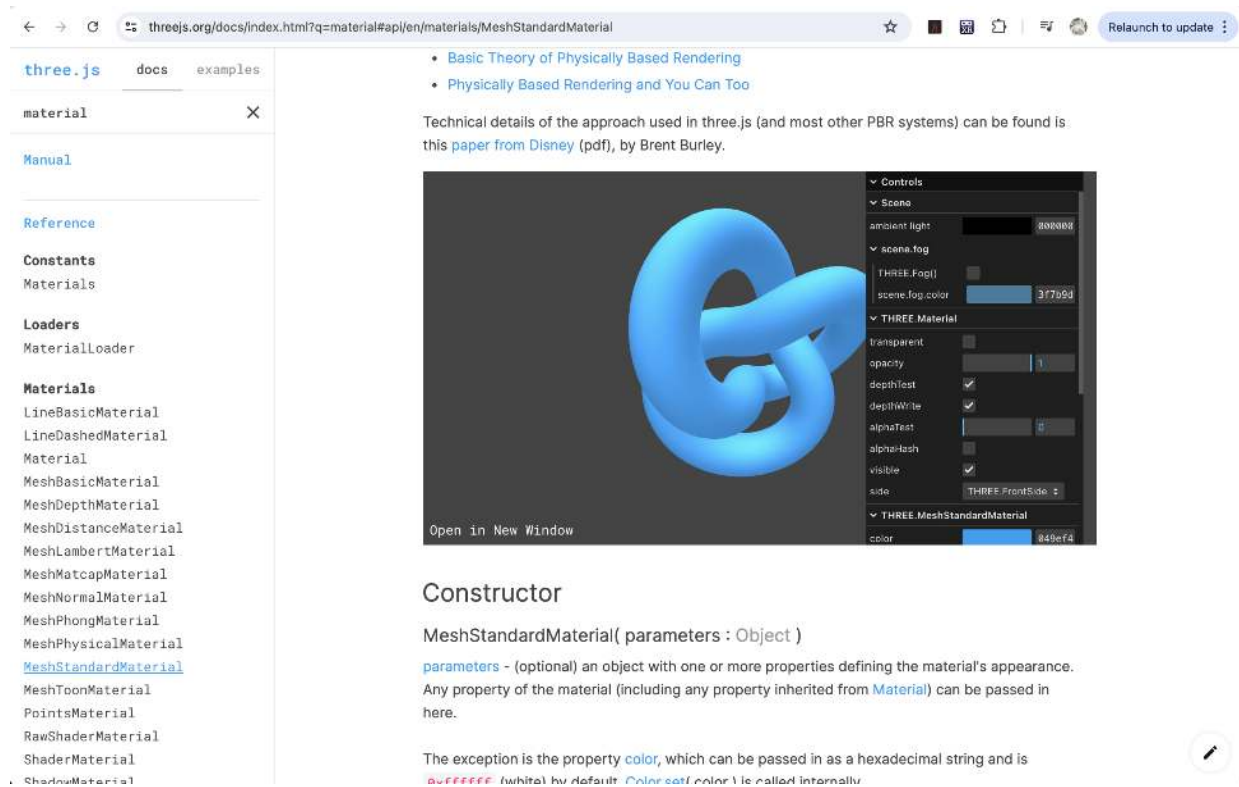
```
const material = new THREE.MeshPhongMaterial( { color: 0x00FF00, specular: 0x444444, shininess: 60 } );
```

Now the mesh appears shiny.



Caption: An example of using MeshPhongMaterial

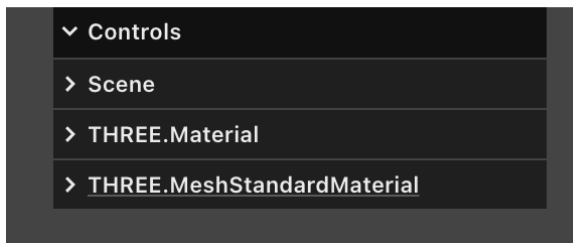
But the material that is most commonly used these days is the **MeshStandardMaterial**. This is very complex. To achieve a shiny result you need to adjust the roughness and metalness values. A great way to review the properties of the material classes is to enter material in the threejs.org documentation search box.



