Getting started with Three.js

- 1. The basics
- 2. Support
- 3. Set the Scene
- 4. Making a Mesh
- 5. Materials
- 6. Lights!
- 7. Render
- 8. Common Object Properties
- 9. Dirty Little Secrets
- Conclusion

I have used Three.js for some of my experiments, and it does a really great job of abstracting away the headaches of getting going with 3D in the browser. With it you can create cameras, objects, lights, materials and more, and you have a choice of renderer, which means you can decide if you want your scene to be drawn using HTML 5's canvas, WebGL or SVG. And since it's open source you could even get involved with the project. But right now I'll focus on what I've learned by playing with it as an engine, and talk you through some of the basics.

For all the awesomeness of Three.js, there can be times where you might struggle. Typically you will need to spend quite a large amount of time with the examples, reverse engineering and (in my case certainly) hunting down specific functionality and occasionally asking questions via GitHub. If you have to as questions, by the way, I've found Mr. doob and AlteredQualia are extremely helpful!

1. The basics

I will assume that you have at least a passing knowledge of 3D, and reasonable proficiency with JavaScript. If you don't it may be worth learning a bit before you try and play with this stuff, as it can get a little confusing.

In our 3D world we will have some of the following, which I will guide you through the process of creating:

1. A scene
2. A renderer
3. A camera
4. An object or two (with materials)

You can, of course, do some cool things, and my hope is that you will go on to do that and start to experiment with 3D in your browser.

2. Support

Just a quick note on support in the browsers. Google's Chrome browser is, in my experience, the best browser to work with in terms of which renderers are supported, and the speed of the underlying JavaScript engine. Chrome supports Canvas, WebGL and SVG and it's blazingly fast. Firefox is a close second, with the advent of version 4. Its JavaScript engine does seem to be a touch slower than Chrome's, but again its support for the render technologies is great. Opera and Safari are in the process of adding WebGL support, but their current versions only support canvas. Internet Explorer (version 9+) support canvas rendering only, and I've not heard anything of Microsoft planning to add WebGL capabilities.

3. Set the Scene

I'll assume you've chosen a browser that supports all the rendering technologies, and that you want to render with canvas or WebGL, since they're the more standard choices. Canvas is more widely supported than WebGL, but it's worth noting that WebGL runs on your graphics card's GPU, which means that your CPU can concentrate on other non-rendering tasks like any physics or user interaction you're trying to do.

Irrespective of your chosen renderer you should bear in mind that the JavaScript will need to optimised for performance. 3D isn't a lightweight task for a browser (and it's awesome that it's even possible), so be careful to understand where any bottlenecks are in your code, and remove them if you can!

So with that said, and on the assumption you have downloaded and included three.js in your HTML file, how do you go about setting up a scene? Like this:

```
// set the scene size
var WIDTH = 400,
HEIGHT = 300;

// set some camera attributes
```

```
var VIEW_ANGLE = 45,
ASPECT = WIDTH / HEIGHT,
NEAR = 0.1,
FAR = 10000;

// get the DOM element to attach to
// - assume we've got jQuery to hand
var $container = $('#container');

// create a WebGL renderer, camera
// and a scene
var renderer = new THREE.WebGLRenderer();
var camera = new THREE.PerspectiveCamera(
                VIEW_ANGLE,
                ASPECT,
                NEAR,
                FAR );

var scene = new THREE.Scene();

// the camera starts at 0,0,0 so pull it back
camera.position.z = 300;

// start the renderer
renderer.setSize(WIDTH, HEIGHT);

// attach the render-supplied DOM element
$container.append(renderer.domElement);
```

Not too tricky, really!

4. Making a Mesh

So we have a scene, a camera and a renderer (I opted for a WebGL one in my sample code) but we have nothing to actually draw. Three.js actually comes with support for loading a few different standard file types, which is great if you are outputting models from Blender, Maya, Cinema4D or anything else. To keep things simple (this is about getting started after all!) I'll talk about primitives. Primitives are geometric meshes, relatively basic ones like Spheres, Planes, Cubes and Cylinders. Three.js lets you create these types of primitives easily:

```
// set up the sphere vars
var radius = 50, segments = 16, rings = 16;

// create a new mesh with sphere geometry -
// we will cover the sphereMaterial next!
var sphere = new THREE.Mesh(
new THREE.SphereGeometry(radius,
segments,
rings),

sphereMaterial);

// add the sphere to the scene
scene.add(sphere);
```

All good, but what about the material for the sphere? In the code we've used a variable **sphereMaterial** but we've not defined it yet. First we need to talk about materials in a bit more detail.

