Community Experience Distilled

# Three.js Essentials

Create and animate beautiful 3D graphics with this fast-paced tutorial

Jos Dirksen

[PACKT] open source* community experience distilled PUBLISHING

# Three.js Essentials

Create and animate beautiful 3D graphics with this fast-paced tutorial

**Jos Dirksen**

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

# Three.js Essentials



First published: July 2014

Production reference: 1300614

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-086-4

`www.packtpub.com`

Cover image by Suyog Gharat (`yogiee@me.com`)

# Credits


**Author**
Jos Dirksen

**Reviewers**
Samrat Ambadekar
Chris "2pha" Brown
Justin Tolman

**Commissioning Editor**
Ashwin Nair

**Acquisition Editor**
Rebecca Pedley

**Content Development Editor**
Akshay Nair

**Technical Editors**
Taabish Khan
Pooja Nair

**Copy Editors**
Sarang Chari
Gladson Monteiro

**Project Coordinator**
Lima Danti

**Proofreaders**
Maria Gould
Ameesha Green

**Indexers**
Mehreen Deshmukh
Tejal Soni

**Production Coordinator**
Kyle Albuquerque

**Cover Work**
Kyle Albuquerque

# About the Author

**Jos Dirksen** has worked as a software developer and architect for more than a decade. He has quite a lot of experience in a large range of technologies that range from backend technologies, such as Java and Scala, to frontend development using HTML5, CSS, and JavaScript. Besides working with these technologies, he 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 be best used to create beautiful data visualizations, the results of which you can see on his blog at `http://www.smartjava.org/`.

Jos currently works as an enterprise architect for Malmberg, a large Dutch publisher of educational material. He helps to create a new digital platform for the creation and publication of educational content for primary, secondary, and vocational education. Previously, he 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.

Jos has already written a book on Three.js named *Learning Three.js: The JavaScript 3D Library for WebGL*, *Packt Publishing*, which is an in-depth description of all the features Three.js provides. Besides his interest in frontend JavaScript and HTML5 technologies, he is also interested in backend service development using REST and traditional web service technologies. He has already written two books on this subject. He is the co-author along with *Tijs Rademakers* of *Open-Source ESBs in Action*, *Manning Publications*, an action book that was published in 2008. In 2012, he published a book on how to apply SOA Governance in a practical manner, titled *SOA Governance in Action*, *Manning Publications*.

# Acknowledgment

Writing a book isn't something you do by yourself. A lot of people have helped and supported me when I was doing this, and my special thanks go out to the following people:

- All the guys from Packt Publishing who have helped me during the writing, reviewing, and laying out part of the process. Great work guys!
- I, of course, have to thank Ricardo Cabello, also known as Mr. dò_ób, for creating the great Three.js library.
- Many thanks go to the reviewers. They have provided me with great feedback and comments that really helped me improve the book; their positive remarks have really helped me shape the book!

And, of course, I'd like to thank my family. I'd like to thank my wife, Brigitte, for supporting me and my two girls, Sophie and Amber, who can always find reasons to pull me away from the keyboard and computer.

# About the Reviewers

**Samrat Ambadekar** is a user experience and interaction designer based in California, U.S. He holds a Master's degree in Human Computer Interaction from the Georgia Institute of Technology, Atlanta. He has more than five years of experience as a designer and a developer. His work and interests span across interaction design, interactive environments, augmented reality, and information visualization. You can track him at `www.samratambadekar.com`.

**Chris "2pha" Brown** is a web and Drupal developer based out of Brisbane, Australia. He has been dabbling with 3D from around 2003 when he started making custom characters and mods for *Unreal Tournament 2003*. Since then, he has kept his 3D skills up to date by occasionally working on both personal and professional 3D projects. After completing a multimedia degree in 2007, he went into a Flash developer role where he discovered Papervision3D, which allowed his 3D creations to be visualized on the Web via Flash. With new tools and technologies such as HTML5 and Three.js becoming more popular and widespread, he has incorporated these into his 3D/web workflow. When not creating awesome stuff with Drupal, Three.js, and 3D, he can be found trying to get a knee down on his bike or sitting on a beach in Thailand. You can keep up to date with him at `www.2pha.com`.

**Justin Tolman** currently works as a contract-based web and mobile developer in Boise, Idaho. Programming and electronics have interested him since childhood. He started building websites in 2005 and has worked with WebGL and Three.js since 2011. On the mobile side of things, he has written native applications for Android and iOS devices. His favorite operating system is Linux. He has trouble deciding between Firefox and Chrome, so we'll just say that his favorite browser is *not Internet Explorer*. He has also dabbled in robotics and automation. He has done technical support work for a US Defense contractor, and is a former military intelligence cryptographer and linguist for the US Army.

In addition to his experience with a wide range of technologies and a variety of programming languages, Justin also has a keen interest in human languages. He is a native English speaker, is fluent in Thai, and has also studied Laotian, Spanish, and Arabic. In his spare time, he enjoys reasoning puzzles and games, reading, and outdoor activities. He has a scuba diving certification and is a member of Mensa.

---

I would like to thank the developers and contributors of Three.js for creating a great library of tools for 3D on the Web.

---

# www.PacktPub.com

## Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.


`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

## Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

## Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

# Table of Contents

# Preface

Web development has been changing a lot in the last couple of years. JavaScript libraries have matured, browsers have become more powerful, and the HTML5 spec is well supported on most systems. Currently, though, the Web mostly focuses on 2D to represent information, create games, and build websites. There is also, however, a great and standardized way to create 3D graphics. This is called WebGL; it provides an API to create hardware-accelerated 3D graphics.

However, the WebGL API isn't very easy to use. It requires you to program in C, and you really need to have an in-depth knowledge of how WebGL works internally to work with this API. Luckily though, there are a number of JavaScript libraries available that hide the complexity of WebGL and provide you with an easy-to-use API to create 3D applications and visualizations. Currently, the best of these JavaScript libraries, the one that is explained in this book, is called Three.js.

With Three.js, you're provided with an easy-to-use API and a whole range of advanced materials and shapes that you can use to access WebGL. In this book, we'll walk you through the most essential features of Three.js. We do this by creating some advanced visualizations that cover various parts of the Three.js API.

## What this book covers

*Chapter 1*, *Get Up and Running with Three.js*, explains how to get the source code and set up a local environment to get started with the development of Three.js. At the end of this chapter, you'll have a simple Three.js scene that we will further expand on in the following chapters.

*Chapter 2*, *Creating a 3D World Globe and Visualizing Open Data*, shows how you can create a rotating 3D globe. While going through this example, you'll learn how to use materials, how to set up camera controls, and how to enhance your object using the various available textures. As a bonus, we'll also explain how you can use the HTML5 canvas as a texture.

*Chapter 3*, *Navigate around a Randomly Generated Maze*, explains how you can create a random 3D maze through which you can navigate a cube. In this chapter, we'll show you how to use animations, lights, and the rotation and position properties.

*Chapter 4*, *Visualizing Audio Data with a Particle System*, uses the HTML5 Web Audio API in combination with Three.js to visualize audio. This chapter shows what you can do with the particle system from Three.js. It shows this through visualizations of the waveform, the amplitude, and a 3D histogram.

*Chapter 5*, *Programmatic Geometries*, shows an alternative way through which you can create 3D geometries. Three.js, out of the box, offers a number of standard geometries, but creating geometries from scratch is also very easy. In this chapter, we'll show you the various properties you need to configure to correctly set up your own geometries. We do this by creating a programmatic terrain and a programmatic tree.

*Chapter 6*, *Combining HTML and Three.js with CSS3DRenderer*, explains how to use CSS3DRenderer to apply a 3D transformation to HTML elements. We'll show a couple of examples that explain what is possible with CSS3DRenderer and how to use it to animate any HTML element.

*Chapter 7*, *Loading and Animating External Models Using Blender*, shows how you can load and display models created by external 3D modeling programs. This chapter will also show you a Blender-based workflow that explains how you can model your geometries in Blender and then show and animate these models through Three.js.

# What you need for this book

Three.js doesn't have any special requirements. All you need for this book is a text editor and a modern web browser. Some examples run better with a local web browser or require you to disable some security sessions. The steps you need to take to accomplish this are explained in the first chapter of this book.

# Who this book is for

This book is for developers who already know JavaScript and want to get acquainted with Three.js. Through the examples in this book, you'll quickly learn the most essential parts of Three.js. No special math or WebGL skills are required. All you need to know is some basic JavaScript and HTML. The examples can be freely downloaded and no special tools or products are required to run and play around with the examples. If you want to get started with creating beautiful 3D visualizations, this book will give you the skills to do that.

# Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We use the `position` property to determine where the light should be added and then we add the light to the scene."

A block of code is set as follows:

```
varcurrentGeometry = create3DTerrain(21, 21, 21, 21, 50);
vartargetGeometry = create3DTerrain(21, 21,21, 21, 50);
var container = new THREE.Object3D();

// for each vertices add a sprite
geometry.vertices.forEach(function(e) {
varcssObject = new THREE.CSS3DSprite(createDiv());
cssObject.position = new THREE.Vector3(e.x, e.y, e.z);
container.add(cssObject);
});
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
varcurrentGeometry = create3DTerrain(21, 21, 21, 21, 50);
vartargetGeometry = create3DTerrain(21, 21,21, 21, 50);
var container = new THREE.Object3D();

// for each vertices add a sprite
geometry.vertices.forEach(function(e) {
varcssObject = new THREE.CSS3DSprite(createDiv());
cssObject.position = new THREE.Vector3(e.x, e.y, e.z);
container.add(cssObject);
});
```

Any command-line input or output is written as follows:

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

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "From the **Develop** menu, you can now enable WebGL through **Develop** | **Enable WebGL**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

# Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

# Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## 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.

# Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from the following link: `https://www.packtpub.com/sites/default/files/downloads/0864OS_ColoredImages.pdf`.

# Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

# Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

# Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

# 1
# Get Up and Running with Three.js

Three.js is an open source JavaScript library that allows you to create and render 3D scenes directly in your browser. Three.js provides an extensive API for this with a large set of functions. The online documentation for Three.js, however, is very sparse and doesn't provide information on how to create complex visualizations. In this book, you'll learn the essential parts of Three.js by creating a number of extensive examples so that after reading this book, you'll be able to create these kind of complex graphics yourself.

Before we dive into the examples that will help you learn the essentials of Three.js, we first have to set up an environment that you can use to experiment and play around with. Besides setting up this environment, we'll also introduce a couple of concepts around which Three.js is built and show some techniques that you can use if you run into problems. The following topics will be discussed in this chapter:

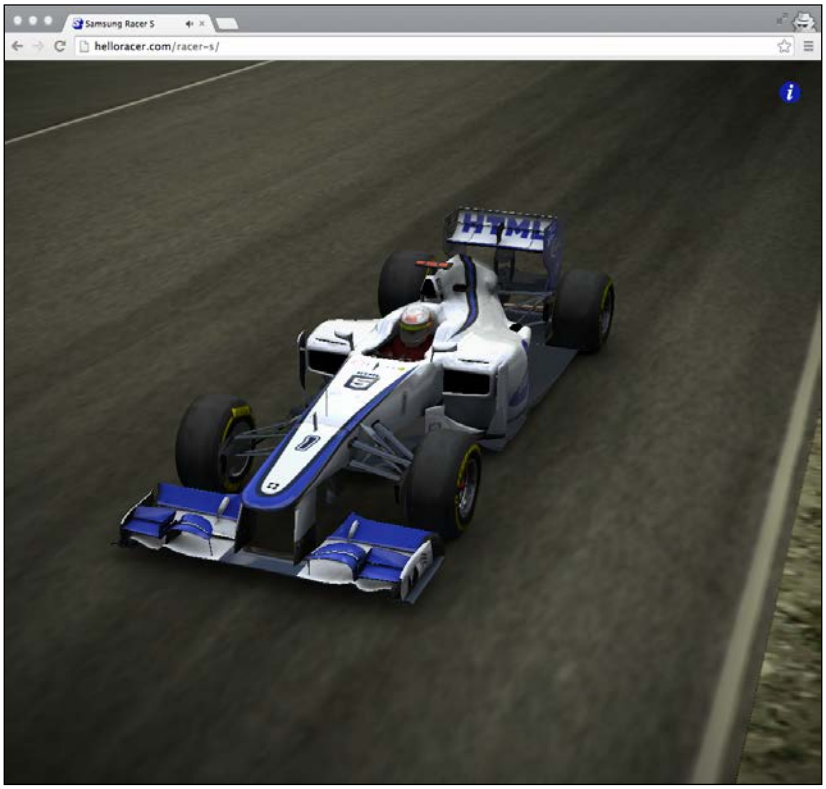
- What is Three.js, and what are its requirements?
- Getting access to the source of the examples discussed in this book
- Setting up a local environment to experiment with the examples
- Creating a minimal Three.js scene
- Extending the scene with additional helper libraries for stats, control, and progress

Let's get started with a quick introduction to Three.js.

# Introducing Three.js

Three.js is an open source JavaScript library that's been around since 2010 and allows you to easily create good-looking 3D scenes directly inside your browser. For rendering, Three.js uses WebGL (`http://www.khronos.org/webgl/`) if a browser supports it, and it can fall back to an HTML5 canvas or an SVG approach if WebGL isn't supported.

Creating 3D scenes using WebGL directly (or using a canvas) is very hard. Three.js provides an easy-to-use API to create and manipulate 3D objects and scenes without having to know too much about WebGL or complex math formulas. The following screenshot shows a Three.js-based game that even runs on mobile browsers (`http://helloracer.com/racer-s/`):




A lot of information about Three.js can be found online at the main Three.js site, `http://threejs.org`. For an extensive reference guide, you can also look at *Learning Three.js: The JavaScript 3D Library for WebGL*, *Jos Dirksen*, *Packt Publishing*.

In the next section, we'll look at the requirements for Three.js to work.

# Looking at the requirements for Three.js

Three.js is a 100 percent JavaScript library and hasn't got any dependencies to other libraries, so it can run completely in a standalone manner. To get the most out of Three.js, however, you need a browser that supports the WebGL standard. Luckily though, most modern browsers on desktop and mobile currently support this standard. The following table provides an overview of the supported desktop browsers:

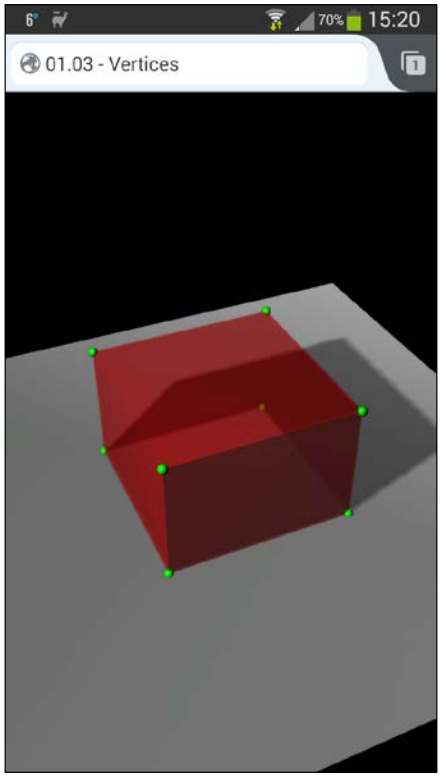
| Browser | Description |
|---|---|
| Internet Explorer | Supports WebGL from version 11. For older versions, you can use the IEWebGL plugin that adds support for WebGL to IE 9 and IE 10. |
| Mozilla Firefox | WebGL is supported from version 4. |
| Google Chrome | Supports WebGL from version 10. |
| Safari | Supports WebGL from version 5.1 and above installed on Mac OS X Mountain Lion, Lion, Snow Leopard, and Maverick.<br>You might need to explicitly enable WebGL in Safari by navigating to **Preferences** \| **Advanced**, and checking the **Show Develop Menu In Menu Bar** option. From the **Develop** menu, you can now enable WebGL through **Develop** \| **Enable WebGL**. |
| Opera | Supports WebGL from version 12.0. You still might need to enable this though. Type `opera:config` in the address bar. Once there, set the value for **WebGL** and **Enable Hardware Acceleration** to `1`. After this, restart the browser and WebGL will be enabled. |

As you can see, almost all modern browsers support WebGL. Support and especially performance on mobile devices, however, is very diverse. The following list shows the browsers that support WebGL and thus Three.js on mobile devices:

- Mozilla Firefox for Android
- Google Chrome for Android (you might need to turn it on explicitly)
- Opera Mobile
- Blackberry Browser

Note that performance is dependent on how old your device is and the OS you're running. It might even be the case that your graphics card might be blacklisted (`http://www.khronos.org/webgl/wiki/BlacklistsAndWhitelists`).

Generally speaking, support on mobile Android devices, especially the modern ones, is pretty good. The following screenshot, for instance, shows how one of the examples from this chapter runs on Firefox for Android on a Samsung Galaxy S3.




Note that as iOS doesn't support WebGL (yet), most of the examples from this book won't run on iOS devices because we will use the WebGL renderer provided by Three.js. Three.js also provides an HTML5 canvas and an SVG renderer that do work on iOS but with rather bad performance. A subset of Three.js's functionalities is provided by the CSS3D renderer, which is explained in *Chapter 6*, *Combining HTML and Three.js with CSS3DRenderer*, and which runs really well on mobile devices.

# Setting up a local development environment

The easiest and fastest way to learn Three.js is by playing around with the examples in this book. In this section, we'll have a quick look at how you can get the source code for this book and set up a local web server to easily test and extend the examples.

## Getting the source code

There are two different ways in which you can get the source code for this chapter. You can download them directly from the Packt website at `http://www.packtpub.com/support`, or you can get the code directly from the GitHub repository. For the first approach, point your browser to the specified URL, download the source code, and extract the `.zip` file into a directory of your choice.

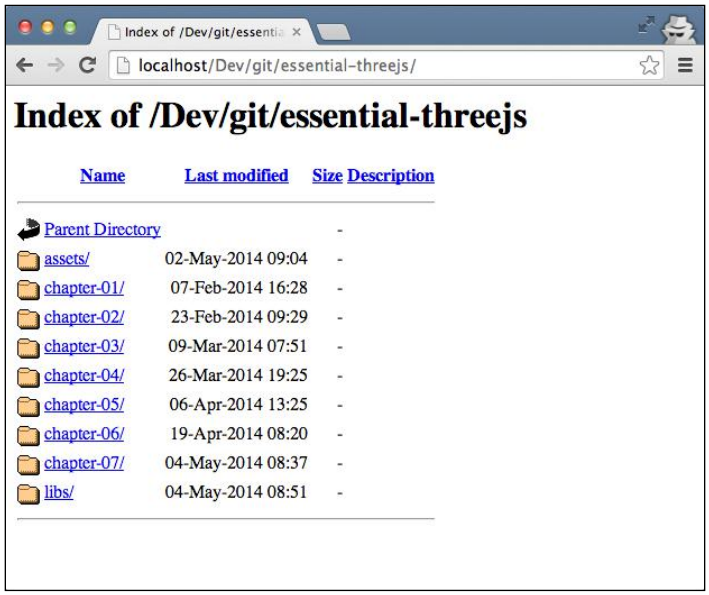
When you want to get the source code directly from GitHub, first make sure you've got Git installed. A good manual on how to install Git for various operating systems can be found online at `http://git-scm.com/book/en/Getting-Started-Installing-Git`. After you've installed Git, all you have to do is run the following command from the command line:

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

This will download all the source code to the directory you're currently in and show an output that looks something like the following code:

```
Cloning into 'essential-threejs'...
remote: Reusing existing pack: 8, done.
remote: Counting objects: 36, done.
remote: Compressing objects: 100% (21/21), done.
remote: Total 44 (delta 6), reused 0 (delta 0)
Unpacking objects: 100% (44/44), done.
Checking connectivity... done
```

Now, you'll have a directory where all the source code can be found as shown in the following screenshot:




You could open the files directly in your browser, but for the complex examples in the following chapters, where resources are loaded asynchronously, this approach won't work. To make sure all the examples work, the best thing to do is set up a local web server.

# Setting up a local web server

Setting up a local web server is fairly straightforward. In this section, we'll explore the following three different approaches:

1. A Python-based approach for those who have Python installed.
2. If you're a Node.js developer or have played around with it, you can use the `npm` command.
3. If the previous two approaches don't work for you, you can always install Mongoose, which has a portable version for Mac and Windows.

Let's start by looking at the Python-based approach.

