Chapter No. 1
"Creating Your First 3D Scene with Three.js"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.1 "Creating Your First 3D Scene with Three.js"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Jos Dirksen has worked as a Software Developer and Architect for more than a decade. He has much experience in a large range of technologies ranging from backend technologies, such as Java and Scala, to frontend development using HTML5, CSS, and JavaScript. Besides working with these technologies, Jos also regularly speaks at conferences and likes to write about new and interesting technologies on his blog. He also likes to experiment with new technologies and see how they can best be used to create beautiful data visualizations, the results of which you can see on his blog at http://www.smartjava.org/.

Jos is currently working as an Enterprise Architect for Malmberg, a large Dutch publisher of educational material. He is helping to create a new digital platform for the creation and publishing of educational content for primary, secondary, and vocational education.

For More Information:
Previously, Jos has worked in many different roles in the private and public sectors, ranging from private companies such as Philips and ASML to organizations in the public sector, such as the Department of Defense.

Besides his interest in frontend JavaScript and HTML5 technologies, he is also interested in backend service development using REST and traditional web service technologies. Jos has already written two books on this subject. He is the coauthor of the Open Source ESBs in action book that was published in 2008, and in 2012 he published a book on how to apply SOA Governance in a practical manner. This book is titled SOA Governance in Action.
Learning Three.js: The JavaScript 3D Library for WebGL

In the last couple of years, browsers have gotten more powerful and are capable platforms to deliver complex applications and graphics. Most of this, though, is standard 2D graphics. Most modern browsers have adopted WebGL, which allows you to not only create 2D applications and graphics in the browser, but also create beautiful and good performing 3D applications, using the capabilities of the GPU.

Programming WebGL directly, however, is very complex. You need to know the inner details of WebGL and learn a complex shader language to get the most out of WebGL. The Three.js library provides a very easy-to-use JavaScript API based on the features of WebGL, so that you can create beautiful 3D graphics, without having to learn the WebGL details.

The Three.js library provides a large number of features and APIs that you can use to create 3D scenes directly in your browser. In this book you'll learn all the different APIs that the Three.js library has to offer through lots of interactive examples and code samples.

What This Book Covers

Chapter 1, Creating Your First 3D Scene with Three.js, covers the basic steps that you need to take to get started with the Three.js library. You'll immediately create your first Three.js scene and at the end of this chapter, you'll be able to create and animate your first 3D scene directly in the browser.

Chapter 2, Working with the Basic Components That Make Up a Three.js Scene, explains the basic components that you need to understand while working with the Three.js library. You'll learn about lights, meshes, geometries, materials, and cameras. In this chapter you will also get an overview of the different lights that the Three.js library provides and the cameras you can use in your scene.

Chapter 3, Working with the Different Light Sources Available in Three.js, dives deeper into the different lights that you can use in your scene. It shows examples and explains how to use a SpotLight, DirectionLight, AmbientLight, PointLight, HemisphereLight, and AreaLight sources. Additionally, it also shows how to apply a LensFlare effect on your light source.

For More Information:
Chapter 4, *Working with the Three.js Materials*, talks about the materials available in the Three.js library that you can use in your meshes. It shows all the properties that you can set to configure the materials for your specific use and provides interactive examples to experiment with the materials that are available in the Three.js library.

Chapter 5, *Learning to Work with Geometries*, is the first of two chapters that explores all the geometries that are provided by the Three.js library. In this chapter you'll learn how to create and configure geometries in Three.js and you can experiment by using the provided interactive examples, with geometries such as plane, circle, shape, cube, sphere, cylinder, Torus, TorusKnot, and Polyhedron.

Chapter 6, *Using Advanced Geometries and Binary Operations*, continues where Chapter 5 left off. It shows you how to configure and use the more advanced geometries that are provided by the Three.js library such as Convex and Lathe. In this chapter you'll also learn how to extrude 3D geometries from the 2D shapes and how you can create new geometries by combining geometries using binary operations.

Chapter 7, *Particles and the Particle System*, explains how to use the particle system from the Three.js library. You'll learn how to create a particle system from scratch, and from the existing geometries. In this chapter you'll also learn how you can modify the way the particles look through the use of sprites and particle materials.

Chapter 8, *Creating and Loading Advanced Meshes and Geometries*, shows you how to import meshes and geometries from external sources. You'll learn how to use the Three.js library internal JSON format in order to save the geometries and scenes. This chapter also explains how to load models from formats like OBJ, DAE, STL, CTM, and PLY.

Chapter 9, *Animations and Moving the Camera*, explores the various types of animations that you can use to make your scene come to life. You'll learn how to use the Tween.js library together with Three.js, and you'll learn how to work with the animation models based on morphs and skeletons.