The screenshot shows the three.js documentation website. The search bar contains the word "material". The left sidebar lists various material classes, with [MeshStandardMaterial](#) highlighted. The main content area shows the title "MeshStandardMaterial" and a 3D viewer displaying a blue knot. To the right of the viewer is a dat.GUI control panel for the material, showing properties like "color" (hex #449ef4) and "roughness".

Caption: An example of using MeshStandardMaterial

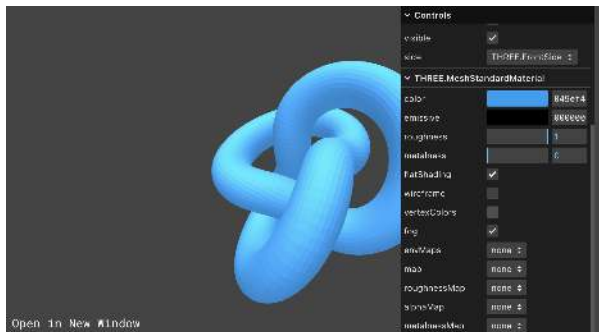
If we look at the page for **MeshStandardMaterial** by clicking on the name in the left column's search results list then we see a ThreeJS app running on the page that includes a dat.GUI panel allowing you to dynamically manipulate the properties of the material.



The panel is split between material and **MeshStandardMaterial**, this is not because the mesh has 2 materials. It is because the **MeshStandardMaterial** class extends the base class of **Material**. All

ThreeJS **Material** classes are extensions of the **Material** class and here you can see the properties that are common to them all.

Notice if you set **opacity** nothing happens until you set **transparent** to true. Check what happens with the **depthTest** and **depthWrite** options. Because a **TorusKnot** has triangles hidden by other triangles in the same **Mesh**. Using both **depthTest** and **depthWrite** is essential. **depthWrite** ensures each rendered pixel saves the distance from camera for that pixel to the depth buffer. **depthTest** decides whether when rendering a pixel to check the depth buffer and only paint the pixel if the current distance of this pixel from the camera is less than what is already saved in the buffer. Try changing the side option, notice that choosing **BackSide** shows the inside of the **Mesh**. Now open the **MeshStandardMaterial** panel and adjust **roughness** and



metalness to see how they behave.

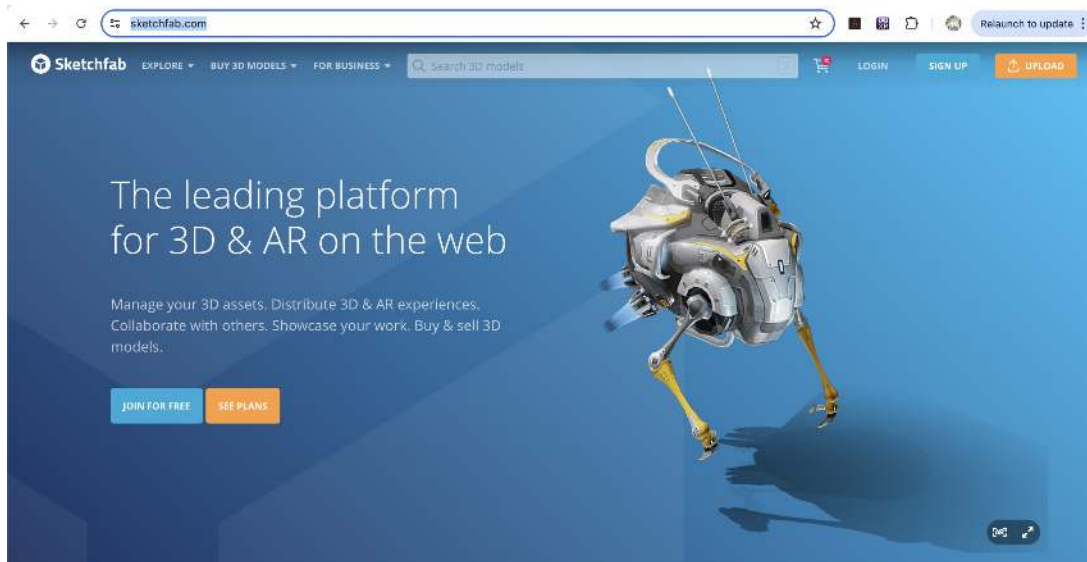
flatShading turns the smooth surface into a faceted one and wireframe changes the render to an outline version.