5. Materials

Without doubt this is one of the most useful parts of Three.js. It provides for you a number of common (and very handy) materials to apply to your meshes:

1. 'Basic' - which just means that it renders 'unlit'
2. Lambert
3. Phong

There are more, but again in the interests of simplicity I'll let you discover those for yourself. In the case of WebGL particularly these materials can be a life-saver. Why? Well because in WebGL you have to write shaders for everything being rendered. Shaders are a huge topic in themselves, but in short they are written in GLSL (OpenGL Shader Language), which tells the GPU how something should look. This means you need to mimic the maths of lighting, reflection and so on. It can get very complicated very quickly. Thanks to Three.js you don't have to do this if you don't want to because it abstracts that away for you. If you want to write shaders, however, you can do that too with a MeshShaderMaterial, so it's a flexible setup.

For now, however, let's apply a lambert material to the sphere:

```
// create the sphere's material
var sphereMaterial = new THREE.MeshLambertMaterial(
```

```
{
// a gorgeous red.
color: 0xCC0000
});
```

It's worth pointing out as well that there are other properties you can specify when you create a material besides the colour, such as smoothing or environment maps. You should check out the Wiki page for the various properties you can set on the materials and, in fact, any object that the engine provides for you. Also threejs.org recently sprung up, which offers a more attractive view of the API.

6. Lights!

If you were to render the scene right now you'd see a red circle. Even though we have a Lambert material applied there's no light in the scene so by default Three.js will revert to a full ambient light, which is the same as flat colouring. Let's fix that with a simple point of light:

```
// create a point light
var pointLight = new THREE.PointLight( 0xFFFFFF );

// set its position
pointLight.position.x = 10;
pointLight.position.y = 50;
pointLight.position.z = 130;

// add to the scene
scene.add(pointLight);
```

7. Render

We now actually have everything set up to render, remarkably. But we actually need to go ahead and do just that:

```
// draw!
renderer.render(scene, camera);
```

You're probably going to want to render more than once, though, so if you're going to do a loop you should really use requestAnimationFrame; it's by far the smartest way of handling animation in the browser. It's not fully supported as yet, so I'd totally recommend that you take a look at Paul Irish's shim.

8. Common Object Properties

If you take time to look through the code for Three.js you'll see a lot of objects 'inherit' from Object3D. This is a base object which contains some very useful properties, such as the **position**, **rotation** and **scale** information. In particular our Sphere is a Mesh which inherits from Object3D, to which it adds its own properties: **geometry** and **materials**. Why do I mention these? Well it's unlikely you're going to want to just have a sphere on your screen that does nothing, and these properties are worth investigating as they allow you to manipulate the underlying details of the meshes and materials on the fly.

```
// sphere geometry
sphere.geometry

// which contains the vertices and faces
sphere.geometry.vertices // an array
sphere.geometry.faces // also an array

// its position
sphere.position // has x, y and z properties
sphere.rotation // same
sphere.scale // ... same
```

9. Dirty Little Secrets

I just wanted to quickly point out a quick gotcha for Three.js, which is that if you modify, for example, the vertices of a mesh, you will notice in your render loop that nothing changes. Why? Well because Three.js (as far as I can tell) caches the data for a mesh as something of an optimisation. What you actually need to do is to flag to Three.js that something has changed so it can recalculate whatever it needs to. You do this with the following:

```
// changes to the vertices
sphere.geometry.__dirtyVertices = true;
```

```
// changes to the normals
sphere.geometry.__dirtyNormals = true;
```

Again there are more, but those two I've found are the most useful. You should obviously only flag the things that have changed to avoid unnecessary calculations.

Conclusion

Well I hope you've found this brief introduction to Three.js helpful. There's nothing quite like actually getting your hands dirty and trying something, and I can't recommend it highly enough. 3D running natively in the browser is a lot of fun, and using an engine like Three.js takes away a lot of the headaches for you and lets you get to making some seriously cool stuff. To help you out a bit I've wrapped up the source code in this lab article, so you can use that as a reference. If you've enjoyed this let me know via Twitter, it's always good to say hello!

□□□ #3
□ □□□ □□□ 2026-10-31 22:21:30 UTC
□ cocobook-sync □□□ 2026-11-04 17:38:15 UTC