Chapter 10, *Loading and Working with Textures*, expands on Chapter 4 where materials were introduced. In this chapter we will dive into the details of textures. It introduces the various types of textures that are available and how you can control a texture that is applied to your mesh. Additionally in this chapter, you are shown how you can directly use the output from the HTML5 video and canvas elements as input for your textures.

For More Information:
Chapter 11, Custom Shaders and Render Post Processing, explores how you can use the Three.js library to apply the post processing effects to your rendered scene. With post processing you can apply effects, such as blur, tilt shift, and sepia to your rendered scene. Besides this, you'll also learn how to create your own post processing effect and create a custom vertex and fragment shader.

Chapter 12, Adding Physics to Your Scene with Physijs, explains how you can add physics to your Three.js scene. With physics, you can detect collisions between objects, make them respond to gravity, and apply friction. This chapter shows how to do so with the Physics JavaScript library.

For More Information:
Creating Your First 3D Scene with Three.js

Modern browsers are slowly getting more powerful features that can be accessed directly from JavaScript. You can easily add video and audio with the new HTML5 tags and create interactive components through the use of the HTML5 canvas. A rather new addition to this feature set is the support of WebGL. With WebGL you can directly make use of the processing resources of your graphics card and create high-performance 2D and 3D computer graphics. Programming WebGL directly from JavaScript to create and animate 3D scenes is a very complex and error-prone process. Three.js is a library that makes this a lot easier. The following list shows some of the things that Three.js makes easy:

- Creating simple and complex 3D geometries
- Animating and moving objects through a 3D scene
- Applying textures and materials to your objects
- Loading objects from 3D modeling software
- Creating 2D sprite-based graphics

For More Information:
Creating Your First 3D Scene with Three.js

With a couple lines of JavaScript you can create anything from simple 3D models to photorealistic real-time scenes as shown:

In the first chapter, we'll directly dive into Three.js and create a couple of examples that show you how Three.js works and you can use them to play around with. We won't dive into all the technical details yet; that's something you'll learn in the later chapters. In this chapter we'll cover the following points:

- Tools required for working with Three.js
- Downloading the source code and examples used in this book
- Creating your first Three.js scene
- Improving the first scene with materials, lights, and animations
- Introducing a couple of helper libraries for statistics and controlling the scene

We'll start this book with a short introduction into Three.js and then quickly move on to the first examples and code samples. Before we get started, let's quickly look at the most important browsers out there and their support for WebGL.

For More Information:
At the moment Three.js works with the following browsers:

<table>
<thead>
<tr>
<th>Browser</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mozilla Firefox</td>
<td>Supported since Version 4.0.</td>
</tr>
<tr>
<td>Google Chrome</td>
<td>Supported since Version 9.</td>
</tr>
<tr>
<td>Safari</td>
<td>Supported since Version 5.1 and newly installed on Mac OS X Mountain Lion, Lion, and Snow Leopard. Make sure you enable WebGL in Safari. You can do this by navigating to Preferences</td>
</tr>
<tr>
<td>Opera</td>
<td>Supported since Version 12.00. You still have to enable this by opening the file opera:config and setting the value of WebGL and Enable Hardware Acceleration to 1. After that, restart the browser.</td>
</tr>
<tr>
<td>Internet Explorer</td>
<td>Internet Explorer had long been the only major player who didn't support WebGL. Starting with IE11, Microsoft has added WebGL support.</td>
</tr>
</tbody>
</table>

Basically, Three.js runs in any of the modern browsers, except most versions of IE. So if you want to use an older version of IE, you've got two options: you can get WebGL support through the use of Google Chrome Frame, which you can download from the following URL: https://developers.google.com/chrome/chrome-frame/. An alternative you can use instead of Google Chrome Frame is the iewebgl plugin, which you can get from http://iewebgl.com/. This installs inside IE and enables WebGL.

Google has officially dropped support for Google Chrome Frame and it doesn't support IE10.

Currently the guys behind Three.js are working on a renderer that uses the new CSS-3D specification, which is supported by a lot of browsers (even IE10). Besides desktop browsers, a number of mobile and tablet browsers also support CSS-3D.