A **MeshStandardMaterial** can have several textures applied try playing with them. **envMaps** control how the surface reflects and refracts light. **Map** replaces the color with a texture, **roughnessMap** allows you to control the roughness of the surface using a texture. If using an **alphaMap**, a texture defining the transparency of the surface, remember to set **transparent** to true. Otherwise nothing will happen.

Using Geometries and Materials to create Meshes is at the heart of a ThreeJS scene. But sometimes primitive and constructive geometries are not enough and you will want to load a geometry that you have either created or found online. That's where loaders come in and we'll review them in the next chapter.

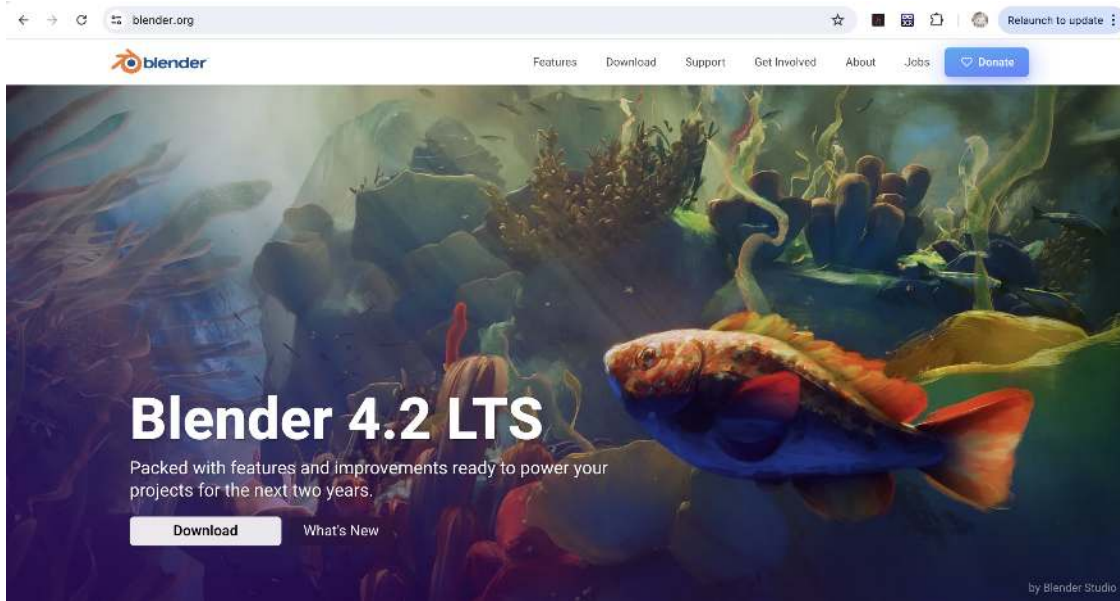
9. Loaders

Before we start to look at the code you'll use when loading complex objects or scenes. Let me first give a plug to the site where I got the assets for this video. [SketchFab](#) has lots of 3D models many of which are excellent.