## Using Python to run a web server

The easiest way to check whether you've got Python installed is by just typing `python` on the command line. If you see something like the following output, you've got Python installed, and you can use it to run a simple web server:

```
> python
Python 2.7.3 (default, Apr 10 2013, 05:09:49)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

All you need to do to run a simple Python-based web server is type the following code on the command line:

```
> python -m SimpleHTTPServer
```

The output will be as shown in the following snippet:

```
Serving HTTP on 0.0.0.0 port 8000...
```

This will start a web server on port 8000, which allows you to access all the files from the directory where you've run this command.

## Using the npm command from Node.js to run a web server

Alternatively, if you haven't got Python installed, you can check whether the npm package manager from the Node.js distribution is available by just typing the following command on the command line:

```
> npm
```

If the output is as shown in the following snippet, then you can use the npm-based approach:

```
Usage: npm <command>
...
```

In this approach, you can run a web server that serves the files from the directory you're in, using the following command:

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

After installing the web server, you can start it using the following command:

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

If both the scenarios discussed so far fail, you can also opt to download a portable web server directly.

## Running a portable version of Mongoose

You can download Mongoose, a portable web server, from `https://code.google.com/p/mongoose/downloads/list`. In Windows, you can just copy the downloaded executable to the folder where you've put the source code files and double-click on the executable to launch the web server. For other platforms as well, copy the file to the directory that contains the source code files. However, you can't run the executable directly; you must start it from the command line. Once started, your web server will be up and running on port 8080.

There is a final alternative if you're not able to install a local web server. You can open the files directly from the browser (or drag-and-drop them in the browser window). For a lot of examples, this will work directly; however, once textures or other features that require the loading of external resources are required, this approach will stop working because of browser security settings. It is possible to change this security policy so that the examples that use those features also work. A good explanation on how to do this is shown in the Three.js website at `https://github.com/mrdoob/three.js/wiki/How-to-run-things-locally`.

# Creating a minimal Three.js web application

At this point, you'll have the source code and a locally running web server. Now, let's look at the basics of Three.js to prepare you for the examples in the following chapters. In this section, we'll introduce you to a basic Three.js scene and the basic building blocks.

# Creating a scene to contain all the objects

Let's start with a minimal Three.js scene where we'll show you the following:

- How to include the correct libraries
- How to create a `THREE.Scene` object
- How to add a `THREE.Camera` object
- How to set up a render loop

The source code that the following example refers to is the `01.01-basic-scene.html` file. The first thing that we need to do is include the Three.js JavaScript library. This is described in the following code:

```
<head>
    <script src="../libs/three.js"></script>
</head>
```

Here, we include the `three.js` library from the `libs` folder. This library also comes in a minified version (the `three.min.js` library), which you can also use and which downloads faster. In our examples, we use the normal one as it makes debugging inside the included Three.js file much easier.

Now, we can add the `script` tag to the HTML page where we will add our Three.js code. This is described in the following code:

```
<script>
    // code
</script>
```

**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.

Now, let's look at what we need for JavaScript to set up a minimal Three.js web application. The first thing we do inside the `script` tag is set up a couple of global variables, which we'll explain later. This setup is described in the following code:

```
// global variables
var renderer;
var scene;
var camera;
```

Next, we'll look at the `init()` function that we'll use to initialize the Three.js library once the complete document is loaded. This is described in the following code:

```
function init() {
    // Three.js initialization code
}

window.onload = init;
```

With the `window.onload` function, we tell the browser to call the `init()` function when the document is loaded. The `init()` function itself looks like as described in the following code:

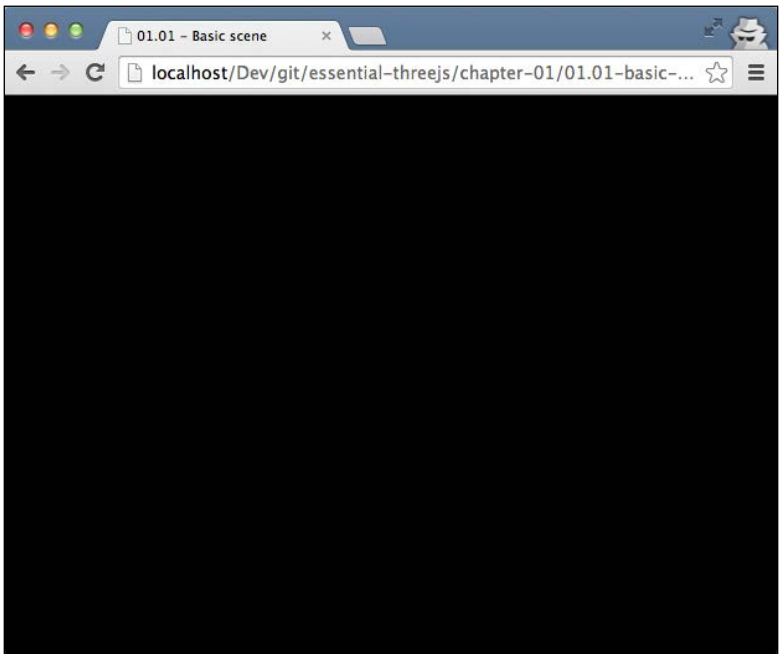
```
function init() {
    scene = new THREE.Scene();

    renderer = new THREE.WebGLRenderer();
    renderer.setClearColor(0x000000, 1.0);
    renderer.setSize(window.innerWidth, window.innerHeight);
    renderer.shadowMapEnabled = true;

    camera = new THREE.PerspectiveCamera(
        45, window.innerWidth / window.innerHeight, 0.1, 1000);
    camera.position.x = 15;
    camera.position.y = 16;
    camera.position.z = 13;
    camera.lookAt(scene.position);

    document.body.appendChild(renderer.domElement);
    render();
}
```

In the previous fragment of code, we created three basic Three.js objects. First, we created a `THREE.Scene` object. This object is the container that will hold all the objects we want to render. Next, we created a `THREE.WebGLRenderer` object. This is the Three.js object that we'll use to render the created `THREE.Scene` object. Finally, the `THREE.PerspectiveCamera` object determines what we see. The last call of the `init()` function is a call to the `render()` function, which is shown in the following code:

```
    function render() {
        // render using requestAnimationFrame
        renderer.render(scene, camera);
        requestAnimationFrame(render);
}
```

In this function, you can see that we use the `renderer.render()` function to visualize the scene using the camera. In this function, we also use the `requestAnimationFrame` function to set up a render loop. With the `requestAnimationFrame` function, we tell the browser to determine when it thinks it is best to call the supplied function (the `render` function in this case). This way, we can offload the graphical rendering of the scene from the normal JavaScript thread; this provides a much smoother experience and better performance.

As you can probably guess, when we open this example (the `01.01-basic-scene.html` file) in the browser, the result is still just a black screen. This is because we don't have any objects and any light in the scene yet. This is seen in the following screenshot:

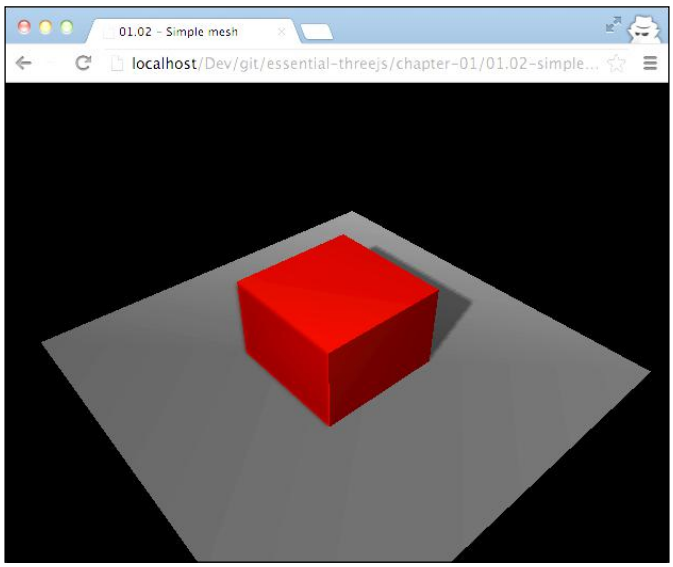



Even though this is just a black scene, this is the minimal Three.js skeleton you can create.

In the next section, we'll make the scene a bit more interesting by adding a ground plane, a cube, and a light.

# Adding a mesh created from geometry

Now that we've got our basic scene set up, we'll add the elements to be rendered. In this section, we'll create the scene shown in the following screenshot (the `01.02-simple-mesh.html` file):




In the previous screenshot, you can see a simple cube that is rendered on top of a square floor. In the following couple of code fragments, we'll show you how to do this. The first thing we do is add the cube. This is described in the following code:

```
var cubeGeometry = new THREE.CubeGeometry(6, 4, 6);
var cubeMaterial = new THREE.MeshLambertMaterial({
  color: "red"
});
var cube = new THREE.Mesh(cubeGeometry, cubeMaterial);
cube.castShadow = true;
scene.add(cube);
```

Now, the first thing we need to do is create the shape and the geometry of the cube. We do this by creating a new `THREE.CubeGeometry` object where we specify the width, the height, and the depth of the cube. Now that we have the shape, we must also specify a material. This material is used to determine how an object will react to the light sources in the scene. For instance, you can use it to determine its color, reflectivity, whether it's transparent, and much more. In this example, we will create a `THREE.MeshLambertMaterial` object and set the cube's color to red.

We'll discuss materials in more detail in the next chapter. For now, it's enough to know that a material requires a light source to be present in the scene to determine the color it should show. Now, we need to combine these two components; we do this by creating a `THREE.Mesh` object, where we provide the geometry and the material as arguments. This `THREE.Mesh` object is then added to the scene as shown in the previous code fragment.

For the floor plane, we do pretty much the same. Consider the following code:

```
var planeGeometry = new THREE.PlaneGeometry(20, 20);
var planeMaterial = new THREE.MeshLambertMaterial({
  color: 0xcccccc
});
var plane = new THREE.Mesh(planeGeometry, planeMaterial);
plane.receiveShadow = true;

plane.rotation.x = -0.5 * Math.PI;
plane.position.y = -2;

scene.add(plane);
```

What is different here is that we set two additional properties of the create `THREE.Mesh` object. We set its position in the scene and we set its rotation. The rotation is needed because normally a `THREE.PlaneGeometry` object is oriented vertically, and as we want to have a horizontal plane, we need to rotate it by 90 degrees (which is 0.5*Pi in radians). As you can see in the previous code, we also set its position. If we don't do this, our ground will cut through half of the cube as the center of the cube is at the position 0, 0, 0 (because we didn't specify it explicitly). The height of the cube is 4, so if we offset the position of the ground by 2, the cube will fit in completely.

Now all that is left to do is to add the light and tell Three.js that we want to see shadows. This is described in the following code:

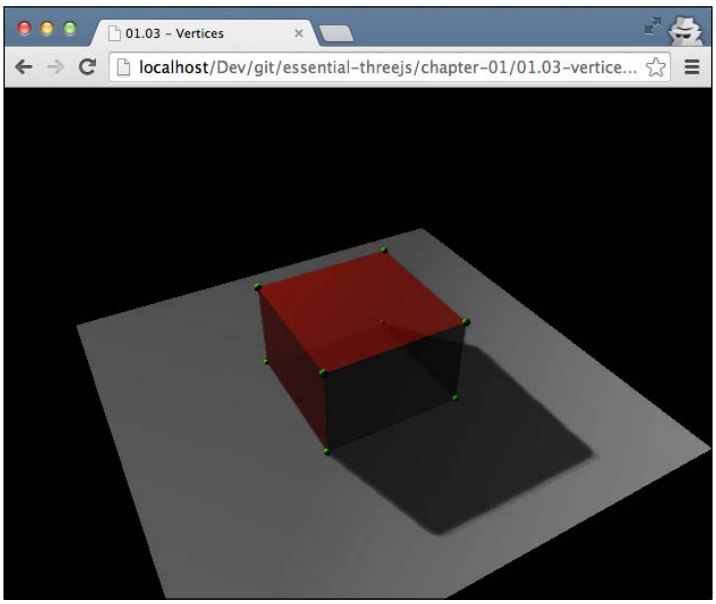
```
var spotLight = new THREE.SpotLight(0xffffff);
spotLight.position.set(10, 20, 20);
spotLight.castShadow = true;
scene.add(spotLight);
```

For this scene, we've created a `THREE.SpotLight` object that works like a spot on the ceiling or a flashlight. We use the `position` property to determine where the light should be added and then we add the light to the scene. To enable shadows, we must tell Three.js which objects cast a shadow and which objects should receive a shadow. For this example, it's only useful to show shadows on the ground plane, so we set the `plane.receiveShadow` object to `true` to enable shadow rendering on this plane. Now, we just need to set the `castShadow` object to `true` on the cube and the light, and we're almost done. Finally, we need to explicitly enable shadow rendering on the renderer. This is done by setting the `renderer.shadowMapEnabled` object to `true`. With all this code in place, you get the result you've seen in the screenshot at the beginning of this section.

Before we move on to the next section, we'd like to go one step deeper into two important concepts of Three.js (or modeling in general): vertices and faces.

## What are vertices?

In the later chapters of this book, we'll sometimes mention vertices, so it's good to understand what we're talking about in that case. A vertex (plural, vertices) is a single point with an *x*, *y*, and *z* coordinate. When we create a geometry, we're actually creating a number of vertices that together define the shape of the object. If we show the vertices for the cube we created earlier, you will get something like the following screenshot (the `01.03-vertices.html` file):

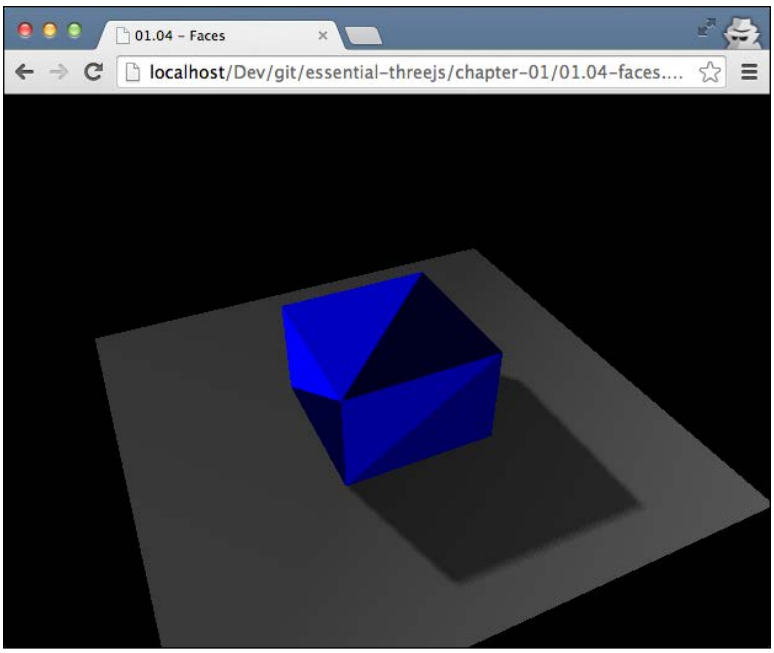


Here, each vertex is shown as a small green sphere. So, when we create a `THREE.CubeGeometry` object, we actually create eight vertices.

Vertices themselves don't determine the shape of an object. This shape is defined by how the vertices are connected to one another. This concept is called faces.

## Combining vertices into faces

A face in Three.js consists of three vertices that together fill out an area. This area is called a face. The easiest way to understand this is by looking at the following screenshot (the `01.04-faces.html` file):




Here, you can see that each side of the cube is divided into two faces, and the corners of each face are defined by a vertex.

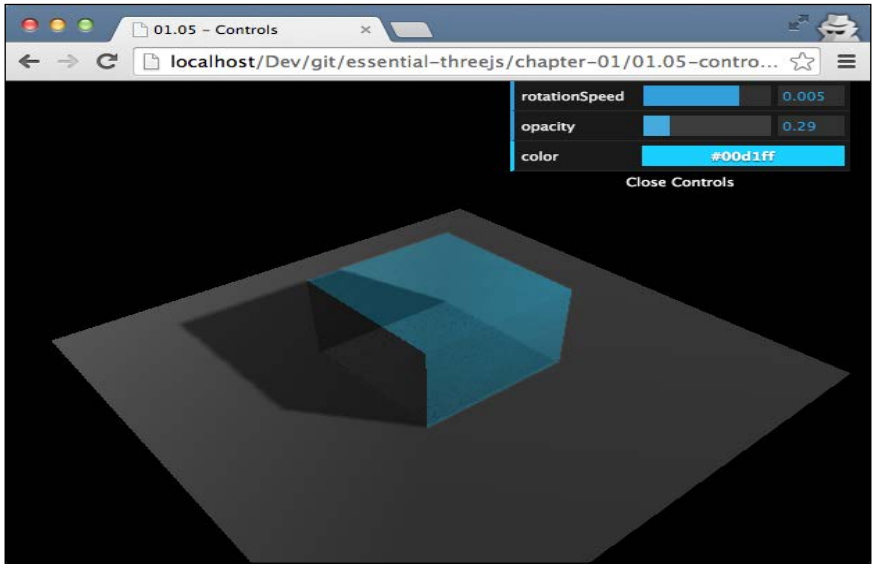
Now that you've seen the basic components of a scene and know about faces and vertices, we'll add some helper elements to the scene to best prepare you for the more complex examples in the following chapters.

# Enhancing the basic scene

You can easily experiment with the previous examples by changing the code and refreshing your browser. In this section, we'll introduce two additional components that will make experimenting easier and allow you to keep an eye on the performance of your 3D web application.

# Adding easy controls with the dat.GUI library

The first thing that we do is add the dat.GUI library (`https://code.google.com/p/dat-gui/`) to our basic scene. Even though this library isn't part of Three.js, we can use it to easily add a simple UI element that can be used to control some properties of your scene. For instance, in the following screenshot (the `01.05-controls.html` file), you can use the menu on the top right-hand side of the window to control a number of properties of the scene:




Adding a control element to our scene is very simple and only requires a couple of lines of code. First, of course, we need to include the JavaScript dat.GUI library. This is described in the following line of code:

```
<script src="../libs/dat.gui.min.js"></script>
```

Next, what we need to do is create a JavaScript object, which you do counter intuitively in JavaScript by calling the `new function()`, which contains the values we want to change using the `dat.GUI` library. This is described in the following code:

```
control = new function() {
  this.rotationSpeed = 0.005;
  this.opacity = 0.6;
  this.color = cubeMaterial.color.getHex();
};
addControlGui(control);
```

The `addControl` function creates the menu you see on the top right-hand side of the window. Consider the following code:

```
function addControlGui(controlObject) {
  var gui = new dat.GUI();
  gui.add(controlObject, 'rotationSpeed', -0.01, 0.01);
  gui.add(controlObject, 'opacity', 0.1, 1);
  gui.addColor(controlObject, 'color');
}
```

Now, whenever we use the UI to change one of the elements, the value in the supplied control object also changes. In the render loop that we created, we can update the corresponding values to reflect the changes in the objects on the screen. This is described in the following code:

```
function render() {
   ...
  scene.getObjectByName('cube').material.opacity =
    control.opacity;

  scene.getObjectByName('cube').material.color =
    new THREE.Color(control.color);

  renderer.render(scene, camera);
  requestAnimationFrame(render);
}
```

The more complex your scene becomes, the longer it will take to render and update, which might adversely affect the frame rate. To keep track of this, we'll add a simple counter that shows the current frame rate.

# Add a statistics element to show the frame rate

For statistics, we use an external library called stats.js, which happens to be created by the author of Three.js. You can get this library from `https://github.com/mrdoob/stats.js`. Adding a statistics element with this library is fairly straightforward. First, of course, you have to include the corresponding JavaScript library, which is described in the following line of code:

```
<script src="../libs/stats.min.js"></script>
```

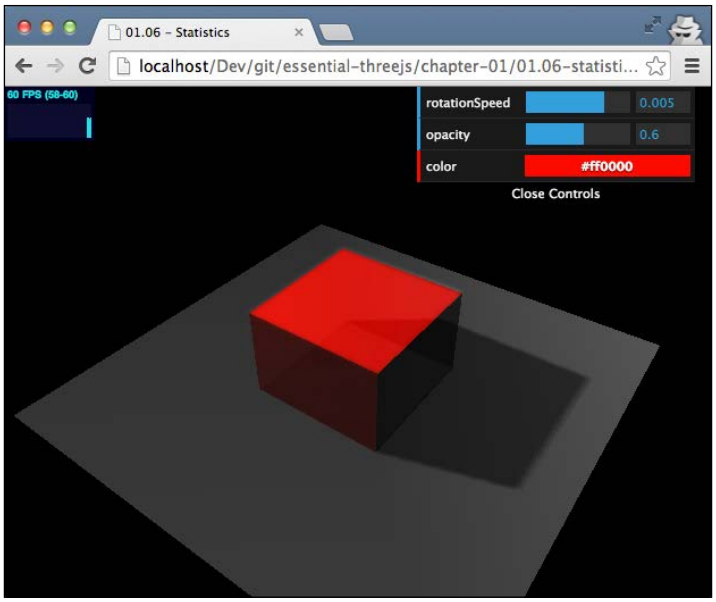
Next, we need to configure what kind of statistics we want to show and where to show them. This is described in the following code:

```
function addStatsObject() {
    stats = new Stats();
    stats.setMode(0);
    stats.domElement.style.position = 'absolute';
    stats.domElement.style.left = '0px';
    stats.domElement.style.top = '0px';
    document.body.appendChild( stats.domElement );
}
```

Stats.js supports two modes. Mode 0, which we will use here, shows the **frames per second** (**FPS**). If we set the mode to 1, we will see the number of milliseconds needed to render the frame. In this function, we also position the stats element at the top left-hand side of the window. Now, all we need to do is call the `update()` function on the `stats` object whenever we render a new frame. So, logically, we add this to the `render()` function. This is described in the following code:

```
function render() {
  ...
  stats.update();
  ...
}
```

The same scene with a statistics element looks like as shown in the following screenshot (the `01.06-statistics.html` file):




For the last subject in this chapter, we'll give you two tips that will make it easier for you to experiment and play around with the examples.

# Debugging the examples in this book

While creating JavaScript applications, it's important to understand what happens when your application runs in the browser. In the past, we used to use pop-up alerts for this, but luckily, modern browsers (especially Firefox and Chrome) come with good development tools. In this section, we'll quickly show two approaches that you can use to get a better understanding of what happens inside your code when you run the application in a browser. First, we'll look at how to log in to the browser's console log, and after that, we'll quickly show how you can use breakpoints (in Chrome, for this example) to see and explore the current state of your web application.

# Using console logging for debugging

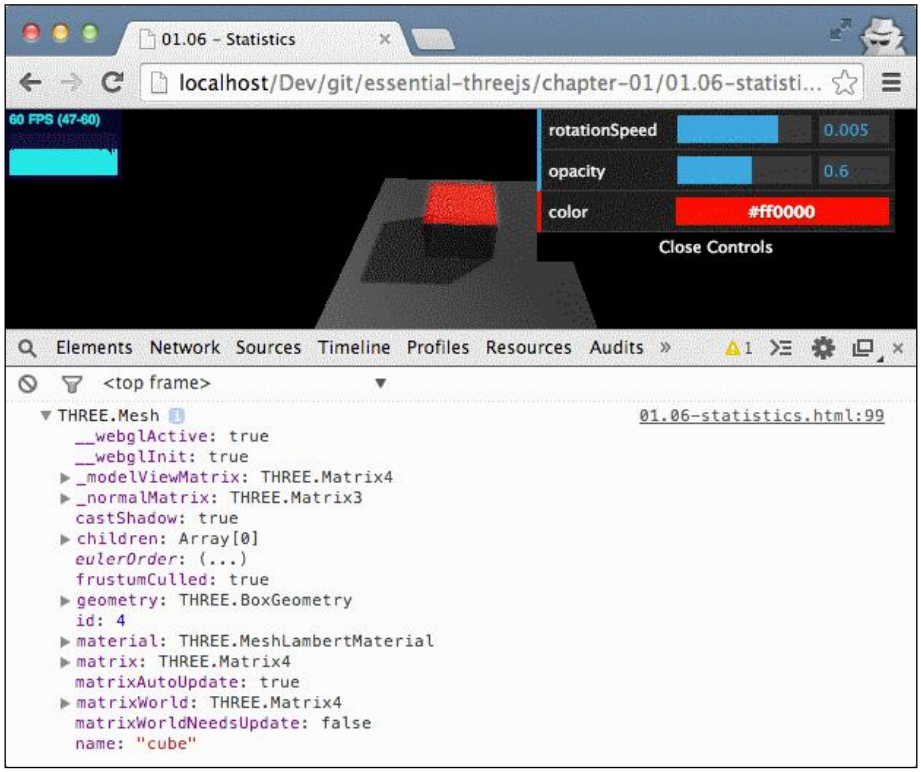
Logging from JavaScript used to be hard, but with modern browsers, there is finally a general way of logging from JavaScript. Just by adding the following line of code to your JavaScript code, you can log a statement:

```
console.log('Logging something to the browsers console log');
```

Let's add a couple of log statements to the `init()` function that we saw earlier. We'll log a simple text message, but we'll also log the cube mesh we added. To do this, we just add the following two lines of code at the end of the `init()` function:

```
console.log('Log statement from the init function');
console.log(cube);
```

In Chrome, the result looks like the following screenshot. You can open the console from the menu through **View** | **Developer** | **JavaScript Console** or by accessing the settings and then going to **Tools** | **JavaScript Console**. In Firefox, you can find the console in **View** | **Firebug**.



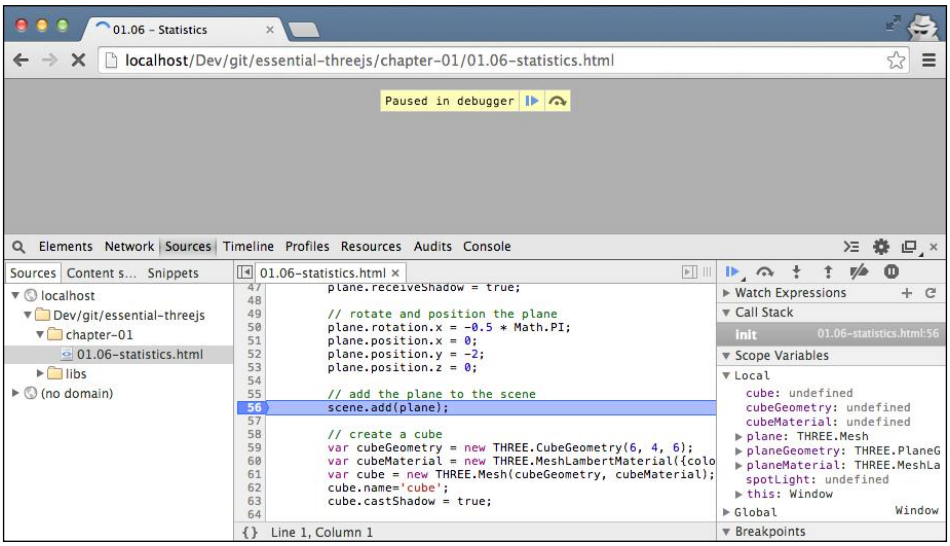

In the preceding screenshot, you can see the string we logged with the first log statement. What's more interesting though is the `console.log(cube)` output. With this log statement, all the properties from the object are logged. This way, we can quickly check the value of specific properties and explore referenced objects; we can even change them directly from here. An important aspect to keep in mind when using logging in this way is performance. When you do a lot of logging in your web application, its performance will go down. So keep in mind that you remove the logging when you move your code to production.

This is a very easy way to get extra information during runtime. If you, however, want to add some logging to the render loop, the output may be a bit overwhelming. This function will be called approximately 60 times a second, so you can imagine the amount of output you'll get. There is a very easy way to solve this by using a JavaScript debugger. In the following section, I will show an example using the debugger from Chrome, but each modern browser has a very capable debugger.

# Looking at objects with breakpoints in Chrome

When you use a debugger, you can easily access all the values of the JavaScript objects you're working with. If you run into issues that you can't easily solve by just adding log statements, using a debugger is probably a good choice.

In the following screenshot, you can see the Chrome debugger in action:




Here, we clicked on the **Sources** tab and opened the file that contained the JavaScript code. In this file, all you have to do is click on the line number where you want to add a breakpoint. When the code runs to that point, even if it is called from the `requestAnimationFrame` object, the execution will stop and you can then browse through all the variables and objects that are available. This works great whenever you run into some changed behavior.

# Summary

That was that for the first introductory chapter. In this chapter, we explained a number of different subjects. Now, we'll summarize the most important ones.

Three.js is an open source JavaScript library that can run in all modern browsers without requiring any plugins, and for older IE browsers, there is a plugin available that enables WebGL support. For mobile browsers, it's a bit different. The performance of WebGL on these platforms depends on the combination of device, OS, and browser. If you want to run the examples on a mobile device, your best bet is to go for a modern Android device and use Chrome or Firefox as the browser. If you use iOS, you're out of luck, as iOS doesn't support WebGL yet. You can, however, still use the CSS3D renderer (see *Chapter 6*, *Combining HTML and Three.js with CSS3DRenderer*), which runs really well on iOS but does have a more limited API.

If you create your first Three.js application, you can run it in a number of different ways. The best way, though, is to set up a local web server, which is very easy to do. If you can't run a local web server, there is always the option to disable some of the security policies in your browser.

To understand how Three.js works, you need to understand the basic concepts of Three.js. The first thing you do in Three.js is create a `THREE.Scene` object. This object serves as a container in which you can place the other Three.js objects, such as meshes and lights. A `THREE.Mesh` object represents a 3D object and consists of a geometry and a material. The geometry defines the shape of an object, and the material defines what it will look like.

In this chapter, we also introduced a couple of external libraries that make working with Three.js easier. You can use the dat.GUI library to easily add a simple UI element to control specific properties in your scene. To keep track of performance, we introduced the Stats.js library.

Once you try to get your scene up and running, you might run into some unexpected behavior. There are different ways to determine what is going on. You can, for instance, use the `console.log` function, or set a breakpoint in the developers section of your favorite browser.

In the next chapter, we'll show you how to create a 3D world globe and use that example to explain camera control, textures, integration with HTML5 canvas, and a number of other essential parts of Three.js.

# 2

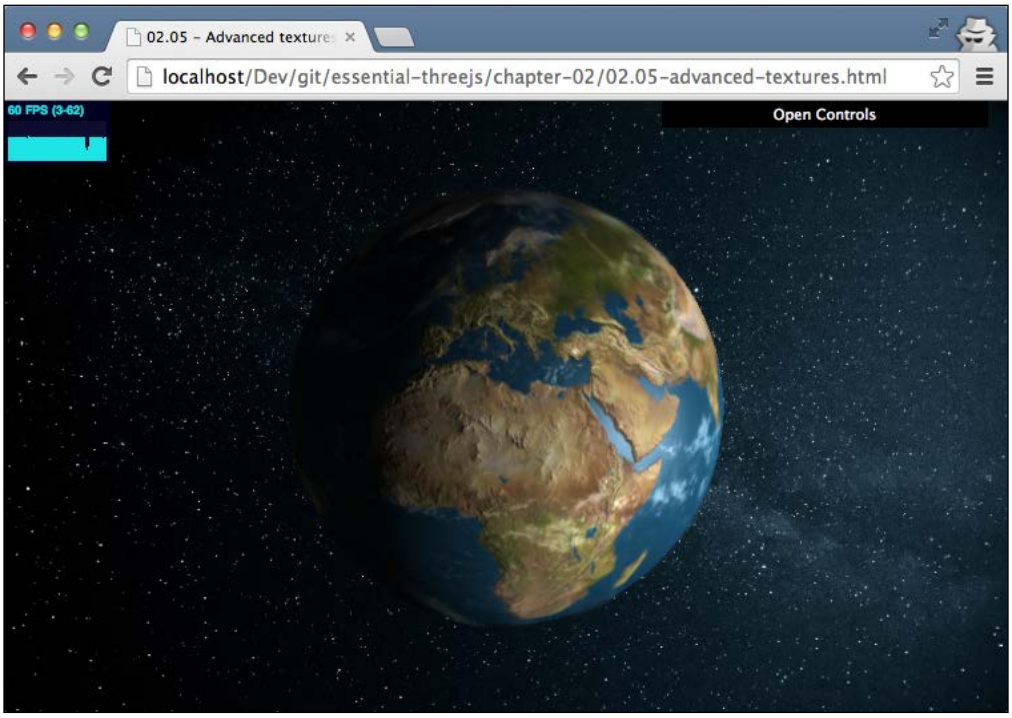
# Creating a 3D World Globe and Visualizing Open Data

In this chapter, we're going to use the example of a 3D-rendered globe to explain a number of core Three.js concepts. We'll show you how to build this example step by step and explain the following features provided by Three.js:

- Three.js provides a number of easy ways to control the camera. In this example, we'll show you how you can use the orbit camera control.
- Through the use of textures, we can quickly enhance the look of the simple meshes that are rendered. In this example, we'll show you how to use simple and advanced textures to transform a simple sphere into an object that looks like the earth.
- We'll also dive a little bit deeper into lighting. In this example, you'll see how ambient and directional lighting can be added to a scene.
- When you add a lot of meshes to a scene, the performance will suffer. Three.js allows you to combine multiple geometries with a single geometry to improve performance. We'll demonstrate how to do this.
- Most of the examples we'll discuss use a perspective camera. Three.js also provides a camera that presents us with an orthogonal projection. We'll add this camera to the scene and show some of its features.
- The normal way to render a scene is to call the `render()` function on the renderer. There is, however, also a different approach through which we can create various render or effect steps and combine these together. We'll show how you can configure and use this alternative rendering approach.

Before we show you the required JavaScript and HTML code required to accomplish the examples, let's look at the first result that we're aiming for in this chapter.

In the following screenshot, you can see a rotating 3D world, complete with clouds, mountains, and a starry background. When you open this example (the `02.05-advanced-textures.html` file), you can use the mouse to rotate and pan around it.




In the next few sections, we'll explain how to create this rotating earth. The first thing that we need to do is create a simple globe and set up the camera.

# Setting up the globe and camera controls

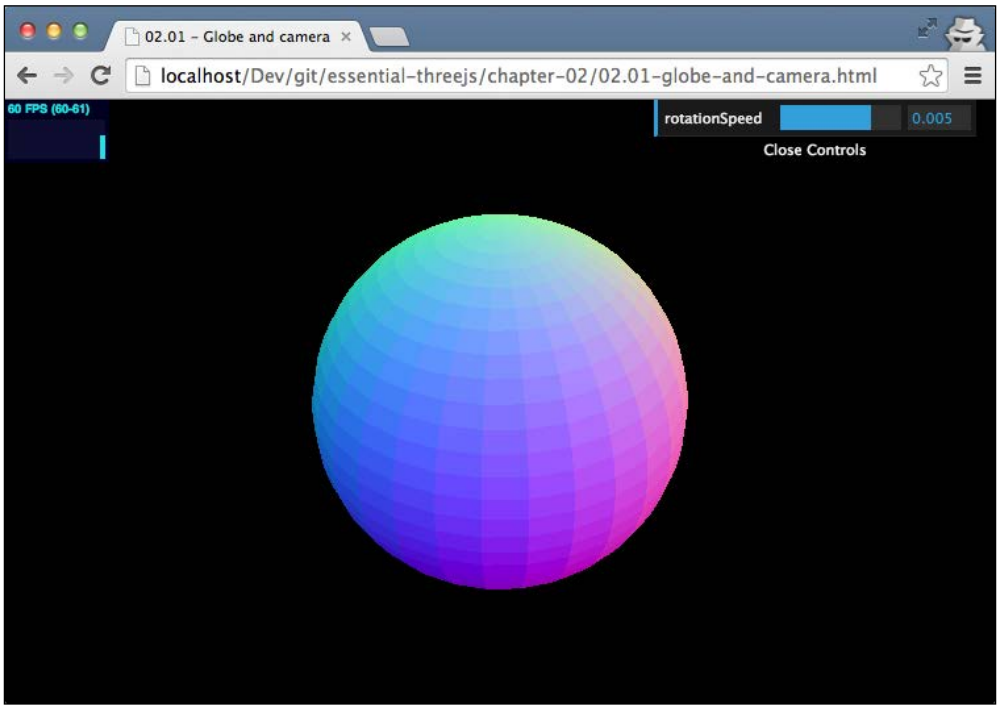
In this section, we'll create a simple sphere, add it to the scene, position the camera, and add some camera controls. First, let's create and add a sphere. This is described in the following code:

```
var sphereGeometry = new THREE.SphereGeometry(15, 30, 30);
var sphereMaterial = new THREE.MeshNormalMaterial();
var earthMesh = new THREE.Mesh(sphereGeometry, sphereMaterial);
earthMesh.name = 'earth';
scene.add(earthMesh);
```

If you look back at the previous chapter where we created the cube, you will see that we took the same approach. We first created geometry, then some material; we then combined it with a mesh and added it to the scene. The geometry in this example takes the following three arguments (note that the constructor has four arguments, but only the first argument, the radius, is required):

- The radius of the sphere
- The number of width segments
- The number of height segments

The radius defines the size of the sphere, and the segments decide how many faces the cube will be divided into. The easiest way to understand this is by just looking at the following output (the `02.01-globe-and-camera.html` file):




Here, you can see that the sphere isn't completely round but is divided into a number of squares. The number of squares is based on the parameters provided. You can also see that we've used a new kind of material: `THREE.MeshNormalMaterial`. With this material, the color of a mesh isn't determined by lighting, but is determined by its normal vector, or in other words, by the angle of the face. This is a good material to use when you're testing the position of your camera, the way the geometry looks, and whether the scene is initialized correctly.

Just this sphere by itself isn't that interesting. So, let's add the camera controls. This is described in the following code:

```
// Position the camera and point it at the center of the scene
camera.position.x = 35;
camera.position.y = 36;
camera.position.z = 33;
camera.lookAt(scene.position);

// add controls
cameraControl = new THREE.OrbitControls(camera);
```

To add a camera control, the only thing that you need to do is wrap the existing camera with a control object, in this case, a `THREE.OrbitControls` object. To use these controls, we also need to add the following line of code to the `render()` function:

```
function render() {
  ...
  cameraControl.update();
  ...
}
```

With this line added, you can now use the mouse (or even swipe actions) to move around the scene using the controls listed in the following table:

| Control | Action |
|---|---|
| Left mouse button and move | Rotate and roll the camera around the scene |
| Scroll wheel | Zoom in and zoom out |
| Middle mouse button and move | Zoom in and zoom out |
| Right mouse button and move | Pan around the scene |

So far, the example looks very basic. In the next section, we'll add the basic textures to make it look like a real planet.

# Adding basic textures to the globe

In this step, we're going to add two different textures. First, we're going to add a satellite image of the earth, and to make it look more realistic, we'll also add a couple of clouds. Let's first look at the texture of the earth that we'll use. This is shown in the following image:

To use this image as a texture in Three.js, we need to change the way we create the material. For readability, we have moved the material creation code to a separate function. This is described in the following code:

```
function createEarthMaterial() {
  // 4096 is the maximum width for maps
  var earthTexture = THREE.ImageUtils.loadTexture(
    "../assets/textures/planets/earthmap4k.jpg");

  var earthMaterial = new THREE.MeshBasicMaterial();
  earthMaterial.map = earthTexture;

  return earthMaterial;
}
```

In the `createEarthMaterial` function, we create `MeshBasicMaterial` object, a material that doesn't change the way it looks according to light sources. Before we can use an image as a texture, we first need to load it. We can do this by using the `THREE.ImageUtils.loadTexture` function, which loads the image asynchronously. The final step that we need to do is set the `map` property of the material to the texture we just created.

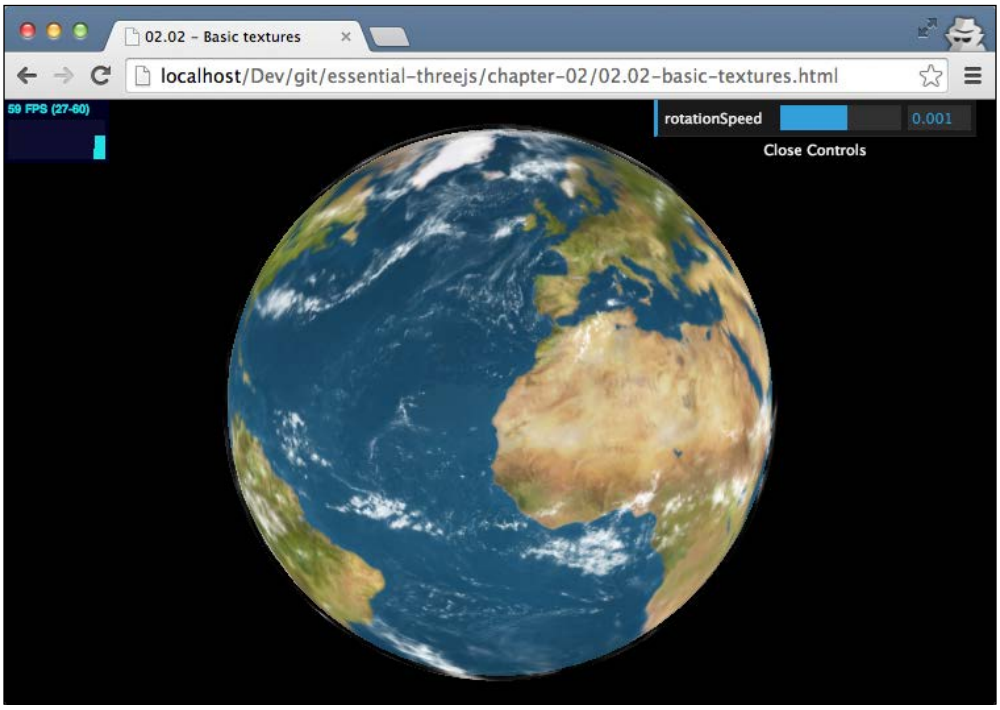
At this point, we have a sphere that looks a bit like the earth but doesn't have any clouds yet. To render the clouds, we create another sphere and position it at exactly the same location, but we make this one slightly larger. This is described in the following line of code:

```
var cloudGeometry = new
  THREE.SphereGeometry(sphereGeometry.parameters.radius*1.01,
  sphereGeometry.parameters.widthSegments,
  sphereGeometry.parameters.heightSegments);
```

Next, we define the material for this new sphere. This is described in the following code:

```
function createCloudMaterial() {
    var cloudTexture = THREE.ImageUtils.loadTexture(
      "../assets/textures/planets/fair_clouds_4k.png");
    var cloudMaterial = new THREE.MeshBasicMaterial();
    cloudMaterial.map = cloudTexture;
    cloudMaterial.transparent = true;
    return cloudMaterial;
}
```

This is almost the same process, but for this sphere, we set the material to be transparent. This is so that we can also see the earth below the clouds. This results in the following rendered scene (the `02.02-basic-textures.html` file):



This scene is better than the way it would be without textures, but it is still missing a lot. Next, we'll add some lighting and see what that does.

# Adding directional and ambient lighting

Three.js offers a number of different types of light. The following table shows the most important lights that are available:

| Name | Description |
|---|---|
| `AmbientLight` | A simple light whose color is added to the color of an object's material. |
| `PointLight` | A single point in space that emanates light evenly in all directions. |
| `SpotLight` | A light with a cone effect, for instance, a spot in the ceiling or a torch. |
| `DirectionalLight` | A light that acts like a very remote light source. All light rays run parallel to each other. The sun, for instance, can be seen as an infinite source of light. |

Before we create the lights, we first need to change the material. In the previous example, we used `THREE.MeshBasicMaterial`. Now, we'll change the material to `THREE.MeshPhongMaterial`. This material reacts to light sources, while `THREE.MeshBasicMaterial` does not. This is described in the following code:

```
var earthMaterial = new THREE.MeshPhongMaterial();
earthMaterial.map = earthTexture;
```

In this example, we'll use an `AmbientLight` object and a `DirectionalLight` object. First, we add the directional light, which represents the sun in our case. This is described in the following code:

```
var directionalLight = new THREE.DirectionalLight(0xffffff, 1);
directionalLight.position = new THREE.Vector3(100, 10, -50);
directionalLight.name = 'directional';
scene.add(directionalLight);
```

The constructor for `THREE.DirectionalLight` takes two parameters. The first one is the color and the second one is the intensity. So, if you want to have a brighter light, you can change the second parameter; if you want light of a different color, change the first parameter. After creating the light, we set the position of the light and we then add it to the scene. With this setup, the back of the earth (the side not facing this directional light) will be completely black. To make it look a little bit more realistic, we add an ambient light so that everything is a bit brighter. This is described in the following code snippet:

```
var ambientLight = new THREE.AmbientLight(0x111111);
scene.add(ambientLight);
```

`AmbientLight` only takes a single parameter, which determines the color of the light. If you now look at the scene, it will slowly start to look more real (the `02.03-add-lighting.html` file). The scene is shown in the following screenshot:


As a basic globe, this already looks pretty nice. However, without any other stars, it still looks wrong. In the next section, we'll add a starry background.

# Combining with a starry background

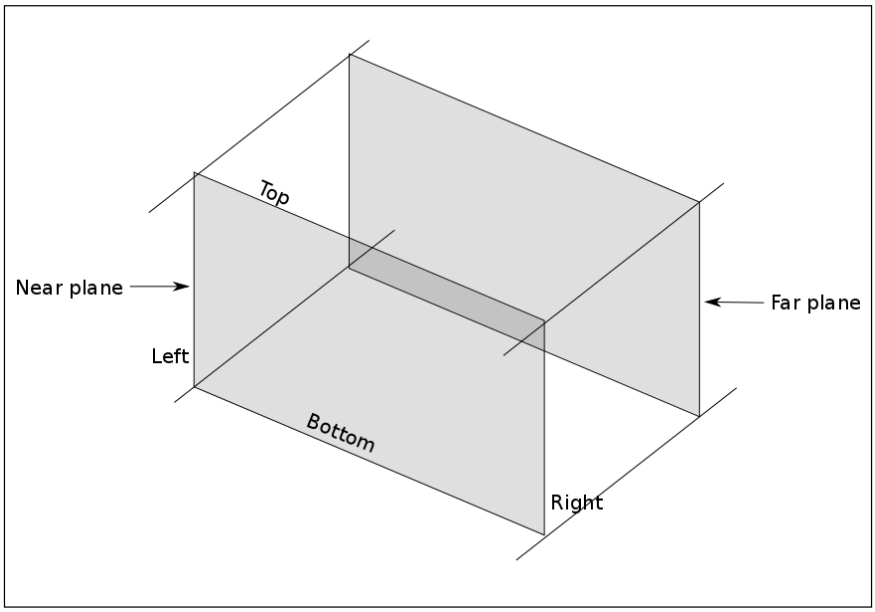
There are different approaches you can take to set up a background in Three.js. We could, for instance, create a very large cube, apply a texture to the inside of the cube, and make sure it encompasses our globe. In our case, however, we'll use an approach that'll show you a very interesting feature of Three.js: the `THREE.OrthographicCamera` object. With an orthographic camera, all elements in the scene are rendered in the same size regardless of how far it is from the camera. This is, for instance, the effect you see in some older games, such as Baldur's Gate or Civilization. We'll use this camera to show a scene where we can create a plane geometry on which we will define a starry texture. So, first off, let's create the camera. Consider the following code:

```
cameraBG = new THREE.OrthographicCamera(
          -window.innerWidth,
           window.innerWidth,
           window.innerHeight,
          -window.innerHeight,

          -10000, 10000);
cameraBG.position.z = 50;
```

What you can see here is that this camera takes other properties compared to what we've seen in the previous chapter for the perspective camera. The reason is that as the orthographic camera renders all the objects in the same size, regardless of the distance, we just need to tell the camera about the cuboid area in which we are interested. The following table explains the properties you need to specify on this camera:

| Argument | Description |
|---|---|
| `left` | This property defines the border for the leftmost position to be rendered. |
| `right` | This property defines the border for the rightmost position to be rendered. |
| `top` | This property defines the border for the topmost position to be rendered. |
| `bottom` | This property defines the border for the bottommost position to be rendered. |
| `near` | This property defines the point, based on the position of the camera, from where the scene will be rendered. |
| `far` | This property defines the point, based on the position of the camera, to which the scene will be rendered. |

It's easier to understand how this camera works by looking at the following figure:




Here, you can see the cuboid area that we defined, and this is rendered when the orthographic camera is used. Now, let's create a new scene and add a 2D plane (`THREE.PlaneGeometry`) with a starry background as the texture. This is described in the following code:

```
sceneBG = new THREE.Scene();
var materialColor = new THREE.MeshBasicMaterial({ map:
  THREE.ImageUtils.loadTexture(
  "../assets/textures/planets/starry_background.jpg")});
var bgPlane = new THREE.Mesh(
  new THREE.PlaneGeometry(1, 1), materialColor);
bgPlane.position.z = -100;
bgPlane.scale.set(
  window.innerWidth * 2, window.innerHeight * 2, 1);
sceneBG.add(bgPlane);
```

What we did here was create a simple horizontal plane with a starry texture. We moved it to the back of our planet (`position.z = -100`) and then scaled it to fill the complete screen. If we were to render this with the renderer, we would get a nice fullscreen plane that shows the image we used for the texture.

As we're not interested in just the background but want to combine the rendering of the earth with the rendered background, we can't use the renderer we've used so far. We need to use an object called `EffectComposer`. With an `EffectComposer` object, we can define various render passes, which are combined into a single image shown in the browser.

The required JavaScript for this object isn't part of the standard Three.js JavaScript file. So, we need to add the following code to the JavaScript by including statements at the top of the page:

```
<script src="../libs/EffectComposer.js"></script>
<script src="../libs/RenderPass.js"></script>
<script src="../libs/CopyShader.js"></script>
<script src="../libs/ShaderPass.js"></script>
<script src="../libs/MaskPass.js"></script>
```

The next thing we need to do is set up the `EffectComposer` object. Let's first look at the following code:

```
// setup the passes
var bgPass = new THREE.RenderPass(sceneBG, cameraBG);
var renderPass = new THREE.RenderPass(scene, camera);
renderPass.clear = false;
var effectCopy = new THREE.ShaderPass(THREE.CopyShader);
effectCopy.renderToScreen = true;

// add these passes to the composer
composer = new THREE.EffectComposer(renderer);
composer.addPass(bgPass);
composer.addPass(renderPass);
composer.addPass(effectCopy);
```

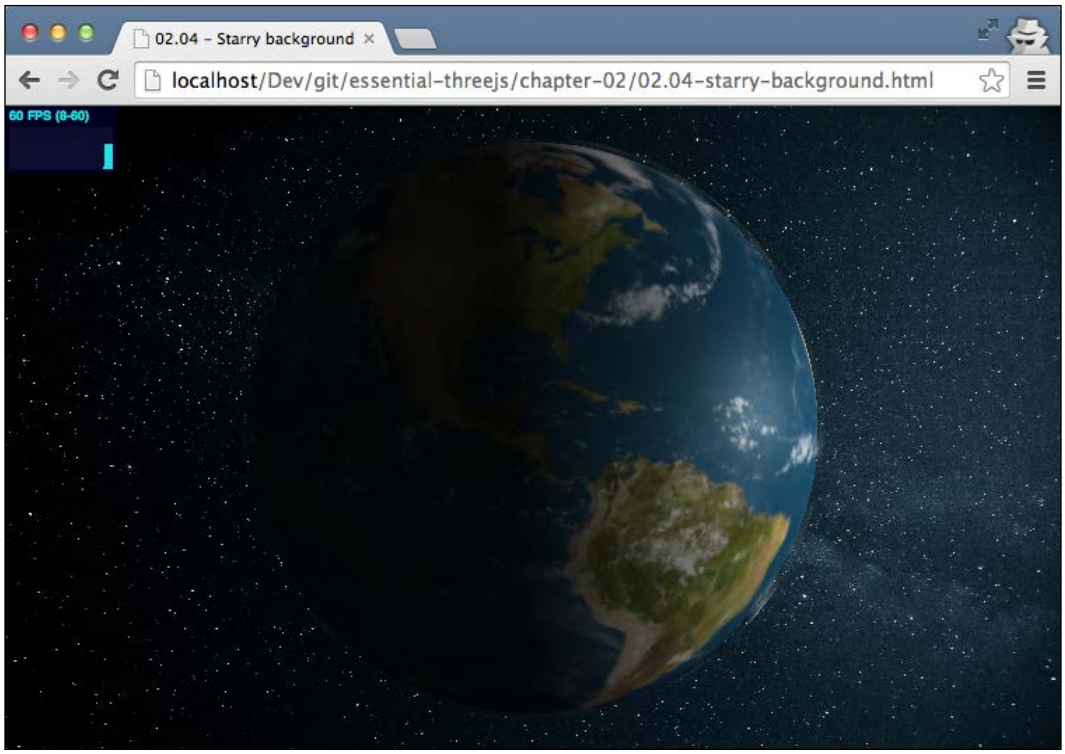
In this example, we first define two `THREE.RenderPass` objects. With a `Three.RenderPass` object, you can render a scene with a specific camera, but the result isn't rendered directly to the screen but kept internally for further processing. The normal behavior of a `Three.RenderPass` object is to clear the current output from the renderer before rendering. This is why we need to set the `renderPass.clear` property on the second `THREE.RenderPass` object. If we don't do this, we'll only see the rotating earth and not the background.

Once both the scenes have been rendered, we need a way to copy them to the screen. For this, we use the `THREE.CopyShader` object, which we can add to the `THREE.EffectComposer` object using a `THREE.ShaderPass` object, as we can't use the `THREE.CopyShader` object directly. The `THREE.ShaderPass` object doesn't pass the output to the screen automatically. To accomplish this, we set the `renderToScreen` property of the `THREE.ShaderPass` object to `true`. Now, all we need to do is create the `THREE.EffectComposer` object and add the three passes we just created in the order of their execution; so, we'll first perform background rendering, then foreground rendering, and finally copy it to the screen.

The final change that we need to make is in the render loop. Here, we replace the call to the `renderer.render` function with a call to the `composer.render` function. This is described in the following code:

```
function render() {
  ...
  renderer.autoClear = false;
  composer.render();
  ...
}
```

Additionally, as you can see in this code fragment, we also need to set the `autoClear` property of the renderer to `false`. If we don't do this, we'll only see the result from the latest `THREE.RenderPass` object. Now, when we run the example (the `02.04-starry-background.html` file), we get a beautiful-looking rotating earth in front of a starry background, as shown in the following screenshot:




With the background added, it is really starting to look like a real planet. However, when you look closely, you'll see that there is lack of depth in the texture. For instance, you can't see any mountain ranges. In the following section, we'll introduce two different texture types that will really improve how the earth will look.

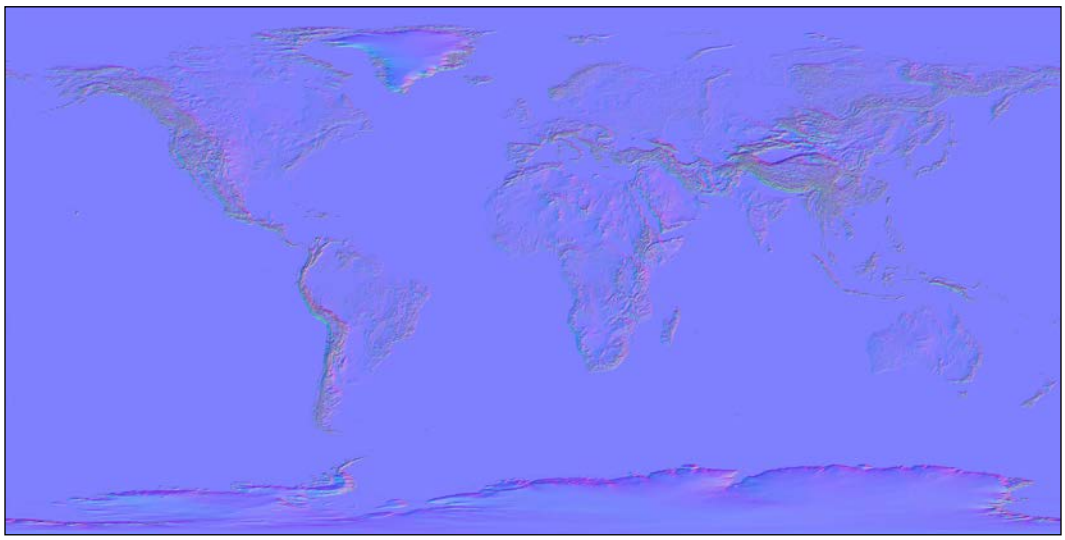
# Improving the look with more advanced textures

To further improve the model of the earth, we'll add some more depth to the mountain ranges and other elevations and change the reflectivity of the water bodies. Let's start by adding more depth to the scene.

# Using a normal map to simulate elevations

In 3D modeling, there are two different approaches to simulate the depth using a texture. The first one is a **bump map**. With a bump map, you can create a black and white texture, where the color of each pixel determines the elevation. Even though this works for simple models, it doesn't result in a very detailed depth rendering.

For a more detailed approach, we can also use a **normal map**. A normal map doesn't store the intensity of each pixel but rather stores the direction (orientation) of each pixel. The map we will use in this example is shown in the following image:



Using a normal map works in the same way as using a standard texture. Consider the following code:

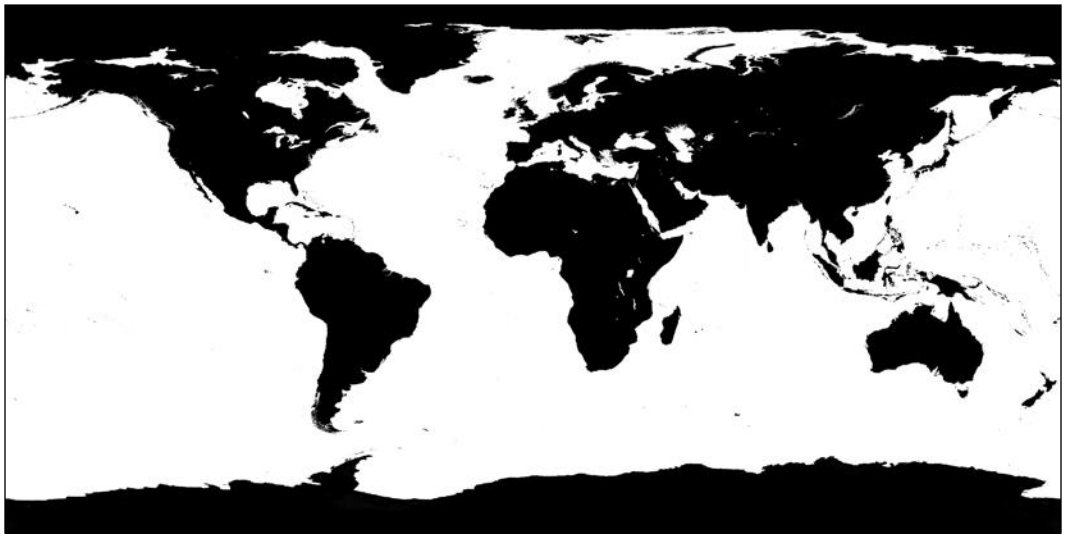
```
function createEarthMaterial() {
  ...
  var normalMap = THREE.ImageUtils.loadTexture(
    "../assets/textures/planets/earth_normalmap_flat4k.jpg");
  ...
  earthMaterial.normalMap = normalMap;
  earthMaterial.normalScale = new THREE.Vector2(0.5, 0.7);
  ...
}
```

To load the texture, you can use the `THREE.ImageUtils.loadTexture()` function and assign the result to the `normalMap` property. You can play around with how large the effect of this normal map is by using the `normalScale` property, where the first property defines scaling along the *x* axis and the second one along the *y* axis.

Before we show the result, let's quickly look at a specular map.

# Using a specular map to define the reflectivity of an area

When you look at satellite images of the earth, you can see that different areas of the earth reflect light in a different manner. In general, you can say that the oceans reflect the light from the sun very well and the continents don't. The following image shows how the picture looks. We'll use this image as an input.

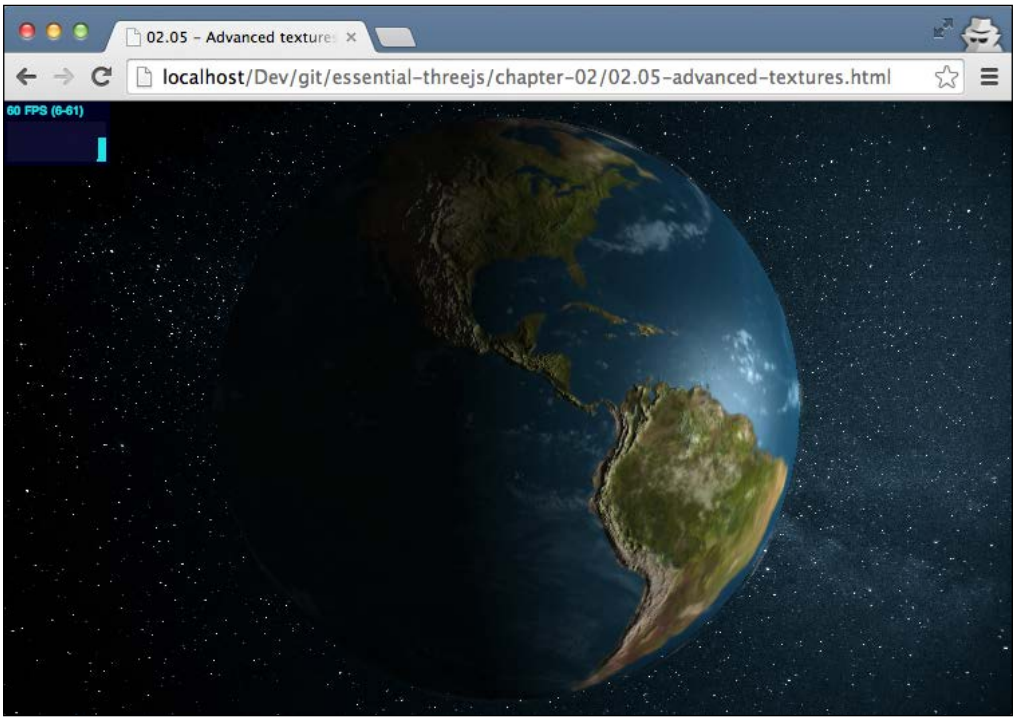


Here, you can see that the continents are black, so they won't reflect the directional light we created, and the oceans are white. To use this image to improve our globe, we set this image as a texture to the `specularMap` property of the material we're using. This is described in the following code:

```
function createEarthMaterial() {
  ...
  var specularMap = THREE.ImageUtils.loadTexture(
    "../assets/textures/planets/earthspec4k.jpg");

  earthMaterial.specularMap = specularMap;
  earthMaterial.specular = new THREE.Color(0x262626);
  ...
}
```

We load this texture the same way we've done before and assign it to the `specularMap` property. Finally, we can define the color of the reflection using the `specular` property.

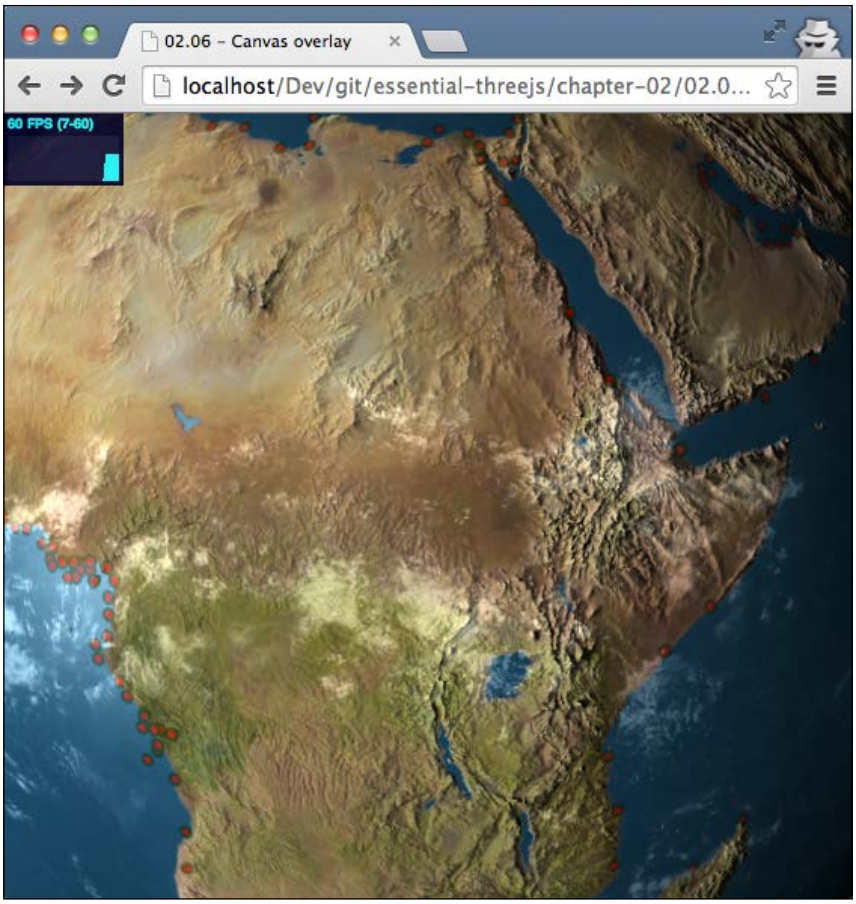
With all these features combined, we get the following screenshot as the result (the `02.05-advanced-textures.html` file):




Here, you can see that the mountain ranges in South America stand out. This is based on the normal map. You can also see that there is a small reflection in the body of water just north of South America, which isn't seen on land.

# Adding 2D information using HTML canvas as a texture

So far, we've just used the basic features of Three.js to render the globe. In this section, we'll look a bit further and use this globe as a way to visualize data. We'll show you how you can use an HTML5 canvas as a texture to visualize data on the 3D globe. What we're aiming for in this section is shown in the following screenshot (the `02.06-canvas-overlay.html` file):




All the red circles that you see in this screenshot are ports, and they are shown on the rotating globe. To get this result, we first have to get a source for the data. For this, I've used the **World Port Index** (**WPI**) database that you can download from `http://msi.nga.mil/NGAPortal/MSI.portal?_nfpb=true&_pageLabel=msi_portal_page_62&pubCode=0015`. I've converted this database into a simple CSV file that shows the following information:

```
31140;30980;ROTTERDAM;NL;51;54;N;4;29;E;192;37243/1;...
7640;6585;NEW YORK CITY;US;40;42;N;74;1;W;CP02;12335;...
```

Each line contains the location and a set of other properties for all the ports in the world. We'll use this information to create an HTML5 canvas that we'll use as a texture for the globe. The code to render this to an HTML5 canvas is as follows:

```
canvas = document.createElement("canvas");
canvas.width = 4096;
canvas.height = 2048;

var context = canvas.getContext('2d');

var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
  if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
    // Convert the CSV file to a list of arrays
    var ports = CSVToArray(xmlhttp.responseText, ";");
    // now we process each of the inidividual arrays
    // which represent a single line from the CVS file
    ports.forEach(function(e) {

      if (e[25] === 'L') {
        var posY = parseFloat(e[4] + "." + e[5]);
        var sign = e[6];
        if (sign === 'S') posY = posY * -1;

        var posX = parseFloat(e[7] + "." + e[8]);
        var sign = e[9];
        if (sign === 'W') posX = posX * -1;

        var x2 = ((4096 / 360.0) * (180 + posX));
        var y2 = ((2048 / 180.0) * (90 - posY));

        context.beginPath();
        context.arc(x2, y2, 4, 0, 2 * Math.PI);
        context.fillStyle = 'red';

        context.fill();
        context.lineWidth = 2;
        context.strokeStyle = '#003300';
        context.stroke();
      }
    });
  }
}

xmlhttp.open("GET", "../assets/data/wpi.csv", true);
xmlhttp.send();
```

Let's look at this code step by step. First, we used an `XMLHttpRequest` function to asynchronously load the CSV file that contained all the port locations by using the `xmlhttp.open` and `xmlhttp.send` methods. When the CSV file is loaded, we walk through all the lines and check whether a specific value is set to `L`. If this property is set, it means the port we're looking at is a large one. We do this so that only the large ports are shown on the globe. For each large port, we get the latitude and longitude, and we convert these to *x* and *y* coordinates. On this coordinate, we render a small circle using the `context.arc`, `context.fill`, and `context.stroke` functions. If we show the canvas at this point, we'll get something that appears like the following screenshot:


As you can see, if we plot all the large ports, we can already discern the shape of the continents.

The only thing we need to do now is assign this canvas as a texture to a sphere.

Note that we need to create a new sphere, as the `map` property of the globe is already used for the satellite image.

Consider the following code:

```
function createOverlayMaterial() {
    var olMaterial = new THREE.MeshPhongMaterial();
    olMaterial.map = new THREE.Texture(addCanvas());
    olMaterial.transparent = true;
    olMaterial.opacity = 0.6;
    return olMaterial;
}

var overlayGeometry = new THREE.SphereGeometry(
  sphereGeometry.parameter.radius,
  sphereGeometry.widthSegments,
  sphereGeometry.heightSegments);
var overlayMaterial = createOverlayMaterial();
var overlayMesh = new THREE.Mesh(
  overlayGeometry, overlayMaterial);
overlayMesh.name = 'overlay';
scene.add(overlayMesh);
```

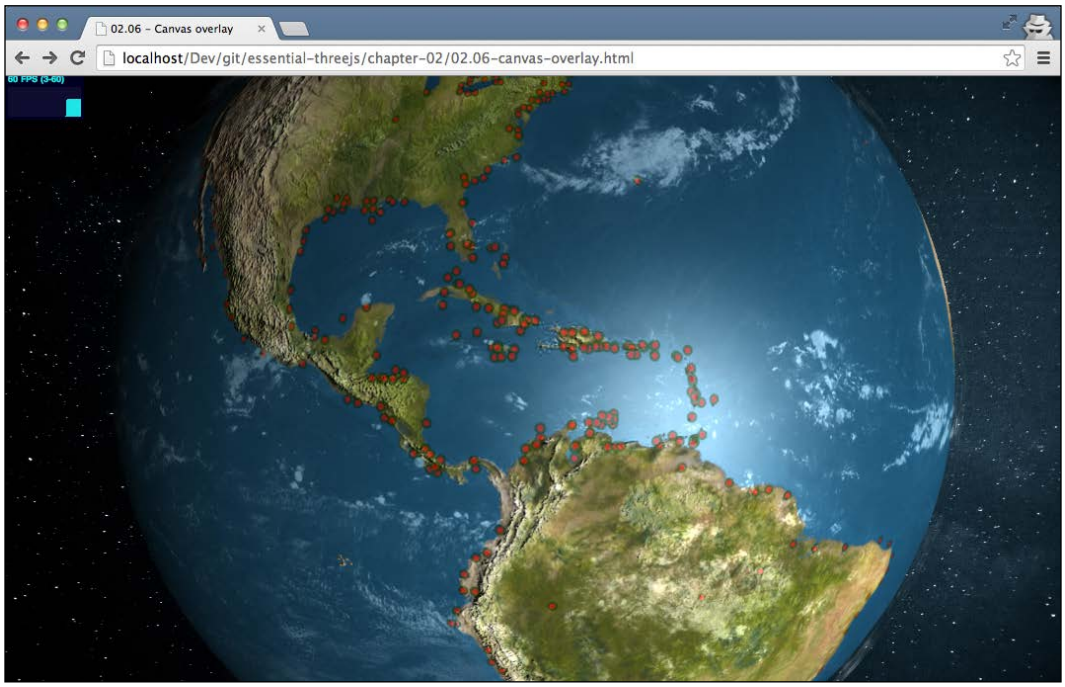
As you can see in the `createOverlayMaterial` function, we set the `map` property to the result of the `addCanvas` function, which is the `canvas` element we saw in the previous code. Now, we only need to make a small change to the `render` function, as described in the following code:

```
function render() {
  ...
  scene.getObjectByName('overlay')
    .material.map.needsUpdate = true;
  ...
}
```

Here, we tell Three.js that it needs to update the texture that we just created at the start of the rendering phase. The reason we do this is that we don't know when the CSV file is loaded and processed. With this approach, we make sure that the texture, which uses the canvas as an input, is updated on each render loop.

Note that we could also do this at the end of the `XMLHttpRequest` callback for better performance, as the picture drawn on the canvas doesn't change after it is loaded for the first time.

The following screenshot shows the final result:




# Summary

We've seen a lot of concepts of Three.js in this chapter. The most important points to remember are as follows:

- We can create a good-looking earth by just starting with a basic `THREE.SphereGeometry` object.
- Three.js provides a large number of camera controls for use. In this chapter, we used the `OrbitControls` property to quickly support the zoom and pan functions for our scene.
- There are different types of textures available. In this chapter, we used a map texture that renders an image on top of an object. Also, we used a normal texture to define detailed heights, and finally, a specular map for reflections.

- Besides the perspective camera, Three.js also provides an orthographic camera that renders objects the same size regardless of their distance from the camera.
- You can combine various renders together using the `THREE.EffectComposer` object. This is done by defining separate passes, which are combined into a single result.
- You can use the output of an HTML5 canvas as a texture. This can be easily used to plot additional data onto a 3D object.

In the next chapter, we'll show you how to create a simple 3D maze game where you'll learn more about animations, collision detection, and different light sources.

# 3
# Navigate around a Randomly Generated Maze

In the previous chapter, we showed you how to set up a scene where we showed a 3D rotating globe. You learned about the basics of materials, geometries, and lights. In this chapter, we're going to create a simple game that we'll use to explain some other features provided by Three.js. To be more specific, we're going to create a random 3D maze through which you have to navigate a rolling cube using the arrow keys on your keyboard. Through this example, we'll explore the following Three.js features:

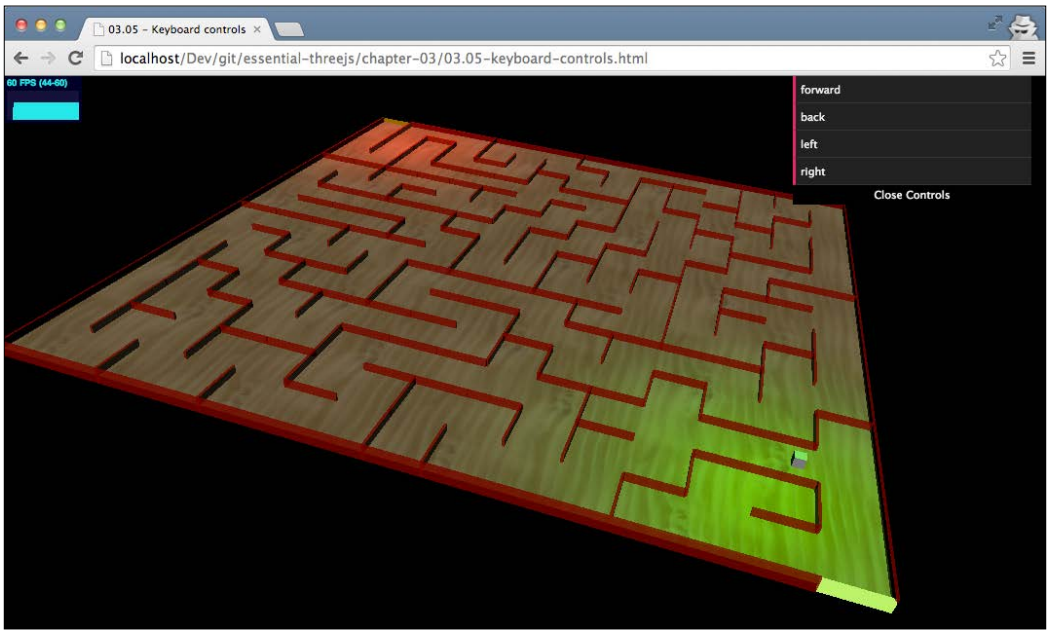
- With Three.js, it is easy to rotate and move objects around a scene. For this, there are two different approaches, which we'll explore in this chapter. First, rotating and translating objects through standard Three.js properties and second, using matrix transformations.
- We've already seen a couple of light sources. In this chapter, we'll introduce the `THREE.SpotLight` light source. This light source sends out a beam of light.
- We've already set up an animation loop in *Chapter 1*, *Get Up and Running with Three.js*. In this chapter, we'll explain an easy way to create more advanced animations by combining the `Tween.js` JavaScript library with Three.js.
- In this chapter, we'll also expand on how you can use textures in Three.js by explaining how to set up a repeating texture.
- When you create games or more advanced 3D scenes, it is important to know when objects touch each other. We'll show you an approach to set up collision detection using `THREE.Raycaster`.

- Three.js comes with a number of different camera controls. In the previous chapter, we looked at the `THREE.OrbitControls` object. In this chapter, we'll show you how you can use the `THREE.TrackballControls` object to easily move and pan around the created scene.
- Finally, we'll also explain the use of the keyboard to control Three.js elements in our scene.

Before we dive into the individual steps needed to create the 3D maze, you can already look and play around with the final result.

# The result we're aiming for in this chapter

Open up the `03.05-keyboard-controls.html` file in your browser, and you'll be shown the following screenshot:




You can move around the scene by using the arrow keys on your keyboard, and the goal is to reach the colored wall segment at the top of the maze. Once you hit a wall, you'll be moved back to the starting point.

Now, let's look at the steps you need to take to create this interactive scene. The first thing we need to do is create the maze structure by using some of Three.js standard geometries.

# Creating the maze

A maze is a rather simple shape that consists of a number of walls and a floor. So, what we need is a way to create these shapes. Three.js, not very surprisingly, doesn't have a standard geometry that will allow you to create a maze, so we need to create this maze by hand. To do this, we need to take two different steps:

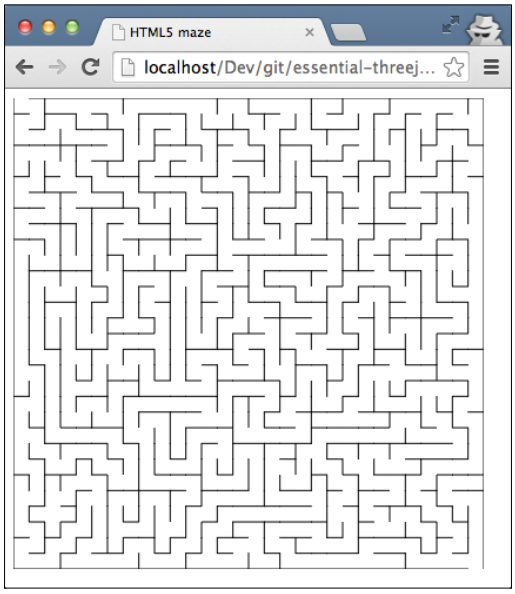
1. Find a way to generate the layout of the maze so that not all the mazes look the same.
2. Convert that to a set of cubes (`THREE.BoxGeometry`) that we can use to render the maze in 3D.

There are many different algorithms that we can use to generate a maze, and luckily there are also a number of open source JavaScript libraries that implement such an algorithm. So, we don't have to start from scratch. For the example in this book, I've used the following **random-maze-generator** project that you can find on GitHub at the following link:

```
https://github.com/felipecsl/random-maze-generator
```

# Generating a maze layout

Without going into too much detail, this library allows you to generate a maze and render it on an HTML5 canvas. The result of this library looks something like the following screenshot (the `03.0A-maze.html` file):



You can generate this by just using the following JavaScript:

```
var maze = new Maze(document, 'maze');
 maze.generate();
 maze.draw();
```

Even though this is a nice looking maze, we can't use this directly to create a 3D maze. What we need to do is change the code the library uses to write on the canvas, and change it to create Three.js objects. This library draws the lines on the canvas in a function called `drawLine`:

```
drawLine: function(x1, y1, x2, y2) {
  self.ctx.beginPath();
  self.ctx.moveTo(x1, y1);
  self.ctx.lineTo(x2, y2);
  self.ctx.stroke();
}
```

If you're familiar with the HTML5 canvas, you can see that this function draws lines based on the input arguments. Now that we've got this maze, we need to convert it to a number of 3D shapes so that we can render them in Three.js.

# Converting the layout to a 3D set of objects

To change this library to create Three.js objects, all we have to do is change the `drawLine` function to the following code snippet:

```
drawLine: function(x1, y1, x2, y2) {
  var lengthX = Math.abs(x1 - x2);
  var lengthY = Math.abs(y1 - y2);

  // since only 90 degrees angles, so one of these is always 0
  // to add a certain thickness to the wall, set to 0.5
  if (lengthX === 0) lengthX = 0.5;
  if (lengthY === 0) lengthY = 0.5;

  // create a cube to represent the wall segment
  var wallGeom = new THREE.BoxGeometry(lengthX, 3, lengthY);
  var wallMaterial = new THREE.MeshPhongMaterial({
    color: 0xff0000,
    opacity: 0.8,
    transparent: true
  });

  // and create the complete wall segment
  var wallMesh = new THREE.Mesh(wallGeom, wallMaterial);

  // finally position it correctly
  wallMesh.position = new THREE.Vector3(
          x1 - ((x1 - x2) / 2) - (self.height / 2),
```

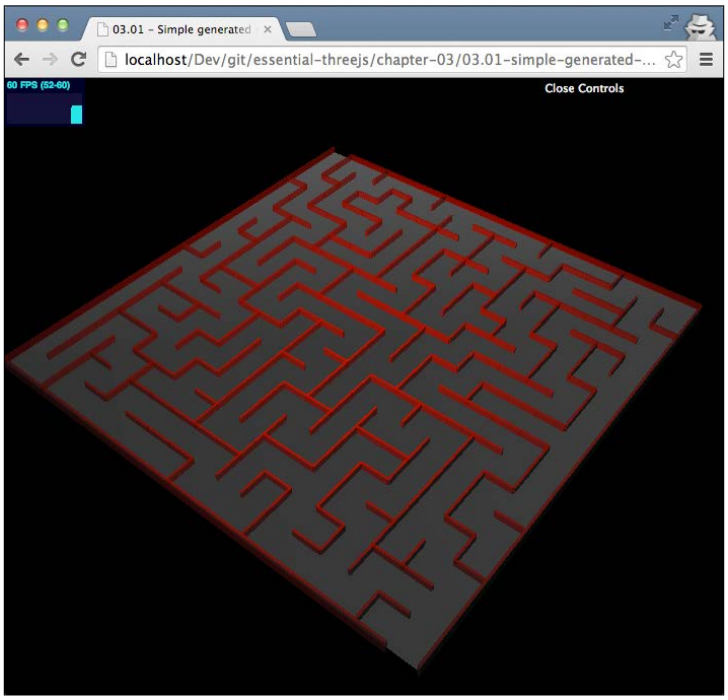
```
            wallGeom.height / 2,
             y1 - ((y1 - y2)) / 2 - (self.width / 2));

    self.elements.push(wallMesh);
    scene.add(wallMesh);
  }
```

In this new `drawLine` function, instead of drawing on the canvas, we create a `THREE.BoxGeometry` object whose `length` and `depth` are based on the supplied arguments. Using this geometry, we create a `THREE.Mesh` object and use the `position` attribute to position the mesh on a specific points with the *x*, *y*, and *z* coordinates. Before we add the mesh to the scene, we add it to the `self.elements` array. We don't need this at this moment, but we'll use this later on in this chapter for collision detection.

Now we can just use the following code snippet to create a 3D maze:

```
var maze = new Maze(scene,17, 100, 100);
maze.generate();
maze.draw();
```

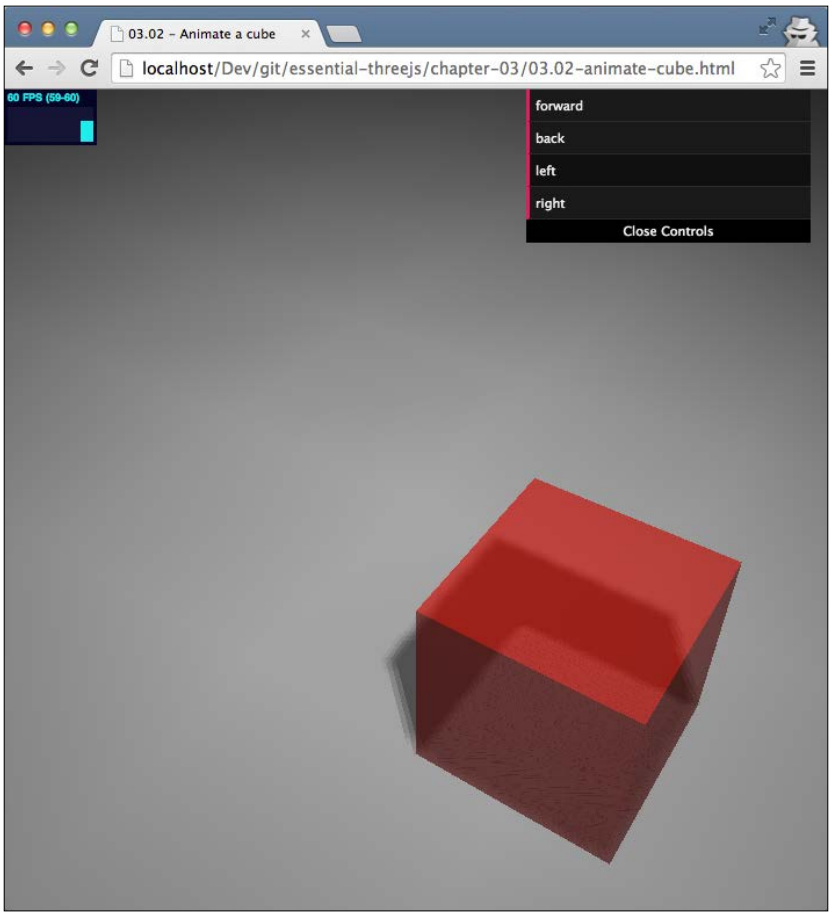
As you can see, we've also changed the input arguments. These properties now define the scene to which the maze should be added and the size of the maze. The result from these changes can be seen when you open the example file: `03.01-simple-generated-maze.html`. Have a look at the following screenshot:



Every time you refresh, you'll see a newly generated random maze. Now that we've got our generated maze, the next step is to add the object that we'll move through the maze.

# Animating the cube

Before we dive into the code, let's first look at the result we're aiming for. Open the example file, `03.02-animate-cube.html`, and you'll see something like the following screenshot:




Using the controls at the top-right corner, you can move the cube around. What you'll see is that the cube rotates around its edges, not around its center. In this section, we'll show you how to create that effect. Let's first look at the default rotation, which is along an object's central axis, and the translation behavior of Three.js.

# The standard Three.js rotation behavior

Let's first look at all the properties you can set on `THREE.Mesh`. They are shown as follows:

| Function/property | Description |
|---|---|
| `position` | This property refers to the position of an object, which is relative to the position of its parent. In all our examples, so far the parent is `THREE.Scene`. |
| `rotation` | This property defines the rotation of `THREE.Mesh` around its own *x*, *y*, or *z* axis. |
| `scale` | With this property, you can scale the object along its own *x*, *y*, and *z* axes. |
| `translateX(amount)` | This property moves the object by a specified amount over the *x* axis. |
| `translateY(amount)` | This property moves the object by a specified amount over the *y* axis. |
| `translateZ(amount)` | This property moves the object by a specified amount over the *z* axis. |

If we want to rotate a mesh around one of its own axes, we can just call the following line of code:

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

We've used this to rotate the ground area from a horizontal position to a vertical one. It is important to know that this rotation is done around its own internal axis, not the *x*, *y*, or *z* axis of the scene. So, if you first do a number of rotations one after another, you have to keep track at the orientation of your mesh to make sure you get the required effect. Another point to note is that rotation is done around the center of the object—in this case the center of the cube. If we look at the effect we want to accomplish, we run into the following two problems:

- First, we don't want to rotate around the center of the object; we want to rotate around one of its edges to create a walking-like animation
- Second, if we use the default rotation behavior, we have to continuously keep track of our orientation since we're rotating around our own internal axis

In the next section, we'll explain how you can solve these problems by using matrix-based transformations.

# Creating an edge rotation using matrix-based transformation

If we want to perform edge rotations, we have to take the following few steps:

- If we want to rotate around the edge, we have to change the center point of the object to the edge we want to rotate around.
- Since we don't want to keep track of all the rotations we've done, we'll need to make sure that after each rotation, the vertices of the cube represent the correct position.
- Finally, after we've rotated around the edge, we have to do the inverse of the first step. This is to make sure the center point of the object is back in the center of the cube so that it is ready for the next step.

So, the first thing we need to do is change the center point of the cube. The approach we use is to offset the position of all individual vertices and then change the position of the cube in the opposite way. The following example will allow us to make a step to the right-hand side:

```
cubeGeometry.applyMatrix(new THREE.Matrix4().makeTranslation
                         (0, width / 2, width / 2));
cube.position.y += -width / 2;
cube.position.z += -width / 2;
```

With the `cubeGeometry.applyMatrix` function, we can change the position of the individual vertices of our geometry. In this example, we will create a translation (using `makeTranslation`), which offsets all the *y* and *z* coordinates by half the width of the cube. The result is that it will look like the cube moved a bit to the right-hand side and then up, but the actual center of the cube now is positioned at one of its lower edges. Next, we use the `cube.position` property to position the cube back at the ground plane since the individual vertices were offset by the `makeTranslation` function.

Now that the edge of the object is positioned correctly, we can rotate the object. For rotation, we could use the standard `rotation` property, but then, we will have to constantly keep track of the orientation of our cube. So, for rotations, we once again use a matrix transformation on the vertices of our cube:

```
cube.geometry.applyMatrix(new
                          THREE.Matrix4().makeRotationX(amount);
```

As you can see, we use the `makeRotationX` function, which changes the position of our vertices. Now we can easily rotate our cube, without having to worry about its orientation. The final step we need to take is reset the cube to its original position; taking into account that we've moved a step to the right, we can take the next step:

```
cube.position.y += width/2; // is the inverse + width
cube.position.z += -width/2;
cubeGeometry.applyMatrix(new THREE.Matrix4().makeTranslation(0, -
  width / 2, width / 2));
```

As you can see, this is the inverse of the first step; we've added the width of the cube to `position.y` and subtracted the width from the second argument of the translation to compensate for the step to the right-hand side we've taken.

If we use the preceding code snippet, we will only see the result of the step to the right. In the next section, we'll use the `Tween.js` JavaScript library to animate the step.

# Using Tween.js to add an animation

We've created a function that takes care of a single step. This function makes the appropriate matrix transformations and sets up the animation. The following code fragment shows this function, where we've left out the transformations for clarity:

```
function takeStepRight(cube, start, end, time) {
  var cubeGeometry = cube.geometry;
  var width = 4;
  if (!isTweening) {
    var tween = new TWEEN.Tween({
      x: start,
      cube: cube,
      previous: 0
    })
      .to({
        x: end
      }, time)
      .easing(TWEEN.Easing.Linear.None)
      .onStart(function() {
                    // change center point to edge
      })
      .onUpdate(function() {
           var amount = -(this.x - this.previous);
          // do rotation based on amount
      })
      .onComplete(function() {
        // reset center point to center of cube
        isTweening = false;
      })
```

```
            .start();
        }
    }
```

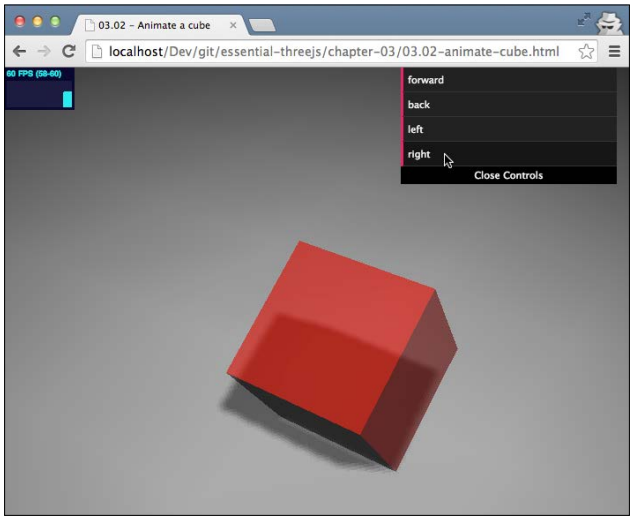
So, whenever we take a step to the right-hand side, we create a new tween. A tween can be used to change the value of a variable from one value to another over a period of time. In this example, we created a tween that changes the *x* variable from the provided `start` to `end` value within a duration of `time`. With a tween, we can also define how the variable changes. Whether it changes quickly in the beginning and slowly at the end, or, for instance, if it changes exponentially. This is called easing, and we can define this on our tween using the `easing` function. In this case, we use `TWEEN.Easing.Linear.None`, which means that the value changes in a linear fashion. We can also use exponential, sinusoidal, and many more easing methods.

Using the `onStart` function, we define what we want to do when this tween is started. In this case, we changed the center point as we explained in the previous section. Next in the `onUpdate` function, which is called a number of times during the animation, we changed the rotation based on the current value of the `x` property, just as we explained in the previous section. Finally, using the `onEnd` function, we reset the center of the cube. With `Tween.start()`, we started the animation directly.

The last thing we need to do is add one line to the `render()` function to update this animation, which in turn will call the function supplied to `onUpdate`:

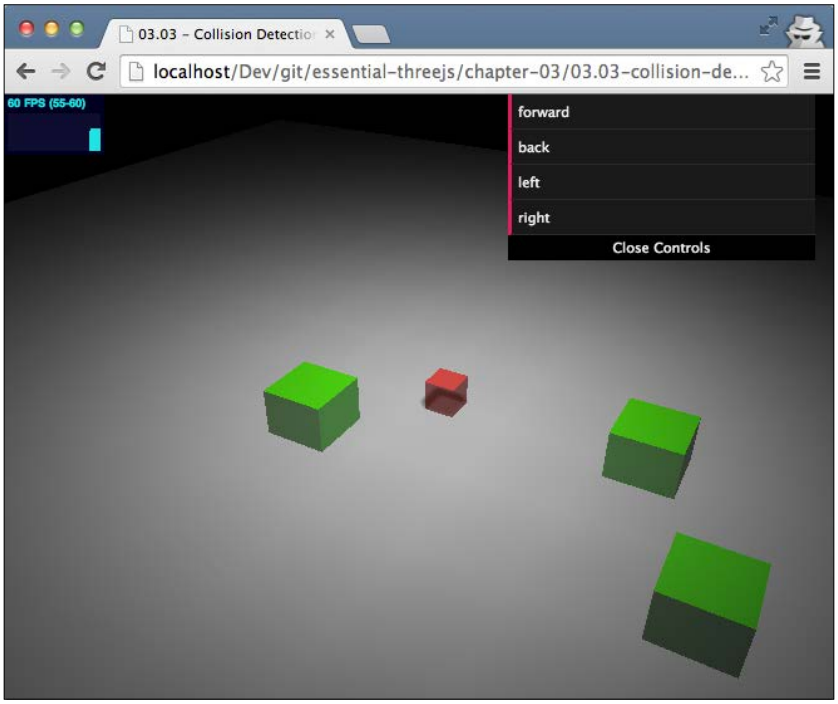
```
TWEEN.update();
```

Now, whenever we take a step, a tween will be created that animates the rotation of the cube around its edge to the new position. You can see the result in the example file, `03.02-animate-cube.html`. Have a look at the following screenshot:

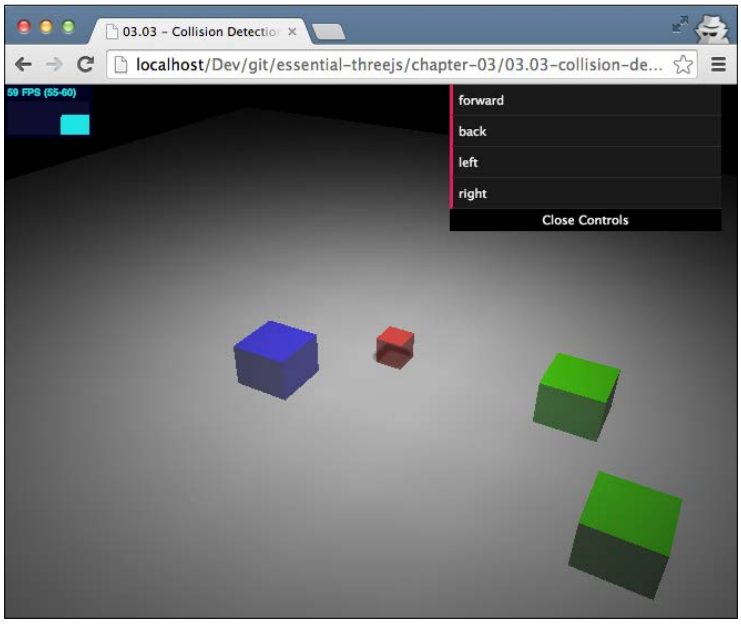


# Setting up collision detection

Now that we've got our animation up and running for our cube, we can look at how we can detect collisions. For this, we're going to use the `THREE.Raycaster` object. With this object, we can send out a ray from a specific point along a certain direction and get a list of all objects that were intersected by the ray. This is often used to select objects using the mouse. If you open the example `03.03-collision-detection.html` file, you can see both the object selection and collision detection in action. Have a look at the following screenshot:



With the controls in the menu on the top-right corner, you can move the red cube around. As soon as it hits one of the green cubes, the red cube is put back in its original position. Additionally, you can use the mouse to click on the green cubes, which will change their color to blue, as shown in the following screenshot:




We'll first show you how we can use the `THREE.Raycaster` object to select objects; after that, we'll show you how you can use pretty much the same approach to implement collision detection.

# Selecting objects

To select objects, we have to take a couple of steps. Let's first look at the following code snippet, and then we'll walk you through the various steps:

```
var projector = new THREE.Projector();

function onDocumentMouseDown(event) {

  var vector = new THREE.Vector3(
       (event.clientX / window.innerWidth) * 2 - 1,
      -(event.clientY / window.innerHeight) * 2 + 1,
       0.5);
  projector.unprojectVector(vector, camera);
```

```
    var raycaster = new THREE.Raycaster(
                    camera.position,
                    vector.sub(camera.position).normalize());
    var intersects =
                     raycaster.intersectObjects(collidableMeshList);

    if (intersects.length > 0) {
      intersects[0].object.material.transparent = true;
      intersects[0].object.material.color =
                                  new THREE.Color(0x0000ff);
    }
  }
  window.onmousedown = onDocumentMouseDown;
```

So, what is done in this code fragment? This is explained as follows:

- First, we created a `THREE.Projecter` object, which we can use to convert the coordinates where you've clicked into the coordinates in your 3D scene.
- Using this projector, we convert the position where we clicked (`vector`) into a position in the scene by using the `unprojectVector` function.
- Now, we can create a `THREE.Raycaster` object to cast rays in the direction we clicked from the point of view of the camera.
- Using the `raycaster.intersectObjects` function, we can check whether the ray we cast intersects any of the objects that are in the `collidableMeshList` array. Any objects that are intersected by the ray are returned in the form of an array of objects from this function.
- If there are objects that are intersected (`intersects.length > 0`) by our ray, we change the color of the first one, which is the nearest one, to blue.

As you can see, this is a fairly straightforward approach to select objects in our scene and do something with the selected object. We use pretty much the same approach for detecting collisions.

# Detecting collisions

If we want to detect a collision in our scene between the cube and the walls, we can use an approach that is pretty much the same as the one we've used in the previous section. Let's walk you step by step through the `detectCollision` function, which handles collision detection in our examples:

```
var cube = scene.getObjectByName('cube');
var originPoint = cube.position.clone();
```

The first thing we need to do is get the current position of our cube. Now, we're going to send rays from the center of our cube to each vertex and check whether they intersect with one of our walls:

```
for (var vertexIndex = 0;
         vertexIndex < cube.geometry.vertices.length;
         vertexIndex++) {
```

For each vertex, we create a `THREE.Raycaster` object, just as we did in the previous code fragment, send a ray to one of the vertices, and check whether we intersect with one of our wall segments, which we stored in the `collidableMeshList` array:

```
  var localVertex = cube.geometry.vertices[vertexIndex].clone();
  var globalVertex = localVertex.applyMatrix4(cube.matrix);
  var directionVector = globalVertex.sub(cube.position);

  var ray = new THREE.Raycaster(
              originPoint, directionVector.clone().normalize());
  var collisionResults =
              ray.intersectObjects(collidableMeshList);
```

The ray sent by the `THREE.Raycaster` object doesn't stop when it reaches the vertex it is aimed at. So the `collisionResults` object will also contain objects that are hit by the ray but haven't collided with the cube yet. So, additionally, we check the distance of the object we intersected with. If that is smaller than the distance between the center of the cube and the vertex, it means we have a collision. Have a look at the following code snippet:

```
  if (collisionResults.length > 0
    && collisionResults[0].distance < directionVector.length()) {
```

At this point, we have a collision; for our maze, it means we stop all the current running animations, remove the cube from its current position, and position it back at the beginning:

```
    var tweens = TWEEN.getAll();

    if (tweens.length > 0) {

      tweens[0].stop();
      TWEEN.removeAll();
      isTweening = false;

      scene.remove(cube);
      cube = createCube();
    }
  }
}
```

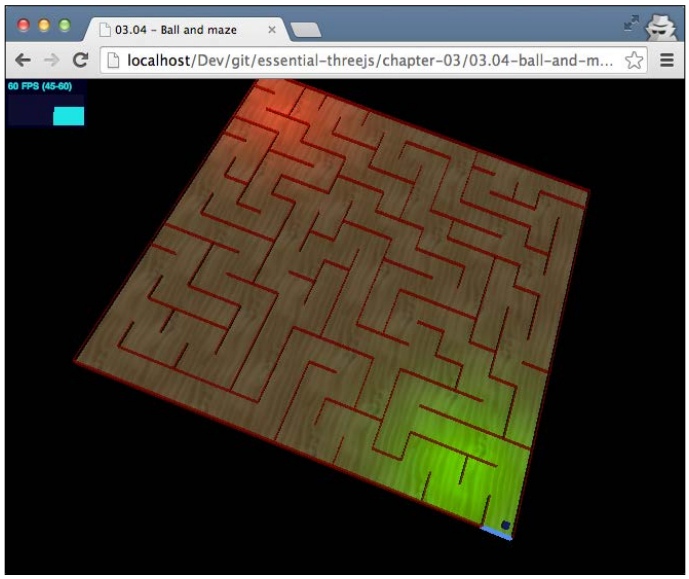
With this approach, using collision detection is very easy. There is also, however, a downside to detecting collisions like this. It is a fairly resource-intensive process. The more complex shapes become, or when the number of shapes increases, the more time it'll cost to determine whether there are collisions. There are different ways in which you can make collision detection less resource intensive. For complex shapes, you could, for instance, use an invisible bounding box, instead of the actual complex shapes. Going into details here is a bit out of the scope of this book. The main thing to remember is that collision detection comes at a price, and you should really look at the performance implications when using this in complex scenes and scenarios.

All parts of the maze and the functionality are pretty much done. We can generate a maze, move the cube around, and detect collisions. Now let's improve what the maze would look like.

# Adding textures and improving the lighting

So far, the maze looks rather simple. We've red walls on top of a gray ground plane. In this section, we'll improve how the scene looks with a couple of simple steps. First, we'll set up a texture for the ground plane, and after that, we'll add some additional lights to the scene.

At the end of these changes, our maze will look something like the following screenshot:



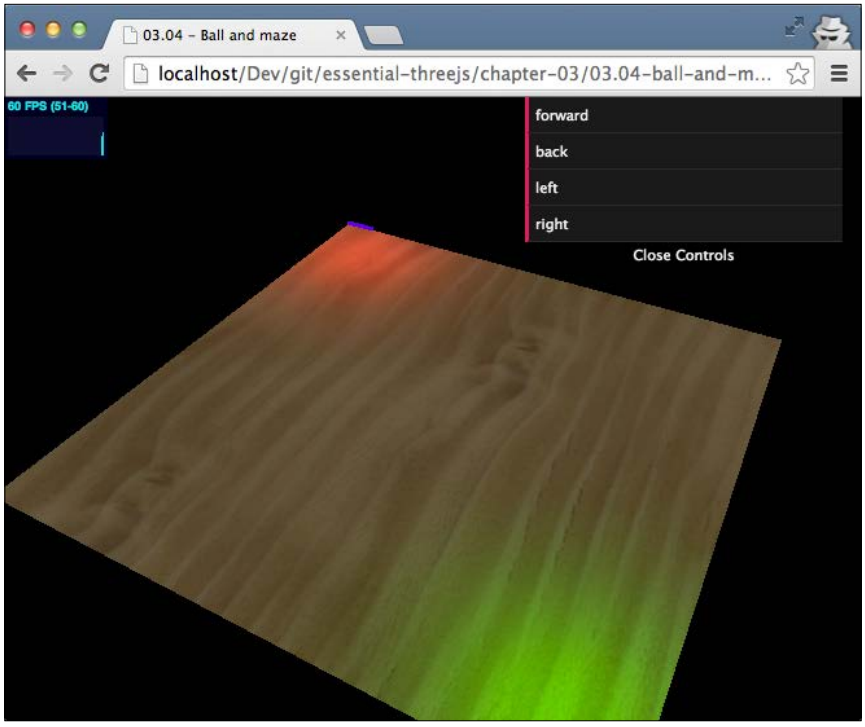

We will start by setting up the texture for the ground plane.

# Adding a repeating texture

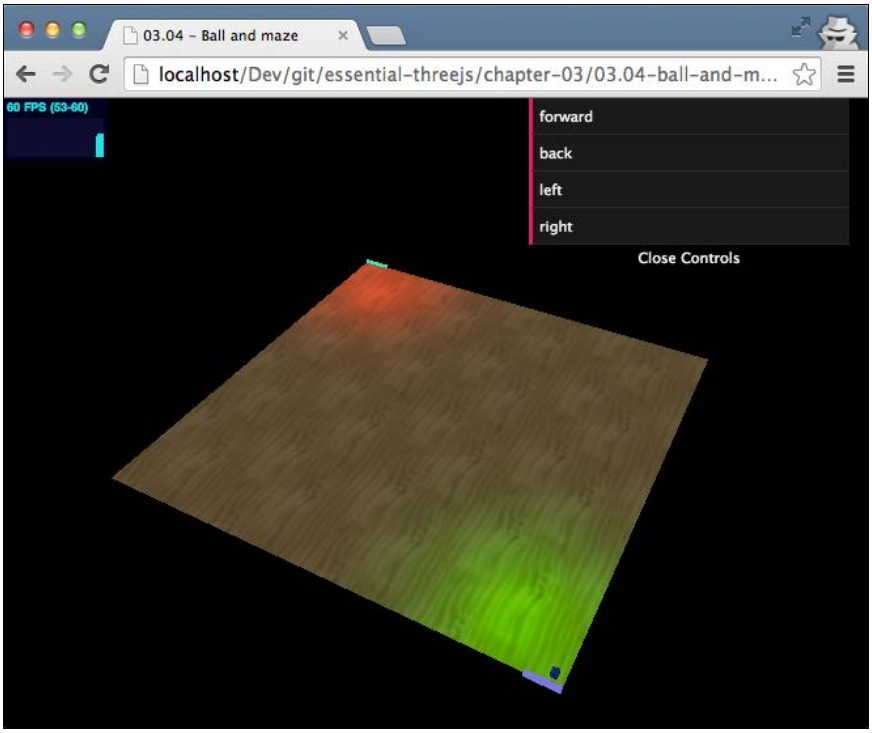
To create a nice wooden texture, as you can see in the previous screenshot, you only have to change the material you use for the ground plane. The changed code now looks like the following code snippet:

```
var planeMaterial = new THREE.MeshPhongMaterial({
  color: 0xffffff
});
planeMaterial.map = THREE.ImageUtils.loadTexture(
            "../assets/textures/wood_1-1024x1024.png")
planeMaterial.map.wrapS = THREE.RepeatWrapping;
planeMaterial.map.wrapT = THREE.RepeatWrapping;
planeMaterial.map.repeat.set(4, 4);
```

Here, you can see that we have used the `map` property of the material to point to the image containing our texture, just as we did in the previous chapter. The standard behavior of Three.js is to stretch the texture to the size of the object. This results in a scene that looks like the following screenshot:



Even though the result looks nice, it looks very zoomed in. By setting the `planeMaterial.map.wrapS` and `planeMaterial.map.wrapT` properties to `THREE.RepeatWrapping`, we tell Three.js that we want to repeat the texture along its *x* and *y* axes. We can specify the number of times the texture is repeated by using the `map.repeat` property. In this example, we repeat the texture four times along the *x* and *y* axes. As you can see in the following screenshot, the texture is repeated:




Now that we've set up the texture for the floor, the next step is to add the red and green light sources at the start and end positions.

# Setting up the light sources

If you look back at the earlier screenshots, you will see that we've added green and red lights to the bottom-left corner and top-right corner, respectively. To accomplish this, we've used a `THREE.SpotLight` object. This light source behaves like a flashlight that emits a beam of light. Using a `THREE.SpotLight` object is pretty much the same as the other lights we've seen so far:

```
// add spotlight for the finish line
var finishLight = new THREE.SpotLight(0xff0000);
```