For More Information:
In this chapter, you'll directly create your first 3D scene and will be able to run this in any of the mentioned browsers. We won't introduce too many complex Three.js features, but at the end of this chapter you'll have created the Three.js scene that you can see in the following screenshot:

For this first scene you'll learn about the basics of Three.js and also create your first animation. Before you start your work on this example, in the next couple of sections we'll first look at the tools that you need to easily work with Three.js and how you can download the examples that are shown in this book.
Chapter 1

Requirements for using Three.js

Three.js is a JavaScript library, so all that you need to create Three.js WebGL applications is a text editor and one of the supported browsers to render the results. I do like to recommend three JavaScript editors, which I’ve started using exclusively over the last couple of years:

- **WebStorm**: This editor from the JetBrains guides has great support for editing JavaScript. It supports code completion, automatic deployment, and JavaScript debugging directly from the editor. Besides this, WebStorm has excellent Git support (and other Version Control Systems). You can download a trial edition from [http://www.jetbrains.com/webstorm/](http://www.jetbrains.com/webstorm/)

- **Notepad++**: This is a general purpose editor that supports a wide range of programming languages. It can easily lay out and format JavaScript. Notepad++ is only for Windows.

- **Sublime Text Editor**: This is a great little editor that has very good support for editing JavaScript. Besides this, it provides many very helpful selection and edit options, which once you get used to, provide a real good JavaScript editing environment. Sublime Text Editor can also be tested for free and can be downloaded from [http://www.sublimetext.com/](http://www.sublimetext.com/)

Even if you don't use these three editors there are a lot of editors available, open source and commercial, which you can use to edit JavaScript and create your Three.js projects. An interesting project that you might want to look into is [http://c9.io](http://c9.io). This is a cloud-based JavaScript editor that can be connected to a GitHub account. This way you can directly access all the source code and examples from this book, and experiment with them.

For More Information:  
I had mentioned that most modern web browsers support WebGL and can be used to run the Three.js examples. I usually run my code in Chrome. The reason is that, most often, Chrome has the best support and performance for WebGL and it has a really great JavaScript debugger. With this debugger you can quickly pinpoint problems, for instance, by using breakpoints and console output. Throughout this book I'll give you pointers on debugger usage and other debugging tips and tricks.

That's enough of an introduction for now; let's get the source code and start with the first scene.

**Getting the source code**

All the code for this book can be accessed from GitHub [https://github.com/](https://github.com/). GitHub is an online Git-based repository that you can use to store, access, and version source code. There are a couple of ways you can get the sources for yourself:

- Clone the Git-based repository
- Download and extract the archive

In the following sections, we'll explore these options in more detail.
Using Git to clone the repository

Git is an open source distributed Version Control System that I have used to create and version all the examples in this book. For this I've used GitHub, a free, online Git-based repository. You can browse this repository by following this link: https://github.com/josdirksen/learning-threejs

To get all the examples you can clone this repository using the `git` command line tool. To do this, you first need to download a Git client for your operating system. For most modern operating systems, a client can be downloaded from http://git-scm.com or you can use the one provided by GitHub itself (for Mac and Windows). After installing Git, you can use this to get a clone of this book's repository. Open a command prompt and go to the directory where you want to download the sources. In that directory, run the following command:

`git clone https://github.com/josdirksen/learning-threejs`

This will start downloading all the examples as shown in the following screenshot:

![Screenshot of git clone process](image)

The directory `learning-three.js` will now contain all the examples that are used throughout this book.

For More Information:
Creating Your First 3D Scene with Three.js

Downloading and extracting the archive

If you don't want to use Git to download the sources directly from GitHub, you can also download an archive. Go to the URL https://github.com/josdirksen/learning-threejs and click on the download link as shown in the following screenshot:

![GitHub repository screenshot](image.png)

Extract this to a directory of your choice, and you'll have all the examples available.

Testing the examples

Now that you've downloaded or cloned the source code, let's do a quick check to see if everything is working and make you familiar with the directory structure. The code and examples are organized per chapter. There are two different ways of viewing the examples. You can either open the extracted or cloned directory in a browser directly and run a specific example, or you can install a local web server. The first approach will work for most of the basic examples, but when we start loading external resources such as models or texture images, just opening the HTML file isn't enough. In this case we need a local web server to make sure that the external resources are loaded correctly. In the following section, we will discuss a couple of different ways you can set up a simple local web server for testing.

Setting up a local web server is very easy, depending on what you've already got installed. We will list a couple of examples on how to do this.