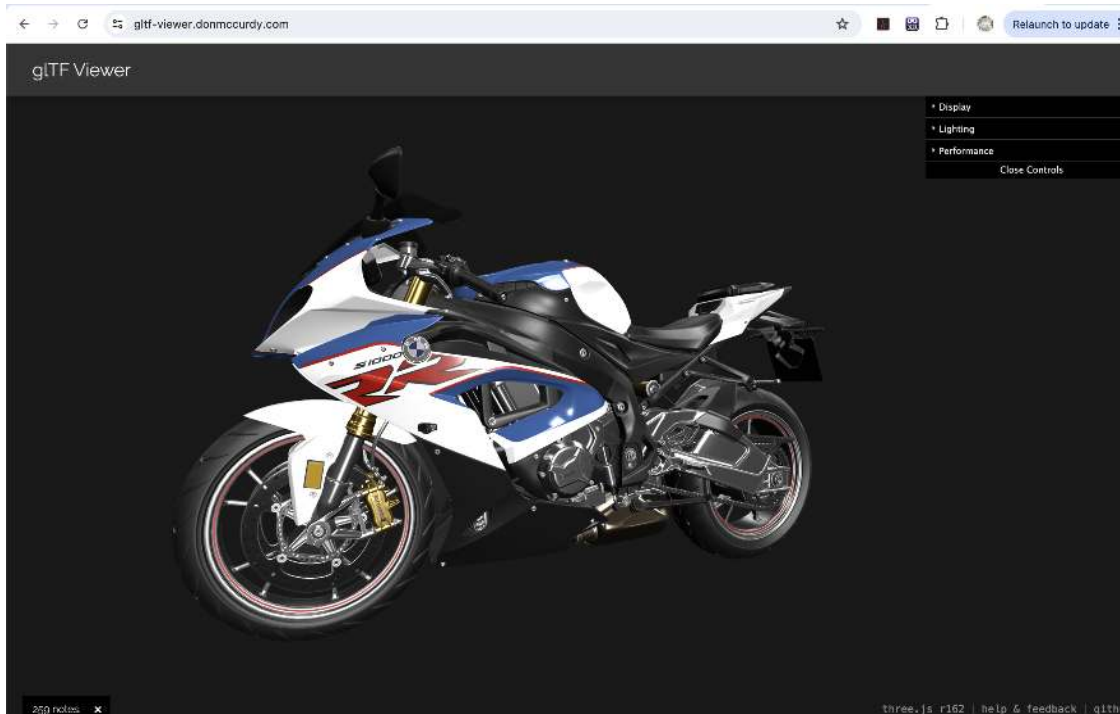
Caption: Sketchfab.com

3D assets come in many different formats and when working with ThreeJS one of the best formats to use is a glb file, a binary version of the [GLTF](#) format. This format is a relatively new one and to be honest you're really unlikely to come across assets in this format. [Blender](#) is a great tool to edit and export assets and it is super easy, when using Blender, to export an asset as a glb file, check-out my YouTube videos, link in the resources.



Caption: blender.org

Once you have a glb file I recommend using Don McCurdy's excellent online [glTF Viewer app](#). You simply drag your glb onto the screen to see it in its full glory. If the glb includes animation it will be playable, it is a very useful tool for ThreeJS developers.



Caption: The GLTF Viewer

To work along with this video open `start/lecture9/app.js`. This is the starting template. I've added the required imports, you'll need the **GLTFLoader** class. Other than that it is a boiler-plate template, creating a scene, renderer and camera and adding a couple of lights. Hopefully, this is starting to look quite familiar.

The first thing we'll do is add a progress bar. I created a simple class for this, and it is already added as an import. To use it simply add

```
this.loadingBar = new LoadingBar();
```

in the constructor method.

If you view the page now it will include a loading bar. It doesn't do anything, we'll need to add code in the loader to update the bar as the asset loads.

Now add

```
this.loadGLTF();
```

The app already includes an empty method called **loadGLTF**.