For More Information:
Python-based approach should work on most Unix/Mac systems

Most Unix/Linux/Mac systems already have Python installed in them. On those systems you can very easily start a local web server:

```
> python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

Do this in the directory where you have checked out/downloaded the source code.

NPM-based approach if you've got Node.js installed

If you've already done some work with Node.js, there is a good chance that you've got NPM installed. With NPM you've got two simple options to set up a quick local web server for testing. The first one uses the HTTP Server module:

```
> npm install -g http-server
> http-server
Starting up http-server, serving ./ on port: 8080
Hit CTRL-C to stop the server
```

Alternatively you can also use the Simple HTTP Server option:

```
> npm install -g simple-http-server
> nserver
```

A disadvantage of this second approach, however, is that it doesn't automatically show the directory listings, whereas the first approach does.

Portable version of Mongoose for Mac/Windows

If you haven't got Python or NPM installed, there is a simple, portable web server, named Mongoose, that you can use. First download the binaries for your specific platform from the following URL: https://code.google.com/p/mongoose/downloads/list. On the Windows platform, copy the downloaded file to the directory containing the examples and double-click on it to start a web browser showing the contents of the directory it is started in.

For More Information:
For other operating systems, you must also copy the executable to the target directory, but instead of double-clicking on the executable you have to launch it from the command line. In both cases, a local web server will be started on port 8080.

By just clicking on a chapter, we can show and access all the examples for that chapter. If I discuss an example in this book, I'll refer to the specific name and folder so that you can directly test and play around with the code.

At this point you should have an editor installed and have access to all the sources. Now it is time to start creating our first Three.js scene.

Creating an HTML skeleton page

The first thing we need to do is create an empty HTML skeleton page that we can use as the base for all our examples. This HTML skeleton is shown as follows:

```html
<!DOCTYPE html>
<html>
<head>
<title>Example 01.01 - Basic skeleton</title>
<script type="text/javascript">
```

For More Information:
src="../libs/three.js"></script>
<script type="text/javascript"
    src="../libs/jquery-1.9.0.js"></script>
<style>
    body{
        /* set margin to 0 and overflow to hidden,
        to use the complete page */

        margin: 0;
        overflow: hidden;
    }
</style>
</head>
<body>

<!-- Div which will hold the Output -->
<div id="WebGL-output">
</div>

<!-- Javascript code that runs our Three.js examples -->
<script type="text/javascript">
    // once everything is loaded, we run our Three.js stuff.
    $(function () {
        // here we'll put the Three.js stuff
    });
</script>
</body>
</html>

---

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
As you can see from this listing, the skeleton is a very simple HTML page, with only a couple of elements. In the <head> element, we will load the external JavaScript libraries that we'll use for the examples. For all the examples, we'll at least load the two mentioned in this listing: Three.js and jquery-1.9.0.js. In the <head> element, we also add a couple of lines of CSS. These style elements remove any scroll bars when we create a full page Three.js scene. In the <body> of this page you can see a single <div> element. When we write our Three.js code, we'll point the output of the Three.js renderer to that element. In the previous code snippet, you can already see a bit of JavaScript. That small piece of code uses jQuery to call an anonymous JavaScript function when the complete page is loaded. We'll put all the Three.js code inside this anonymous function.

Three.js comes in two versions:

- **Three.min.js**: This is the library you'd normally use when deploying Three.js sites on the internet. This is a minimized version of Three.js, created using UglifyJS, which is half the size of the normal Three.js library. All the examples and code used in this book are based on the Three.js r60 project, which was released in August 2013.

- **Three.js**: This is the normal Three.js library. We will use this library in our examples, since it makes debugging much easier when you can read and understand the Three.js source code.

If we view this page in our browser, the results aren't very shocking. As you'd expect, all that you would see is an empty page:

In the next section, you'll learn how to add the first couple of 3D objects and render those to the <div> element that we had defined in our HTML skeleton page.

---

**For More Information:**
Rendering and viewing a 3D object

In this step you'll create your first scene and add a couple of objects and a camera. Our first example will contain the following objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plane</td>
<td>A two-dimensional rectangle that serves as our ground area. This is rendered as the gray rectangle in the middle of the scene.</td>
</tr>
<tr>
<td>Cube</td>
<td>A three-dimensional cube, which we'll render in red</td>
</tr>
<tr>
<td>Sphere</td>
<td>A three-dimensional sphere, which we'll render in blue</td>
</tr>
<tr>
<td>Camera</td>
<td>The camera determines what you'll see in the output</td>
</tr>
<tr>
<td>Axes</td>
<td>x, y, and z axes. This is a helpful debugging tool to see where the objects are rendered.</td>
</tr>
</tbody>
</table>

I'll first show you how this looks in code (the source file with comments can be found in the chapter-01 folder and is labeled 02-first-scene.html) and then I'll explain what's happening:

```javascript
$(function () {
    var scene = new THREE.Scene();

    var camera = new THREE.PerspectiveCamera(45,
                                           window.innerWidth / window.innerHeight,
                                           0.1, 1000);

    var renderer = new THREE.WebGLRenderer();
    renderer.setClearColorHex(0xEEEEEE);  
    renderer.setSize(window.innerWidth, window.innerHeight);

    var axes = new THREE.AxisHelper(20);
    scene.add(axes);

    var planeGeometry = new THREE.PlaneGeometry(60,20,1,1);
    var planeMaterial = new THREE.MeshBasicMaterial({color: 0xcccccc});
    var plane = new THREE.Mesh(planeGeometry, planeMaterial);

    plane.rotation.x=-0.5*Math.PI;
});
```

For More Information:
Creating Your First 3D Scene with Three.js

```javascript
plane.position.x = 15;
plane.position.y = 0;
plane.position.z = 0;

scene.add(plane);

var cubeGeometry = new THREE.CubeGeometry(4, 4, 4);
var cubeMaterial = new THREE.MeshBasicMaterial(
  {color: 0xff0000, wireframe: true});
var cube = new THREE.Mesh(cubeGeometry, cubeMaterial);

cube.position.x = -4;
cube.position.y = 3;
cube.position.z = 0;

scene.add(cube);

var sphereGeometry = new THREE.SphereGeometry(4, 20, 20);
var sphereMaterial = new THREE.MeshBasicMaterial(
  {color: 0x7777ff, wireframe: true});
var sphere = new THREE.Mesh(sphereGeometry, sphereMaterial);

sphere.position.x = 20;
sphere.position.y = 4;
sphere.position.z = 2;

scene.add(sphere);

camera.position.x = -30;
camera.position.y = 40;
camera.position.z = 30;
camera.lookAt(scene.position);

$('#WebGL-output').append(renderer.domElement);
renderer.render(scene, camera);
```

For More Information:
If we open this example in the browser, we will see something that resembles what we're aiming for, but is still a long way off:

Before we start making this more beautiful, I'll first walk you through the code a step at a time so that you understand what the code does:

```javascript
var scene = new THREE.Scene();

var camera = new THREE.PerspectiveCamera(45,
                                       window.innerWidth / window.innerHeight,
                                       0.1, 1000);

var renderer = new THREE.WebGLRenderer();
renderer.setClearColorHex(0xEEEEEE);
renderer.setSize(window.innerWidth, window.innerHeight);
```

For More Information:
Creating Your First 3D Scene with Three.js

Prior to the given example we defined a scene, a camera, and a renderer. The scene variable is a container that is used to store and keep track of all the objects that we want to render. The sphere and the cube that we want to render will be added to this scene later on in the example. In this first fragment, we also create a camera variable. The camera variable defines what we’ll see when we render the scene. In Chapter 2, Working with the Basic Components That Make Up a Three.js Scene, you will learn more about the arguments that you can pass into the camera. Next, we will define a renderer object. The renderer is responsible for calculating what the scene will look like in the browser based on the camera angle. We will create a WebGLRenderer object in this example that will use your graphics card to render the scene.

If you look through the source code and the documentation of Three.js, you’ll notice that there are different renderers available besides the WebGL-based one. There is a canvas-based renderer and even an SVG-based one. Even though they work and can render simple scenes, I wouldn’t recommend using them. They’re very CPU-intensive and lack features such as good material support and shadows.

Here we set the background color of the renderer to almost white (0xEEEEEE) with the setClearColorHex() function and tell the renderer how large the scene needs to be rendered by using the setSize() function.

So far, we’ve got a basic empty scene, a renderer, and a camera. There is, however, nothing yet to render. The following code adds the helper axes and the plane.

```javascript
var axes = new THREE.AxisHelper(20);
scene.add(axes);

var planeGeometry = new THREE.PlaneGeometry(60, 20);
var planeMaterial = new THREE.MeshBasicMaterial({
  color: 0xcccccc
});
var plane = new THREE.Mesh(planeGeometry, planeMaterial);

plane.rotation.x = -0.5*Math.PI;
plane.position.x = 15;
plane.position.y = 0;
plane.position.z = 0;
scene.add(plane);
```

For More Information:
As you can see, we have created an axes object and used the `scene.add()` function to add these axes to our scene. Now we will create the plane. This is done in two steps. First we define what the plane looks like using the new `THREE.PlaneGeometry(60, 20)` code. In this case it has a width of 60 and a height of 20. We also need to tell Three.js what this plane looks like (for example, its color and its transparency). In Three.js we do this by creating a material object. For this first example we’ll create a basic material (by using the `MeshBasicMaterial()` method) with the color `0xcccccc`. Next we combine these two into a Mesh object with the name `plane`. Before we add this plane to the scene we need to put it in the correct position; we do this by first rotating it 90 degrees around the x axis and next we defining its position in the scene by using the `position` property. If you’re already interested in the details of the Mesh object, look at example 06-mesh-properties.html from Chapter 2, Working with the Basic Components That Make Up a Three.js Scene, which shows and explains rotation and positioning. The final step that we need to do is add this plane to the scene, just like we did with the axes.

The cube and sphere are added in the same manner, but with the `wireframe` property set to `true`, so let's move on to the final part of this example:

```javascript
   camera.position.x = -30;
   camera.position.y = 40;
   camera.position.z = 30;
   camera.lookAt(scene.position);

   $('"#WebGL-output"').append(renderer.domElement);
   renderer.render(scene, camera);
```

At this point all the elements that we want to render are added to the scene at the correct positions. I've already mentioned that the camera defines what will be rendered. In this piece of code we position the camera using the x, y, and z position attributes to hover above our scene. To make sure that the camera is looking at our objects, we use the `lookAt()` function to point it at the center of our scene. All that is left to do is append the output from the renderer to the `<div>` element of our HTML skeleton; we use jQuery to select the correct output element, and tell the renderer to render the scene using the provided camera.

In the next couple of sections, we'll make this scene more pretty by adding lights, more materials, and even animations.
Adding materials, lights, and shadows

Adding new materials and lights in Three.js is very simple and is done in pretty much the same way as we explained in the previous section. We start by adding a light source to the scene (for the complete source, look at example 03-materials-light.html):

```javascript
var spotLight = new THREE.SpotLight( 0xffffff );
spotLight.position.set( -40, 60, -10 );
scene.add( spotLight );
```

The `SpotLight()` method illuminates our scene from its position (`spotLight.position.set( -40, 60, -10 )`). If we render the scene at this time, however, you won't see any difference with the previous one. The reason is that different materials respond differently to light. The basic material which we used in the previous example (by using the `MeshBasicMaterial()` method) doesn't do anything with the light sources in the scene. They just render the object in the specified color. So we have to change the materials for our plane, sphere, and cube as shown:

```javascript
var planeGeometry = new THREE.PlaneGeometry(60,20);
var planeMaterial = new THREE.MeshLambertMaterial({
color: 0xffffff});
var plane = new THREE.Mesh(planeGeometry,planeMaterial);
...
var cubeGeometry = new THREE.CubeGeometry(4,4,4);
var cubeMaterial = new THREE.MeshLambertMaterial({
color: 0xff0000});
var cube = new THREE.Mesh(cubeGeometry, cubeMaterial);
...
var sphereGeometry = new THREE.SphereGeometry(4,20,20);
var sphereMaterial = new THREE.MeshLambertMaterial({
color: 0x7777ff});
var sphere = new THREE.Mesh(sphereGeometry,sphereMaterial);
```

In this piece of code, we have changed the material property for our objects to a `MeshLambertMaterial`. Three.js provides two materials that take light sources into account: `MeshLambertMaterial` and `MeshPhongMaterial`. For More Information:

The result as shown in the following screenshot, however, still isn't what we're looking for:

We're getting there, and the cube and sphere are looking a lot better. What is still missing though are the shadows.

Rendering shadows takes a lot of computing power and for that reason shadows are disabled by default in Three.js. Enabling them, though, is very easy. For shadows we have to change the source in a couple of places as shown in the following code snippet:

```javascript
renderer.setClearColorHex(0xEEEEEE, 1.0);
renderer.setSize(window.innerWidth, window.innerHeight);
renderer.shadowMapEnabled = true;
```

For More Information:
The first change that we need to make is to tell the renderer that we want shadows. You can do this by setting the `shadowMapEnabled` property to `true`. If you look at the result from this change, you won't notice anything different yet. That is because we need to explicitly define which objects cast shadows and which objects receive shadows. In our example, we want the sphere and the cube to cast shadows on the ground plane. You can do this by setting the corresponding properties on those objects to `true` as follows:

```javascript
plane.receiveShadow = true;
...
cube.castShadow = true;
...
sphere.castShadow = true;
```

Now, there is just one more thing that you need to do to get the shadows. We need to define which of the light sources in our scene will cause shadows. Not all the lights can cast shadows, and you'll learn more about that in the next chapter, but the `SpotLight()` method that we have used in this example can. We only need to set the correct property and the shadows will finally be rendered:

```javascript
spotLight.castShadow = true;
```

And with this we get a scene complete with shadows from our light source as shown in the following screenshot:
The last feature that we'll add to this first scene is some simple animation. In Chapter 9, Animations and Moving the Camera, you'll learn more advanced animation options.

**Expanding your first scene with animations**

If we want to animate the scene, the first thing that we need to do is find some way to rerender the scene at a specific interval. Before HTML5 and the related JavaScript APIs came along, the way to do this was by using the `setInterval(function, interval)` function. With the `setInterval()` method, we can specify a function that, for instance, would be called every 100 milliseconds. The problem with this function is that it doesn't take into account what is happening in the browser. If you were browsing another tab, this function would still be fired every couple of milliseconds. Besides that, the `setInterval()` method isn't synchronized with the redrawing of the screen. This could lead to higher CPU usage and bad performance.

**Introducing the requestAnimationFrame() method**

Modern browsers luckily have a solution for the problems associated with the `setInterval()` function: the `requestAnimationFrame()` function. With the `requestAnimationFrame()` function, you can specify a function that is called at an interval defined by the browser. You can do any drawing that you need to do in the supplied function and the browser will make sure it is painted as smoothly and efficiently as possible. Using this is really simple (the complete source can be found in the example, `04-materials-light-animation.html`); you just have to create a function that handles the rendering as shown:

```javascript
function renderScene() {
    requestAnimationFrame(renderScene);
    renderer.render(scene, camera);
}
```

In the `renderScene()` function, we call the `requestAnimationFrame()` method again in order to keep the animation going. The only thing that we need to change in the code is that instead of calling the `renderer.render()` method after we've created the complete scene, we call the `renderScene()` function once to kick off the animation:

```javascript
...$
$(
  "#WebGL-output"
).append(renderer.domElement);
renderScene();
```

For More Information:

Creating Your First 3D Scene with Three.js

If you run the given code snippet, you won't see any changes compared to the previous example, because we didn't animate anything yet. Before we add the animation, I want to introduce a small helper library that gives us information about the frame rate that the animation is running at. This library, which is from the same author as Three.js, renders a small graph that shows us the Frames Per Second (FPS) that we're getting for this animation.

To add this statistic, we first need to include the library in the HTML `<header>` tag:

```html
<script type="text/javascript" src="../libs/stats.js"></script>
```

And we add a `<div>` element that will be used as output for the statistics graph:

```html
<div id="Stats-output"></div>
```

The only thing left to do is initialize the statistics and add them to the `<div>` element as shown:

```javascript
function initStats() {  
    var stats = new Stats();
    stats.setMode(0);
    stats.domElement.style.position = 'absolute';
    stats.domElement.style.left = '0px';
    stats.domElement.style.top = '0px';
    $('#Stats-output').append( stats.domElement );
    return stats;
}
```

This function initializes the statistics. The interesting part is the `setMode()` function. If we set it to 0 we'll measure the FPS, and if we set it to 1, we can measure the rendering time. For this example we're interested in FPS, so 0 it is. At the beginning of our anonymous jQuery function, we'll call this function and we've got the statistics enabled:

```javascript
$(function () {  
    var stats = initStats();
    ...
})
```

For More Information:
The only thing left to do is tell the stats object when we're in a new rendering cycle. We can do this by adding a call to the stats.update() method and to the render() function as follows:

```javascript
function render() {
    stats.update();
    ...
    requestAnimationFrame(render);
    renderer.render(scene, camera);
}
```

If you run the code with these additions, you'll see the statistics in the upper-left corner as shown in the following screenshot:

![Screenshot showing statistics in upper-left corner](image-url)
Creating Your First 3D Scene with Three.js

Animating the cube

With the requestAnimationFrame() method and the statistics configured, we've got a place to put our animation code. In this section we'll expand the render() function with code that will rotate our red cube around on all of its axes. Let's start by showing you the following code:

```javascript
function render() {
    ...
    cube.rotation.x += 0.02;
    cube.rotation.y += 0.02;
    cube.rotation.z += 0.02;
    ...
    requestAnimationFrame(render);
    renderer.render(scene, camera);
}
```

That looks simple, right? What we did is that we increased the rotation property of each of the axes by 0.02 every time the render() function was called, which showed as a cube smoothly rotating around all if its axes. Bouncing the blue ball isn't that much harder.

Bouncing the ball

To bounce the ball, we once again add a couple of lines of code to our render() function as follows:

```javascript
function render() {
    ...
    step+=0.04;
    sphere.position.x = 20+( 10*(Math.cos(step)));
    sphere.position.y = 2 + ( 10*Math.abs(Math.sin(step)));
    ...
    requestAnimationFrame(render);
    renderer.render(scene, camera);
}
```

With the cube we changed the rotation property; for the sphere we're going to change its position in the scene. We want the sphere to bounce from one point in the scene to another with a nice, smooth curve.

For More Information:
For this we need to change its position on the x axis and its position on the y axis. The Math.cos() and Math.sin() functions help us in creating a smooth trajectory by using the step variable. I won't go into the details of how this works here. For now all that you need to know is that step+=0.04 defines the speed of the bouncing sphere. In Chapter 8, Creating and Loading Advanced Meshes and Geometries, we'll go into more detail of how these functions can be used for animation, and I'll explain everything. The following screenshot shows the scene with the animation enabled:

Before wrapping up this chapter, I want to add one more element to our basic scene. When working with 3D scenes, animations, colors, and properties like these, it often requires a bit of experimenting to get the correct color or speed. It would be very easy if you could just have a simple GUI that allows you to change these kind of properties on the fly. Luckily, there is one.
Using the dat.GUI library to make experimenting easier

A couple of guys from Google created a library called dat.GUI (you can find the documentation online at http://code.google.com/p/dat-gui/), which allows you to very easily create a simple user interface component that can change the variables in your code. In this part of the chapter, we'll use dat.GUI to add a user interface to our example that allows us to:

- Control the speed of the bouncing ball
- Control the rotation of the cube

Just like we had to do for the statistics, we will first add this library to the <header> of our HTML page by using the following code:

```html
<script type="text/javascript" src="../libs/dat.gui.js"></script>
```

The next thing that we need to configure is a JavaScript object which will hold the properties that we want to change using the dat.GUI library. In the main part of our JavaScript code, we will add the following JavaScript object:

```javascript
var controls = new function() {
    this.rotationSpeed = 0.02;
    this.bouncingSpeed = 0.03;
};
```

In this JavaScript object we will define two properties: this.rotationSpeed and this.bouncingSpeed along with their default values. Next, we will pass this object into a new dat.GUI object and define the range for these two properties as shown:

```javascript
var gui = new dat.GUI();
gui.add(controls, 'rotationSpeed',0,0.5);
gui.add(controls, 'bouncingSpeed',0,0.5);
```

The rotationSpeed and bouncingSpeed properties are both set to a range from 0 to 0.5. All that we need to do now is make sure that in our render loop, we reference these two properties directly, so that when we make changes through the dat.GUI user interface, it immediately affects the rotation and bounce speed of our objects. This is shown as follows:

```javascript
function render() {
    ...
    cube.rotation.x += controls.rotationSpeed;
    cube.rotation.y += controls.rotationSpeed;
    cube.rotation.z += controls.rotationSpeed;
```
step+=controls.bouncingSpeed;
sphere.position.x = 20+(10*(Math.cos(step)));
sphere.position.y = 2+(10*Math.abs(Math.sin(step)));

Now when you run this example (05-control-gui.html), you'll see a simple user interface that you can use to control the bouncing and rotation speeds of the objects:

Using the ASCII effect
Throughout the chapter, we've worked at creating a pretty-looking 3D rendering by using the most modern browser features. Three.js also has a couple of interesting features that you can use to change the way the output is rendered. Before ending this chapter, I want to introduce you to one of these effects: the ASCII Effect. With the ASCII effect you can change our beautiful animation scene to a retro ASCII art-based animation, with a couple of lines of code. For this you have to change the last couple of lines of our main JavaScript loop from:

```javascript
$('#WebGL-output').append(renderer.domElement);
```

For More Information:
To the following:

```javascript
var effect = new THREE.AsciiEffect( renderer );
effect.setSize( window.innerWidth, window.innerHeight );
$('#WebGL-output').append(effect.domElement);
```

You also have to make a small change to the render loop. Instead of calling the `renderer.render(scene, camera)` method, you have to call the `effect.render(scene, camera)` method. The result of this effect is shown in the following screenshot:

![Screenshot of ASCII effect](example.png)

I have to admit that it isn't very useful, but it does nicely show you how easy it is to extend various parts of Three.js just because of its modularity.
Summary
That's it for the first chapter. In this chapter you've already learned a lot about the basic concepts that make up each Three.js scene and this should give you a good starting point for the next chapters. What you should remember from this chapter is the following:

- You can find all the source code for the examples in this and the other chapters online. The best way to learn is to play around and experiment with these examples.
- In the Three.js project, you created a scene to which you added the objects (a geometry together with a material) that you wanted to render.
- The materials that you used defined what the objects looked like. Each material reacted in a different way to light sources.
- Rendering shadows is expensive and needs to be turned on for the renderer, for each object, and for each light.
- You can do easy animations by just changing the position and the rotation properties of the objects in the scene.
- Statistics and custom controls can be easily added with the two helper libraries and a couple of lines of JavaScript.

In the next chapter we'll expand on the example that we've created here. You'll learn more about the most important building blocks that you can use in Three.js in the later chapters.

For More Information:
**Where to buy this book**


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](http://www.packtpub.com/learning-three-js-javascript-3d-library-for-webgl/book).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.