Let's add some code to the **loadGLTF** method. First we create an instance of a **GLTFLoader** and set the path it will use to find the assets.

```
const loader = new GLTFLoader().setPath('../assets/');
```

The asset we are loading is compressed to make the file size smaller, so we need to add a DRACOLoader. This class needs a path to the decoding library files. Now we have an instance of the DRACOLoader we use the set DRACOLoader method of the GLTFLoader instance to assign the draco loader.

```
const dracoLoader = new DRACOLoader();
dracoLoader.setDecoderPath( '../libs/three/examples/jsm/libs/draco/' );
loader.setDRACOLoader( dracoLoader );
```

The key method of a loader is the load method. This takes up to 4 parameters. Parameter 1 is the url of the file. Since I've added a path to the loader using the setPath method, this is just the file name. Then a callback when the file is loaded and parsed, parameter 3 is a progress callback and parameter 4 an error handler.

```
loader.load(
  // resource URL
  'motorcycle.glb',
  // called when the resource is loaded
  gltf => { },
  // called while loading is progressing
  xhr => { },
  // called when loading has errors
  err => { }
);
}
```

The onload callback, parameter 2, receives an object that contains several elements, the one we use here is the scene, adding this to the app scene object. Then we hide the loading bar and start a rendering loop. We set the gltf.scene property to the app property motorcycle so that it can be rotated in the render method.

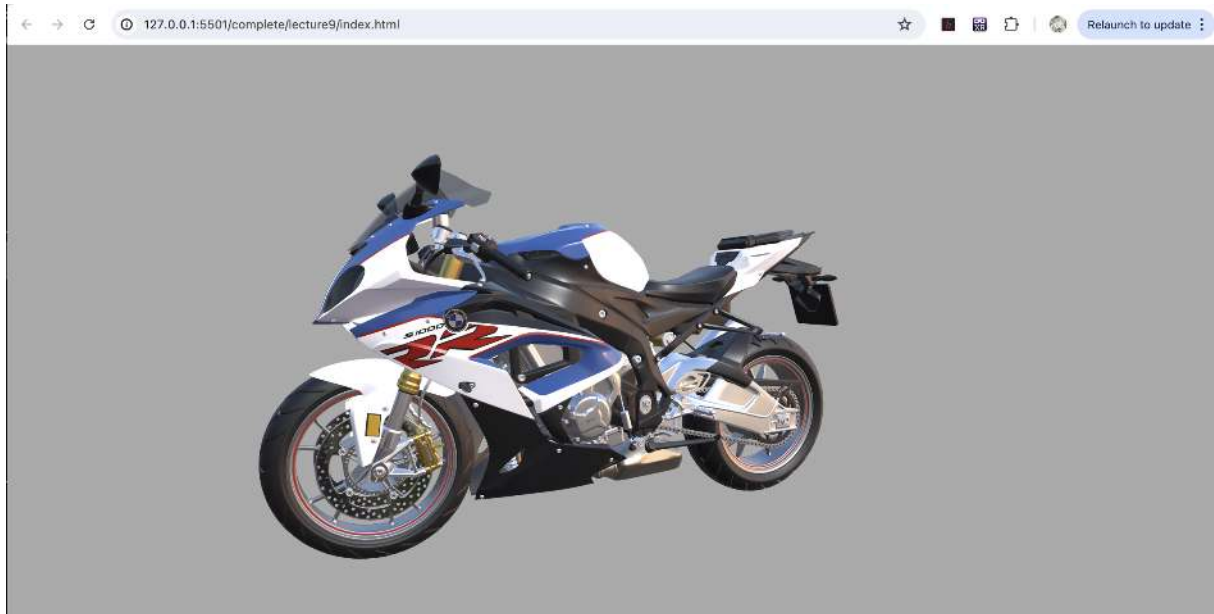
```
gltf => {
  this.motorcycle = gltf.scene;
  this.scene.add( gltf.scene );
  this.loadingBar.visible = false;
  this.renderer.setAnimationLoop( this.render.bind(this));
}
```

onProgress, parameter 3, receives a **XMLHttpRequest** object, which includes loaded and total values, by dividing loaded by total we get a value that ranges between 0 and 1 and is used to update the loading bar progress property. This sets the amount that the grey lozenge is filled by the blue bar. Indicating to the user the amount that has been loaded.

```
xhr => {
  this.loadingBar.progress = (xhr.loaded / xhr.total);
},
```

onError, parameter 4, just sends the err parameter to the console.

When loading a gltf asset the loader uses **MeshStandardMaterial** for the materials. Because of this you should the environment for the ThreeJS scene property. The method **setEnvironment** does this for you.



Caption: The loaded asset

And that's all there is to loading a complex asset. A word of warning though. The scale of your asset maybe very different to your expectation and if you can't see anything when you use this method with your own assets then it is time to start tracking down the problem.

At the start of the onLoad callback. Parameter 2 of the load method, enter:

```
const bbox = new THREE.Box3().setFromObject( gltf.scene );
console.log(`min:${bbox.min.x.toFixed(2)}, ${bbox.min.y.toFixed(2)},
${bbox.min.z.toFixed(2)} - max:${bbox.max.x.toFixed(2)},
${bbox.max.y.toFixed(2)},$ {bbox.max.z.toFixed(2)}`);
```

The **setFromObject** method of a **Box3**, calculates the axis aligned bounding box around an object and its children. It is useful way to get a rough feel for scale.

Once initialized a **Box3** instance has **min** and **max** properties. These are **Vector3** instances. We create a console log string using `toFixed(2)`, this rounds the value to just two decimal places. Making it much more readable. Save and refresh and we can see that in this example the values are in the range + or – 0.04 to 0.11, if you find yours are in the range + or – 10000, then the camera positioning needs to be radically changed or you need to use the **scale** property to scale the size of the `gltf.scene` object to a more suitable value.

In this chapter we used our first complex 3d asset. In the next we'll look at using assets that include animation.

10. Animation

The Three.js animation system is superb it echoes the animation system used by *Unreal* and *Unity*. In this video we're going to get a knight, moving. The knight was prepared using the 3D content creation tool *Blender* and the great online resource *Mixamo*, check [this link](#) to a video describing the technique on my [YouTube channel](#). The file *knight.glb* has four different animations and over the next few minutes I'll show you how to get the Three.js library to use them.



Caption: Editing a 3D asset using Blender

Let's start with the template open `app.js` from the folder `start/lecture10`. Essentially this is the same as the previous video. We setup a scene, camera and renderer. Because we're loading a `glb` file we load an environment map.

We'll be adding code in three places one in the onLoad callback for the glb file, two adding some code to a setter function and finally adding some code to the render loop.

First up the onLoad callback. At the heart of the animation system is the AnimationMixer class, updating this will cause any animations that have been added as actions to the mixer to move the desired part of a skinned mesh or just move an object. The AnimationMixer class takes a single parameter, the root object that any animations will be applied to. Here we set the knight object which is the gltf.scene.

```
this.mixer = new THREE.AnimationMixer( this.knight );
```

Eventually we want to be able to switch animations simply by using the setter function action. To allow for this we want to collect all the animations in the gltf file and store them as named properties of an object. We create an object and a names array then iterate over each animation in the gltf property animations. The GLTFLoader class parser creates an animations array as well as the scene property. We grab the name of each animation in the array, we use the name clip because the ThreeJS animation system thinks of these as instances of the AnimationClip class. To avoid issues over the case of the name string we convert it to lower case. Push it to the names array and store the clip as the name property in the animations object. Now we can access it using just the name. An alternative is to use the static method THREE.AnimationClip.findByName on the gltf animations array. But I prefer the hashed object route.

```
this.animations = {};  
  
const names = [];  
  
gltf.animations.forEach( clip => {  
  const name = clip.name.toLowerCase();  
  
  names.push(name);  
  
  this.animations[name] = clip;  
  
})  
  
console.log( `animations: ${names.join(',')}`);
```

To launch the first animation, we'll use the action setter we are yet to write. The animation we'll use first is look around.

```
this.action = 'look around';
```

Finally in this function we'll setup a simple ui to allow us to switch animation. The GUI class let's us do this very easily. Just create an options object, it needs a single property, name. Create an instance of GUI and add a control. The add method has many alternatives here we use three parameters: options, name, and the names array. By using the names array, GUI will populate a dropdown with each string from the names array as options. We also add an onChange handler, this callback will get the name string selected from the dropdown. All we need to do is set the action setter to this value and once we've added the code to this method it will blend the current animation to the newly selected one.


```
const options = { name: 'look around' };

const gui = new GUI();

gui.add(options, 'name', names).onChange( name => {

  this.action = name

});
```

Time to create the action code. In the code we're going to store the animation name that is currently playing as `actionName`. The first thing we'll do is check if we are trying to set the action to the one currently playing, if we are then we return from the method without doing anything.

```
set action(name){

  if (this.actionName == name.toLowerCase()) return;

  ...
```

Notice we consistently use the lower case version of the name. Now we get the animation from the `animations` object we created earlier by using the name. If this **AnimationClip** exists then we convert this into an **AnimationAction** using the **AnimationMixer** method `clipAction`.

```
const clip = this.animations[name.toLowerCase()];

if (clip!==undefined){

  const action = this.mixer.clipAction( clip );

  ...
```

An **AnimationAction** uses the **AnimationClip** as its data, but extends this to include time, the rate to play the **AnimationClip** and what to do about looping when the time property exceeds the duration of the clip. Using the mixer you can blend multiple **AnimationActions** and when doing this you can use the weight property of an **AnimationAction** to add emphasis to a clip. In this example we're only using a single action at a time.

By default an **AnimationAction** is set to loop indefinitely.

One of the animation actions is die and this should not loop, we check for the name 'die'. If we find this then we set `clampWhenFinished` to true and the loop property to `LoopOnce`. This ensures that the knight character stays on the floor when the die animation completes.

```
if (name=='die'){  
    action.clampWhenFinished = true;  
    action.setLoop( THREE.LoopOnce );  
}  
...
```

Since we want to be able to start and restart our animations we call the `reset` method. This method sets `paused` to false, `enabled` to true, `time` to 0, interrupts any scheduled fading and warping, and removes the internal loop count and scheduling for delayed starting.

```
action.reset();
```

```
...
```

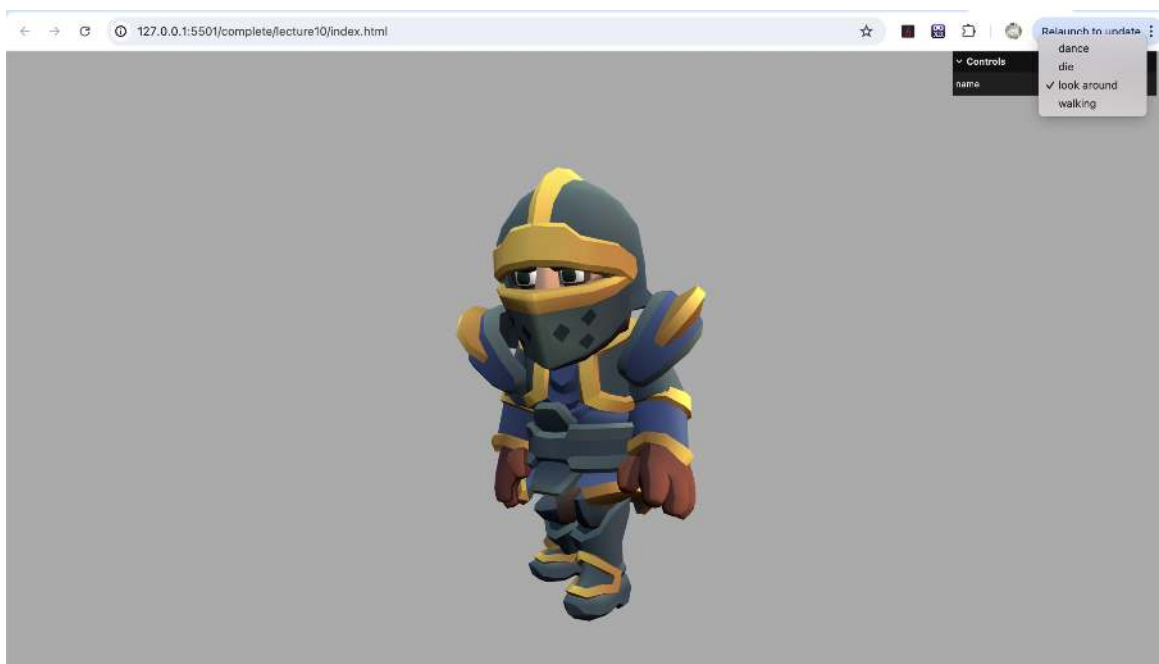
The Three.js animation system allows us to fade in an animation which we want to do as long as the current action is not die. If the knight is lay flat on the ground we want to simply switch to the new animation. We use the variable `nofade` to handle this. Having tested for the previous action we can now store the new action name and set the action to play. If we have a `curAction` and `nofade` is true then the `curAction` is disabled. If `nofade` is false then we set `curAction` to `crossFadeTo` the new action over half a second. Finally we store the new action as `curAction` so we can setup disabling or cross fades later.

```
const nofade = this.actionName !== 'die';
this.actionName = name.toLowerCase();
action.play();
if (this.curAction){
  if (nofade){
    this.curAction.enabled = false;
  }else{
    this.curAction.crossFadeTo(action, 0.5);
  }
}
this.curAction = action;
}
```

If you run the app now you'll be disappointed, no animation will show. We need to add a little code to the animation loop method, render.

The clock instance allows us to get the time that has elapsed since we last called `getDelta`. We need to tell our `AnimationMixer` that it needs to move any animation actions on by this timed amount. Easily done just call the update method passing the dt value.

```
if (this.mixer) this.mixer.update(dt);
```



Caption: The knight character animating

Now you see the knight animating and you can easily switch animations.

Try commenting out the `clampWhenFinished` line to see its effect and the `setLoop` line.

The Three.js animation system is very versatile.

That completes this short introduction to this great library. I look forward to seeing the apps you create now you know the basics. Thanks for completing the course.

II. Where to from here

Now you know the basics of using the ThreeJS library you might like to know I have a number of courses that can take you further on your journey.



The Beginners Guide to 3D Web Game Development with ThreeJS

Learn to write JavaScript code while having fun making 3D web games using the most popular Open Source WebGL library ThreeJS



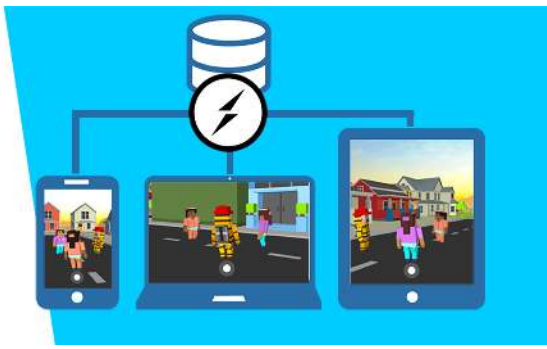
Learn to Create WebXR, VR and AR, experiences with ThreeJS

Learn how to create VR and AR experiences that work directly from the browser, using the WebXR and our favourite Open Source WebGL library,



Learn GLSL Shaders from Scratch

Learn how to harness the power of the GPU in your web pages by learning to code GLSL shaders.



Create a 3D Multi-Player Game using ThreeJS and SocketIO

Learn how to use nodeJS, socketIO and ThreeJS to create a 3d multi-player game



Create a 3D Car Racing Game with ThreeJS and CannonJS

Learn to combine the physics engine CannonJS and ThreeJS to create a fun car racing game
Create a 3D RPG Game with ThreeJS



Create a 3D RPG Game with ThreeJS

Learn how to harness the ThreeJS library to create a 3D RPG game

The Three.js Primer



Three.js is the most popular library for displaying 3D content in a browser, even a smartphone browser!

This e-book begins with a beginner-level primer to real-time 3D concepts and some basic examples to get you started with the most important features that Three.js has to offer.

- You'll find out how to use the **online Three.js Editor**.
- You'll learn about **geometry and materials**.
- How to **load a complex model** and how to **animate your models**.

This is a **quick introduction** to the most important features of the library. After completing the examples, you will have a basic understanding of how to use Three.js in your own Web Apps.