```
finishLight.position.set(-50, 70, -50);
finishLight.castShadow = true;
finishLight.intensity = 0.5;

var finishTarget = new THREE.Object3D();
finishTarget.position.set(-60, 0, -60);
finishLight.target = finishTarget;

scene.add(finishLight);
```

In this code fragment, we've created a `THREE.SpotLight` object that emits red light. With the `position` property, we set the light above the top-left area of our maze, and we set the intensity of the light to `0.5` to make the light a bit softer. By default, all lights point to the center of the scene; using the `target` property, we can tell the light where it needs to look. A light source expects `THREE.Object3D`, or any of its subclasses, such as `THREE.Mesh`, as its target. In our case, we create a new `THREE.Object3D` object and set it as the `target` property. The interesting part is that if we now move the `finishTarget` object, our light will automatically follow it around.

The previous code snippet was used to add the red light; we've used the same approach to add the green light at the bottom-left corner. For a complete overview of the properties you can set on a `THREE.SpotLight` object, look at the following table:

| Property | Description |
|---|---|
| `castShadow` | Set this property to `true` to have this light cast shadows. |
| `shadowCameraNear` | This property signifies the distance from the light that shadows should be created. |
| `shadowCameraFar` | This property refers to the distance from the light that shadows should be created. |
| `shadowCameraFov` | This property tells us how wide the area is for which shadows need to be created. |
| `target` | This property determines where the light is looking. |
| `shadowBias` | This property can be used to slightly offset the position of the rendered shadow. |
| `angle` | This property tells us how wide the beam of light is. It is measured in radians. Its default value is *Math.PI/3*. |
| `exponent` | A light is aimed at a specific target. The farther away the light is from this direction, the more its intensity will decrease. This value determines how fast the light's intensity will decrease. |
| `onlyShadow` | If this property is set to `true`, the light will only cast a shadow. |

| Property | Description |
|---|---|
| `shadowCameraVisible` | If this property is set to `true`, you will see how and where this light source casts a shadow. This is very easy for debugging purposes. |
| `shadowDarkness` | Its default value is 0.5. This property defines how dark the shadow rendered is. |
| `shadowMapWidth` | This property determines how many pixels are used to create the shadow. Increase this when the shadow has jagged edges or doesn't look smooth. |
| `shadowMapHeight` | This property determines how many pixels are used to create the shadow. Increase this when the shadow has jagged edges or doesn't look smooth. |

In the next section, we'll add trackball controls that you can use to move and pan around the scene, and we'll add some keyboard controls that you can use to move the cube around.

# Adding trackball and keyboard controls

All that is left to do now is make it a bit friendlier to use. To do this, we'll add `THREE.TrackballControls`, which you can use to pan and move the camera around, and we'll configure some keyboard controls that you can use to move the cube around.

## Adding trackball controls to the camera

With the trackball controls, you can very easily use your mouse to move the camera around the scene. The following table shows how you can move the camera around when this control is used:

| Control | Action |
|---|---|
| Left mouse button and move | Rotate and roll the camera around the scene |
| Scroll wheel | Zoom in and zoom out |
| Middle mouse button and move | Zoom in and zoom out |
| Right mouse button and move | Pan around the scene |

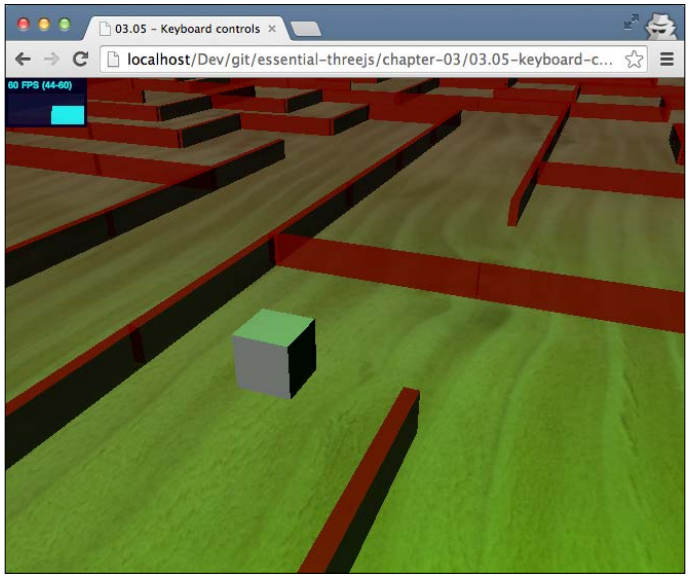
To configure this control, you have to take two steps. The first one is to wrap the camera:

```
controls = new THREE.TrackballControls( camera );
```

Next, we have to make sure that we update the control in our render loop:

```
controls.update();
```

And that's it. Now we can very easily zoom in and pan around the maze as shown in the following screenshot:




So far, you had to use the menu to move the cube around. Even though this works, it isn't the most user-friendly way. In the next section, we will show you how you can use the arrows on your keyboard to move the cube around.

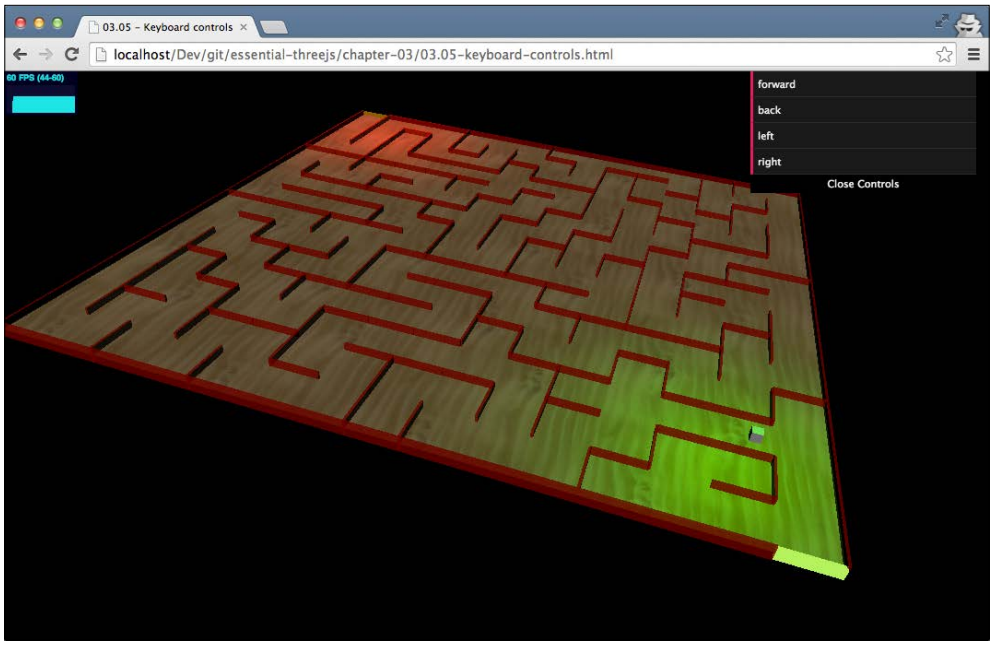
# Configuring keyboard controls

As you've seen before, we've created moving animations for each step the cube can take. If we want it to move to the left-hand side, we call the `takeStepLeft` function, for a move to the right-hand side, the `takeStepRight` function, and so on. The code to do this is actually very easy:

```
document.onkeydown = checkKey;

function checkKey(e) {
  if (e.keyCode == '37') {
```

```
    // left
    takeStepLeft(scene.getObjectByName('cube'),
      0, 0.5 * Math.PI, 100);
  }
  else if (e.keyCode == '38') {
    // up
    takeStepForward(scene.getObjectByName('cube'),
      0, 0.5 * Math.PI, 100);
  }
  else if (e.keyCode == '39') {
    // right
    takeStepRight(scene.getObjectByName('cube'),
      0, 0.5 * Math.PI, 100);
  }
   else if (e.keyCode == '40') {
    // down
    takeStepBackward(scene.getObjectByName('cube'),
      0, 0.5 * Math.PI, 100);
  }
}
```

First, we register the `checkKey` function as the listener for the `onkeydown` event. Now, whenever a key is pressed, the `checkKey` function will be called. In this function, we check `keyCode` of the event, and based on the value, we take a step in the appropriate direction. The final result looks like the screenshot we saw at the beginning of this chapter (`03.05-keyboard-controls.html`):



# Summary

In this chapter, we covered many different parts of Three.js to create the maze with a walking cube. We started by adapting a standard JavaScript library since Three.js didn't provide us with a geometry we could use out of the box. As you've seen, it is pretty easy to adapt these kinds of libraries and use them as input for creating Three.js geometries.

In this chapter, we've also spent a lot of time on rotation. By default, Three.js rotates objects around their center point. For many use cases, this is enough, but for more advanced animations or rotations, you need to approach this differently by moving its rotation point (also called pivot point). If you want to change its rotation point, you first need to translate a geometry and change this pivot point before applying the rotation.

Another aspect of rotation that you need to understand is the difference between the rotation property of a mesh, and applying rotations through a matrix transformation. When you set the rotation property of a mesh, you're not really changing the position of its vertices. You're just telling Three.js that it needs to apply this rotation before rendering the object. In this case, the position of the vertices of the geometry doesn't change. If you apply a matrix transformation directly to the geometry, you change the position of the individual vertices.

Another object of Three.js that we used in this chapter is the `THREE.Raycaster` object. We can use this object to select objects with our mouse or use it for some simple collision detection between objects.

The last big part of this chapter dealt with animations. There are many different ways to do animations in Three.js. You could update the variables yourself in the render loop, but it is often easier to use `Tween.js`, or another tween library such as `TweenLite`, for this purpose. With these kinds of libraries, you can easily change the value of a variable from start to end over a period of time using a specific easing method.

In the earlier chapters, we looked at geometries that consist of vertices and faces. In the next chapter, we'll look at particles. With particles we don't create 3D objects; instead, we work with particles created from the vertices of geometry. Each particle can be animated, and styled separately to create beautiful effects.

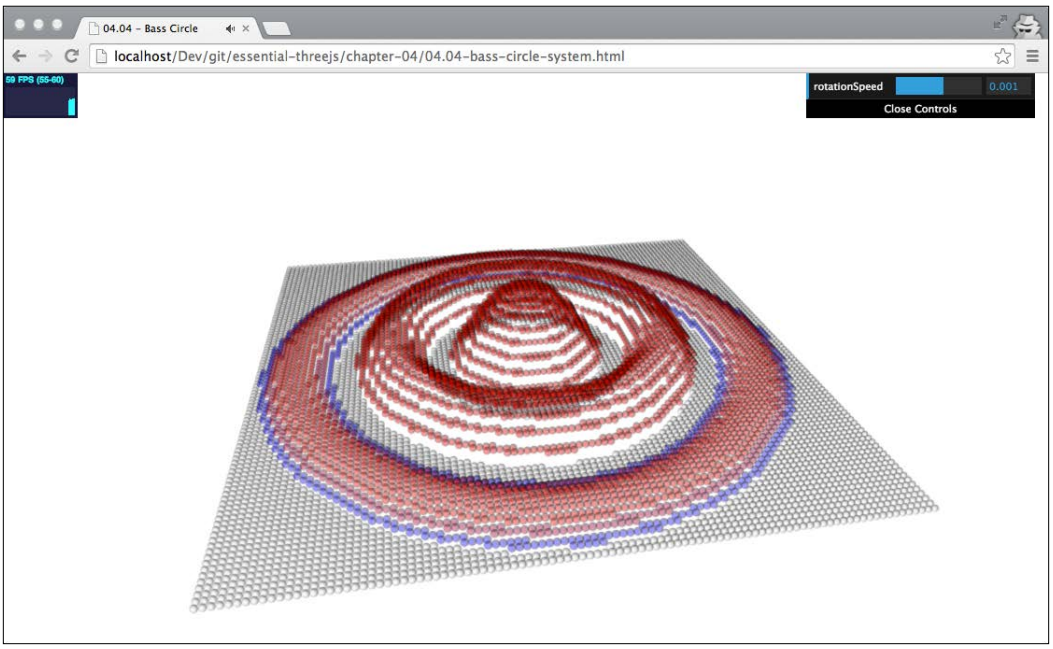
# 4
# Visualizing Audio Data with a Particle System

In the previous chapters, you learned to create scenes that consist of geometries, such as cubes and planes. In *Chapter 3*, *Navigate around a Randomly Generated Maze*, we even used cubes to create a complete maze. In Three.js, there is, however, a different type of object that you can add to a scene instead of the `THREE.Mesh` object we've used so far. It is the `THREE.ParticleSystem` object. With a particle system, the complete geometry isn't rendered; instead, only the vertices are shown. In this chapter, we'll explore how you can use a `THREE.ParticleSystem` object by showing various ways to visualize sound. We'll cover the following topics in this chapter:

- A `THREE.ParticleSystem` object can be created in two different ways. We will show you how to use an empty `THREE.Geometry` object to create a particle system and how to create one from an existing particle system.
- A static particle system isn't that interesting to look at. We will show you how to animate individual particles inside the particle system. We will discuss two approaches to do this: one where we scale the whole geometry and one where we move the individual vertices around.
- When you create a `THREE.ParticleSystem` object, you also provide a material to the system. This material defines what the individual particles look like. In this chapter, we'll explain the various properties of the `THREE.ParticleSystemMaterial`.
- With `THREE.ParticleSystemMaterial`, we can style the particles in the system; however, by default, all the particles have the same color. We will show you how to set the color of each individual particle.

- The last part of this chapter introduces some of the blend modes that are available within Three.js. A blend mode defines how the color of a pixel on the screen interacts with the color of the pixel behind it. For the `THREE.ParticleSystem` object, the blend mode `THREE.AdditiveBlending` is especially interesting.

The example we will use in this chapter to highlight the features of the `THREE.ParticleSystem` object deals with visualizing audio. The following screenshot (the `04.05-bass-circle-system.html` file) provides a preview of what we will create in this chapter:

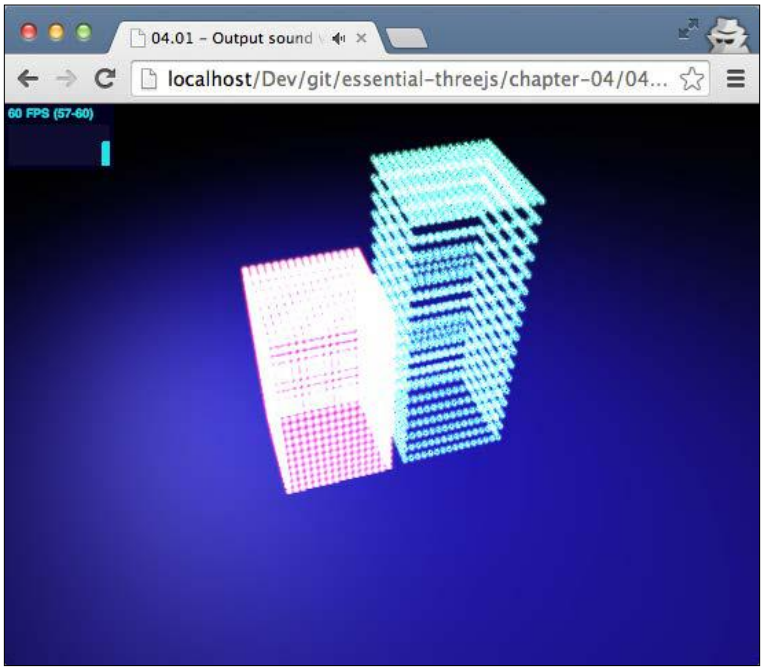



In this screenshot, we've got a particle system that shows the volume of the low, mid, and high ranges. In the middle of the screenshot, you can see the volume of the low range. The middle ring shows the volume of the mid range, and the outer ring shows the high notes.

We will start from the beginning, and the first thing we'll do is show you how to connect the HTML5 Web Audio API to Three.js. We will also show you a simple example of a particle system.

# Visualizing the audio volume

In the first example, we are going to create a 3D visualization of the volume of the left and right channels. In the following screenshot, you can see two cubes where the left cube represents the volume of the left channel and the right one shows the volume of the right channel (the `04.01-output-sound-volume.html` file):




As you can already see in the preceding screenshot, we don't show a complete rendered cube anymore. What you see are individual vertices that make up the cube. These individual vertices are what we call particles.

To create this example, we have to perform the following steps:

1. The first thing we need to do is set up HTML5 Web Audio (`http://www.w3.org/TR/webaudio/`) to be able to play an audio file that we can use as an input.
2. We'll use Web Audio's AnalyserNode interface to determine the volume from the left and right channels.

3. For this example, we need to create two different `THREE.ParticleSystems` objects: one to show the left volume and one to show the right volume.
4. Finally, when we play the audio file, we'll use the volume to animate the `THREE.ParticleSystems` object that we'll create.

The first thing we need to do is set up the HTML5 Web Audio part, which we will see in the next section.

# Setting up the HTML5 Web Audio API

We won't go too deep into how Web Audio works, but you will see just enough to understand how to use it. Web Audio works when you create an AudioContext interface through which you can create various nodes. For instance, you need the AudioBufferSourceNode interface to play a sound, the AudioDestinationNode interface to output a sound to a speaker, the AnalyserNode interface to perform Fast Fourier transformation, and many other nodes. You need to connect all these nodes together to get the effect you want. For the example in this section, we're going to use the following nodes:

- An AudioBufferSourceNode interface to play back an `.ogg` file
- A ChannelSplitterNode interface to split the sound into left and right channels
- Two AnalyserNode interfaces to determine the volume of the sound for each channel
- The default DestinationNode interface provided by the AudioContext interface to play the sound on speakers

The following JavaScript code shows how to accomplish this:

```
context = new AudioContext();

// create the nodes
sourceNode = context.createBufferSource();

splitter = context.createChannelSplitter();

analyser = context.createAnalyser();
analyser.smoothingTimeConstant = 0.4;
analyser.fftSize = 1024;
```

```
analyser2 = context.createAnalyser();
analyser2.smoothingTimeConstant = 0.4;
analyser2.fftSize = 1024;

// connect them together
sourceNode.connect(splitter);
splitter.connect(analyser, 0);
splitter.connect(analyser2, 1);
sourceNode.connect(context.destination);
```

Here, you can see that we've created an `AudioContext` function, which we will use to create other types of nodes and then use the `connect` function to connect these nodes with each other. When we connect the splitter, we use the additional argument to specify which channel (left or right) we want to connect to. With all the nodes connected, we're now ready to play back music.

We also configured the AnalyserNode interface here, which we have created using the `createAnalyser` function, with the `smoothingTimeConstant` and `fftSize` properties. The Web Audio specification explains what these properties do, as listed in the following table:

| Property | Description |
|---|---|
| `smoothingTimeConstant` | This is a value between 0 and 1, where 0 represents no time averaging with the last analysis's frame. The default value is 0.8. |
| `fftSize` | This refers to the size of the **Fast Fourier Transform** (**FFT**) used for frequency-domain analysis. |

We'll show you how to use these properties in our examples in the coming sections, but before we do that, let's first look at how to set up a `THREE.ParticleSystem` object.

# Creating a particle system

In the previous section, we split the sound into a left and right channel using the ChannelSplitterNode interface. In this section, we'll represent the volume of each channel using a `THREE.ParticleSystem` object. First, let's look at the code to create the first `THREE.ParticleSystem` object; we start by creating its geometry. Consider the following code:

```
var boxGeometry = new THREE.BoxGeometry(3, 6, 3, 15, 25, 15);
```

To create a particle system, we need a set of vertices. In this case, to create this set of vertices, we need to create a `THREE.BoxGeometry` object. As a normal box only consists of eight vertices, we use the last three arguments of the `THREE.BoxGeometry` constructor to specify how many segments we want for the width, depth, and height dimensions. By using these properties, we get many more vertices, which make the particle system look a lot better.

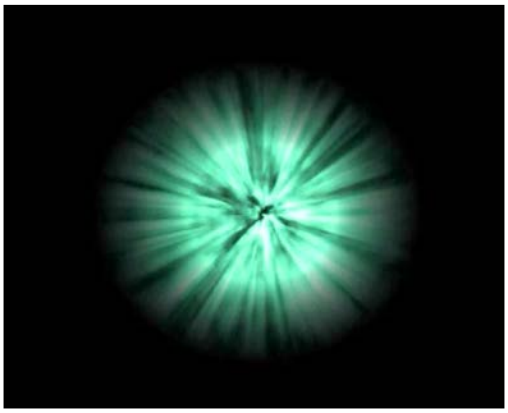
To style a particle system, we use `THREE.ParticleSystemMaterial`. Refer to the following code:

```
// setup the material
var pm = new THREE.ParticleSystemMaterial();
pm.map = THREE.ImageUtils
  .loadTexture("../assets/textures/particles/particle.png");
pm.blending = THREE.AdditiveBlending;
pm.transparent = true;
```

Because a particle system consists of individual particles, we can't use the standard materials provided by Three.js. So, we have to use the `THREE.ParticleSystemMaterial` that can only be used for particle systems and contains the properties described in the following table:

| Property | Description |
|---|---|
| `color` | If you don't use a texture (with the `map` property), this property determines the color of the particles. |
| `opacity` | This property refers to the opacity of each pixel (when the `transparent` property has been set to `true`). |
| `map` | This property refers to the texture to be used for each individual particle. |
| `size` | This property refers to the relative size of a particle; the default value is 1.0. |
| `blending` | This property defines how the particle color or the texture blends together with the color of the pixels that are behind it. |
| `vertexColors` | If this is set to `true`, you can change the color of each individual particle. |
| `transparent` | This property determines whether the particle is transparent or not. It affects the opacity and the way textures are blended. |

In this case, we load a simple texture, as shown in the following image:



We set the `transparent` property to `true` and tell Three.js to use the `THREE.AdditiveBlending` object. This blend mode adds the color of the texture to the color of the pixel that is behind it, resulting in the glowing effect that you see in this example. Finally, we can create the `THREE.ParticleSystem` object. Consider the following code:

```
// create the particle system
var ps = new THREE.ParticleSystem(BoxGeometry, pm);
ps.name = 'cube';
ps.sortParticles = true;
scene.add(ps);
```

To create a `THREE.ParticleSystem` object, we follow the same approach as we followed for a `THREE.Mesh` object. We provide a geometry (`BoxGeometry`) and a material (`pm`) and add it to the scene. Note that we set the `sortParticles` property to `true`. You should always do this whenever you change the vertices (or the position of the vertices) of the system to make sure that they are rendered correctly and don't result in weird overlapping artifacts.

To create the second particle, we just clone the initial material and change the texture we want to use. Consider the following code:

```
var pm2 = pm.clone();
pm2.map = THREE.ImageUtils
  .loadTexture("../assets/textures/particles/particle2.png");
```

Then, we create a `THREE.ParticleSystem` object in the same manner, as shown in the following code:

```
var ps2 = new THREE.ParticleSystem(BoxGeometry, pm2);
ps2.name = 'cube2';
scene.add(ps2);
```

We've also given both of the particle systems a specific name so that we can easily reference them when the need to animate them arises. Now, all we've got to do is play the sound and animate the particle system.

# Playing a sound and animating the particle system

Before we can play a sound, we must first load our sound file into the memory and set it as the buffer in the `sourceNode` object that we created previously. We use the following function to load the file using a standard `XMLHttpRequest` function:

```
function loadSound(url) {
    var request = new XMLHttpRequest();
    request.open('GET', url, true);
    request.responseType = 'arraybuffer';

    request.onload = function() {

        // decode the data
        context.decodeAudioData(request.response, function(buf) {
          playSound(buf);
        }, onError);
    }
    request.send();
}
```

When the file is loaded, the `onload` function is executed, which first decodes the audio. When this is done, it calls the `playSound` function, which starts playing the sound. This is described in the following code:

```
function playSound(buffer) {
    sourceNode.buffer = buffer;
    sourceNode.start(0);
}
```

Now, we finally get to the part where we can animate the cubes. For this, we use the `render` function that we've seen in all the examples so far. In this specific example, we call an `updateCubes` function from this `render` function. Consider the following code:

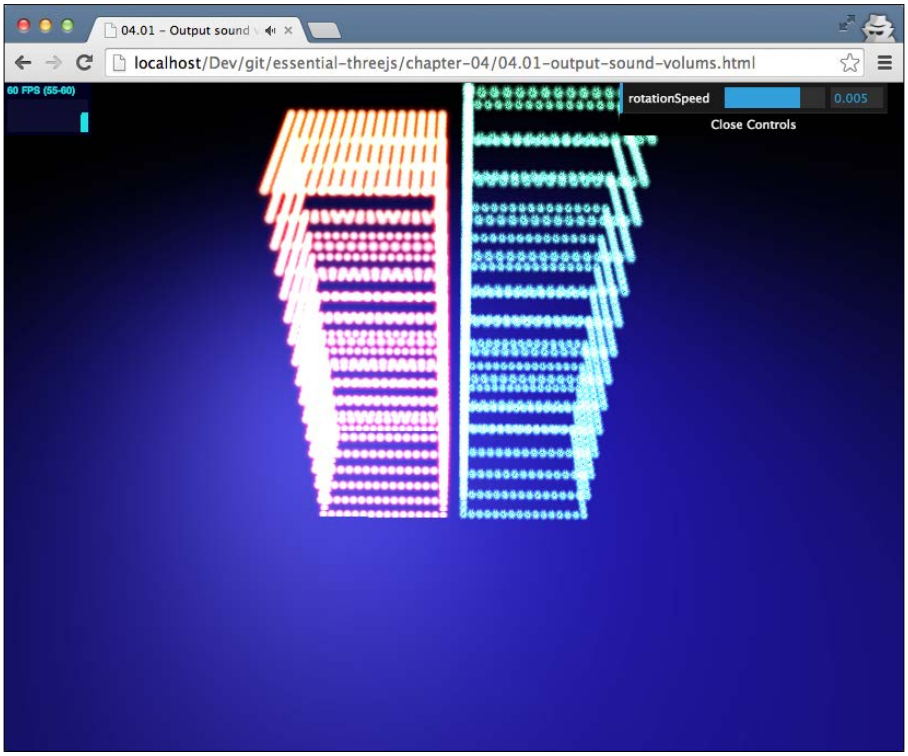
```
function render() {
...
   updateCubes();
...
}
```

In the `updateCubes` function, we use the information from the AnalyserNodes interface to determine the volume and then use that volume to scale the cubes appropriately. This is described in the following code:

```
function updateCubes() {
    // get the average for the first channel
    var array = new Uint8Array(analyser.frequencyBinCount);
    analyser.getByteFrequencyData(array);
    var average = getAverageVolume(array);

    // get the average for the second channel
    var array2 = new Uint8Array(analyser2.frequencyBinCount);
    analyser2.getByteFrequencyData(array2);
    var average2 = getAverageVolume(array2);

    // clear the current state
    if (scene.getObjectByName('cube')) {
        var cube = scene.getObjectByName('cube');
        var cube2 = scene.getObjectByName('cube2');
        cube.scale.y = average / 20;
        cube2.scale.y = average2 / 20;

    }
}
```

To scale the cube according to its volume, we use the `getByteFrequencyData` function from each AnalyserNode interface. This returns an array of the size `frequencyBinCount`, where each entry contains the volume for a range of frequencies. With the `getAverageVolume` function, we add all the entries in the array and divide them by the number of elements (implementation not shown). This value (called `average` and `average2`) is used to scale the two `THREE.ParticleSystem` objects along the *y* axis by setting the `scale.y` property.

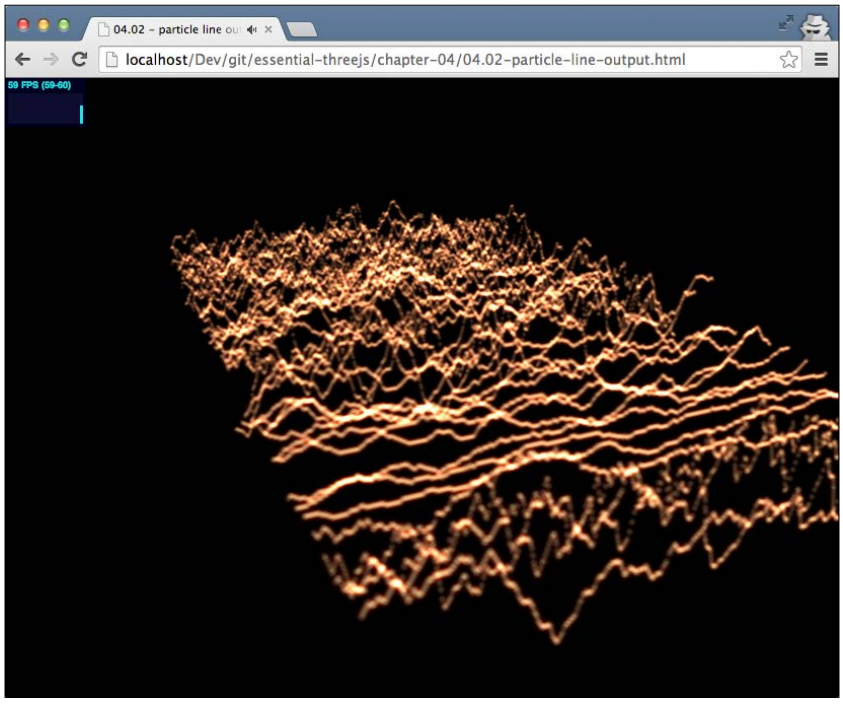
With the `smoothingTimeConstant` property, which we had set when we created the analyzers, we can now configure how smoothly the cubes can be animated. This is shown in the following screenshot:




In the next section, we won't create a `THREE.ParticleSystem` object using an existing geometry, but we'll create a geometry from scratch.

# Creating a particle system by hand

Now that we've seen our first particle system in action, let's look at how we can create a particle system from a custom geometry. For this, we're going to visualize the waveform from our audio as a number of particle systems. The result will look something like the following screenshot, where each line from left to right is a single particle system (the `04.02-particle-line-output.html` file):




To accomplish this sound visualization, we need to perform pretty much the same steps as we saw in the previous example. We need to configure our audio. Next, we need to access a NodeAnalyser interface to get the waveform; based on this information, in this example, we're going to create a new particle system.

# Web Audio's configuration and the render loop

In this example, we won't go into the details of how to set up the Web Audio part, as we've already seen that in the previous example. For this example, in the Web Audio part, we will configure the following:

- An AudioBufferSourceNode interface that we can use to play back an `.ogg` file
- One AnalyserNode interface to determine the waveform
- The default DestinationNode interface provided by the AudioContext interface to play the sound on the speakers

Just as we did in the previous example, we'll update the screen from the `render` function, as shown in the following code:

```
function render() {

    c++;
    if (c % 2 == 0) updateWaves();
    // update stats
    stats.update();

    // and render the scene
    renderer.render(scene, camera);

    // render using requestAnimationFrame
    requestAnimationFrame(render);
}
```

As you can see, we do this by calling the `updateWaves()` function. Note that we've added `(c % 2 == 0)`. We do this to limit the speed with which the waves move around the screen. So, for every two calls to the `render` function, we only update the waves once. Now, let's look at how we created the waves in this example in a bit more detail.

# Creating waves with a custom geometry

All the interesting stuff in this example happens in the `updateWaves()` function, so let's walk through the different parts of this function. Consider the following code:

```
function updateWaves() {

    // get the average for the first channel
    var array = new Uint8Array(analyser.frequencyBinCount);
    analyser.getByteTimeDomainData(array);
```

The first thing we need is the data from the Web Audio analyzer. In the previous example, we used the `analyser.getByteFrequencyData()` function to calculate the volume. In this example, we get the data from the `analyser.getByteTimeDomainData()` function. If we visualize this time domain, we get a waveform. Next, we set up the material. This is described in the following code:

```
// setup the material
    var pm = new THREE.ParticleBasicMaterial();
    pm.map = THREE.ImageUtils
      .loadTexture("../assets/textures/particles/particle.png");
    pm.blending = THREE.AdditiveBlending;
    pm.transparent = true;
    pm.opacity = 0.3;
    pm.size = 1.5;
```

This example uses pretty much the same properties that we saw in the previous example. We set a texture through the `map` property, set up the `THREE.AdditiveBlending` object for the glow effect, and changed the opacity and transparency of the material. The only thing we now need to create a particle system is a geometry. Consider the following code:

```
// create an empty geometry
    var geom = new THREE.Geometry();

// add the vertices to the geometry based on the waveform
    for (var i = 0; i < array.length; i++) {
        var v = new THREE.Vector3(1, array[i] / 8, (i / 15));
        geom.vertices.push(v);
    }
```

To create a geometry from scratch, we first instantiate a new `THREE.Geometry` object. A `THREE.Geometry` object contains an array where it stores all the vertices. This array can be accessed through the `vertices` property. What we do in the previous piece of code is create a new vertex in the form of a `THREE.Vector3` object. When you create this object, you need to specify an *x*, *y*, and *z* value. We just set the *x* value to `1` so that we can draw each wave by varying the *y* and *z* values. We use the *y* value to represent the value in the array (`array[i]`) and the *z* value to just move to the next position for the next value. This way, whenever this function is called, a single waveform is drawn.

Now that we've created the material and added a number of vertices to the geometry, we can combine these to create a `THREE.ParticleSystem` object. This is described in the following code:

```
// create a new particle system
    var ps = new THREE.ParticleSystem(geom, pm);
    ps.sortParticles = true;
```

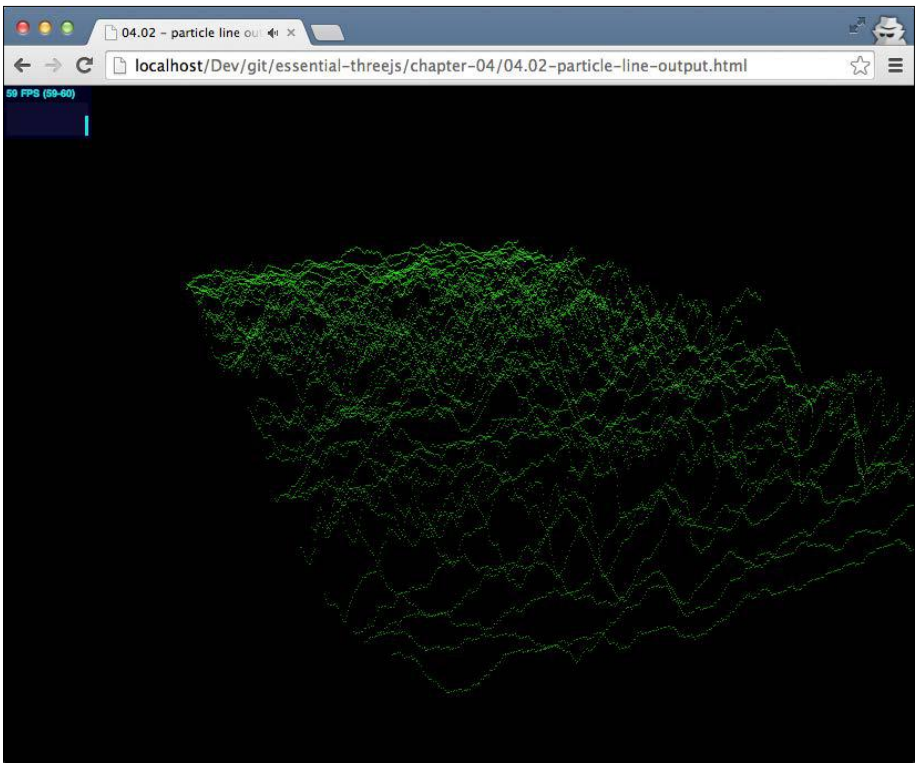
If you look at the screenshot at the beginning of this chapter, you can see that we move the older waves to the back. The following code moves the existing waveforms back and adds the newly created `THREE.ParticleSystem` object:

```
// move the existing particle systems back
    systems.forEach(function(e) {
        e.position.x -= 1.5
    });

// and remove the oldest particle system
    if (systems.length === 40) {
        var oldPs = systems.shift();
        if (oldPs) {
            scene.remove(oldPs);
        }
```

```
        }

    // add the new to the systems array and the scene
        systems.push(ps);
        scene.add(ps);
    }
```

Here, you can see that we iterate through the systems array, an array that we use to keep track of all the `THREE.ParticleSystem` objects we've created. For each particle system in this array, we change the position along the *x* axis. In this example, we limit the maximum number of waves to be shown at the same time to 40. If we don't do this, at a certain point, the animation will become unresponsive as Three.js has to keep track of all the particle systems that we create (around 30 every second). Finally, after we've removed the obsolete systems, we push the newly created ones to the systems array and add them to the scene. The result is a nice visualization of the waveform of the currently playing audio sprinting along the *x* axis.

By playing around with the `THREE.ParticleSystemMaterial` object, it is very easy to change the look and feel of the waveform. This is shown in the following screenshot:



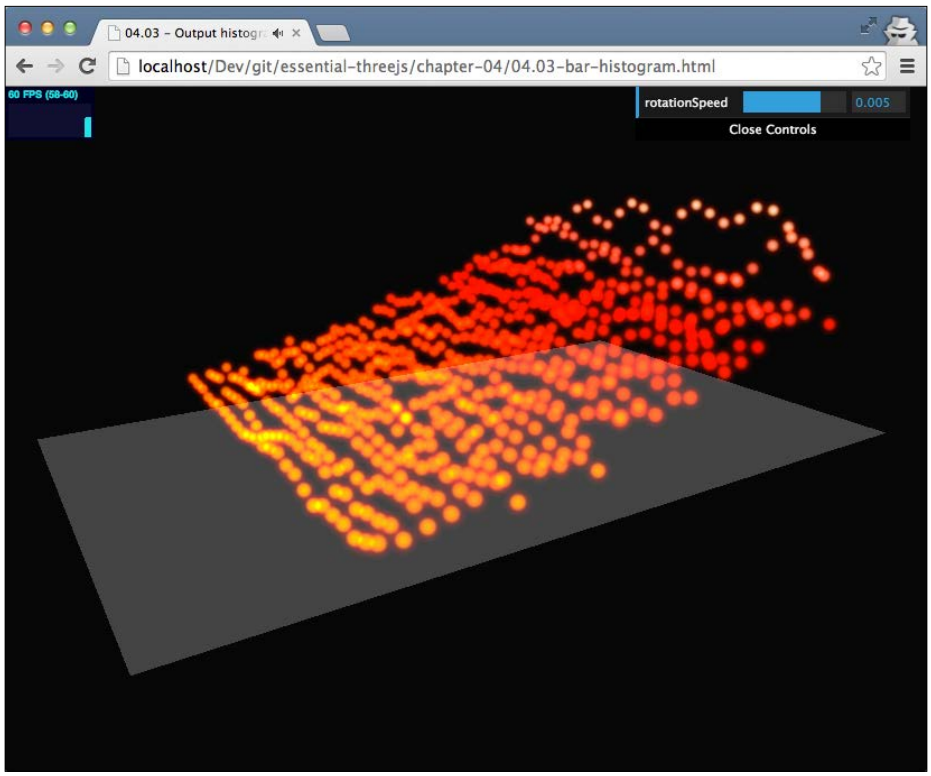
For instance, the waveform in the previous screenshot was created with the material described in the following code:

```
var pm = new THREE.ParticleSystemMaterial();
pm.color = new THREE.Color(0x00ff00);
pm.transparent = true;
pm.opacity = 0.5;
pm.size=0.1;
```

So far, we haven't looked at how to style individual particles. All the particles we've created use the same texture. It is however possible to change the colors of the individual particles. In the next section, we're going to explore how you can do this.

# Customizing colors of individual particles

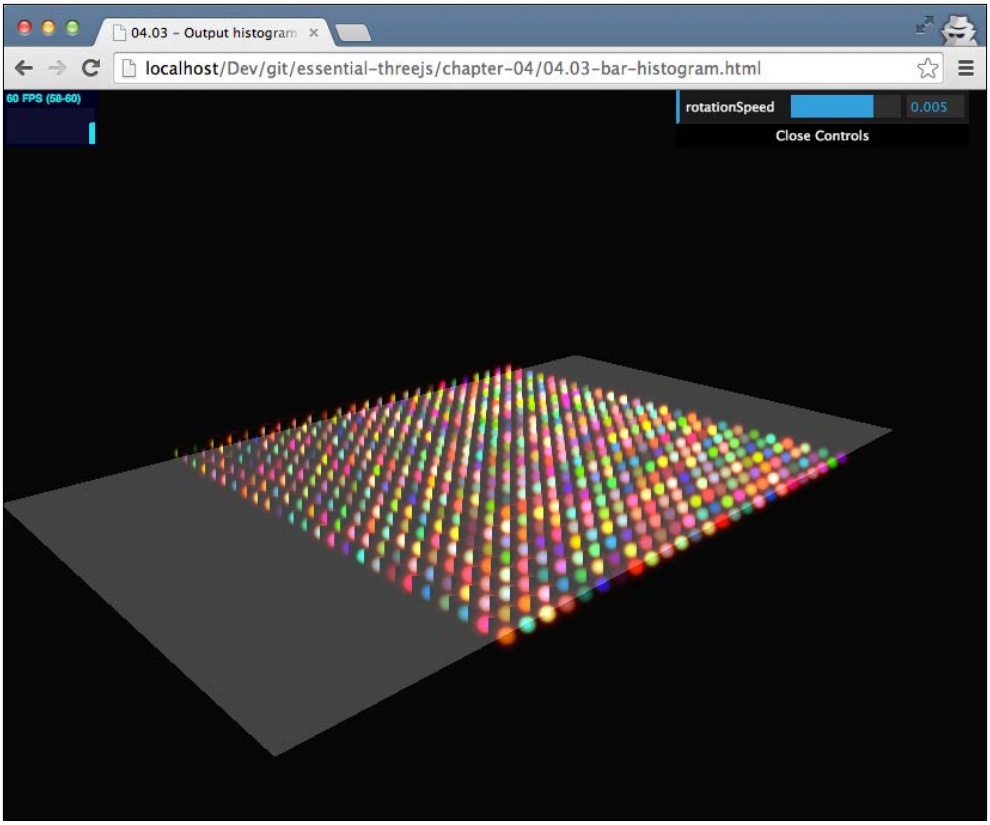
In this example, we're going to create a visualization that shows the amplitude of the individual frequencies of the currently playing audio. For each frequency, the height of a particle and its color will change to reflect the amplitude. The result looks something like the following screenshot:



In this screenshot, you can see the lower frequencies to the left and the higher frequencies to the right. As you can see, the height and color of the individual particles are different. In the following couple of sections, we'll see how to do this.

# Coloring individual particles

Let's start simple and create a `THREE.ParticleSystem` object without any animations. We'll create the particle system that you can see in the following screenshot:




Here, you can see a set of particles where each particle has its own color. To accomplish this, we have to perform the following two steps:

1. First, we have to configure the `THREE.ParticleSystemMaterial` object whose individual particles we want to color.
2. For each vertex in the geometry that we provide to the `THREE.ParticleSystem` object, we have to define a `THREE.Color` object.

Let's start by configuring the material; this is described in the following code:

```
var pm = new THREE.ParticleSystemMaterial();
pm.map = THREE.ImageUtils
  .loadTexture("../assets/textures/particles/particle.png");
pm.blending= THREE.AdditiveBlending;
pm.transparent = true;
pm.size=1.5;
pm.vertexColors = true;
```

Nothing new here, except the last line. With the `vertexColors` property of the `THREE.ParticleSystemMaterial` object, we can tell the particle system how we want the color of the particles to behave. If this property is set to `false`, the color of the particles will use the color set by the `color` property of the material. If this property is set to `true`, as is done in this case, we can set the color of each individual vertex.

To color each individual particle, we need to specify a `THREE.Color` object for each of the particles and add that object to the `colors` property, which is an array of the `THREE.Geometry` object. The following piece of code shows how we did this, the result of which was shown in the previous screenshot:

```
function setupParticleSystem(width, depth) {
  var targetGeometry = new THREE.Geometry();
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < depth; j++) {
      var v = new THREE.Vector3(i / 2 - (width / 2) / 2
        , 0, j / 2 - (depth / 2) / 2);
      targetGeometry.vertices.push(v);
      targetGeometry.colors.push(
              new THREE.Color(Math.random() * 0xffffff));
    }
  }

  var ps = new THREE.ParticleSystem(targetGeometry, pm);
  ps.name = 'ps';
  scene.add(ps);
}
```

In this function, we create a new `THREE.Geometry` object and fill it with a set of vertices. For each vertex that we push into the vertices array of the `THREE.Geometry` object, we also create a random `THREE.Color` object that we push into the colors array of the same `THREE.Geometry` object.

The result is that Three.js will use the `THREE.Color` objects in the colors array to render the individual particles. We'll use the same approach in the next section to create colors based on the amplitude of a specific frequency.

# Coloring the particles based on the amplitude

As you can see in the screenshot at the beginning of the *Customizing colors of individual particles* section, we color the vertices starting from orange at the bottom, then moving on to red in the center, and finally ending with white at the top. We could create this color scale manually, but that takes a lot of work and requires a good understanding of how colors work. Luckily, there is a very good open source library that specifically solves this problem. This library is called Chroma.js and can be downloaded from GitHub at `https://github.com/gka/chroma.js`.

With Chroma.js, you can define a color scale that you can use to automatically get an appropriate color, as shown in the following code:

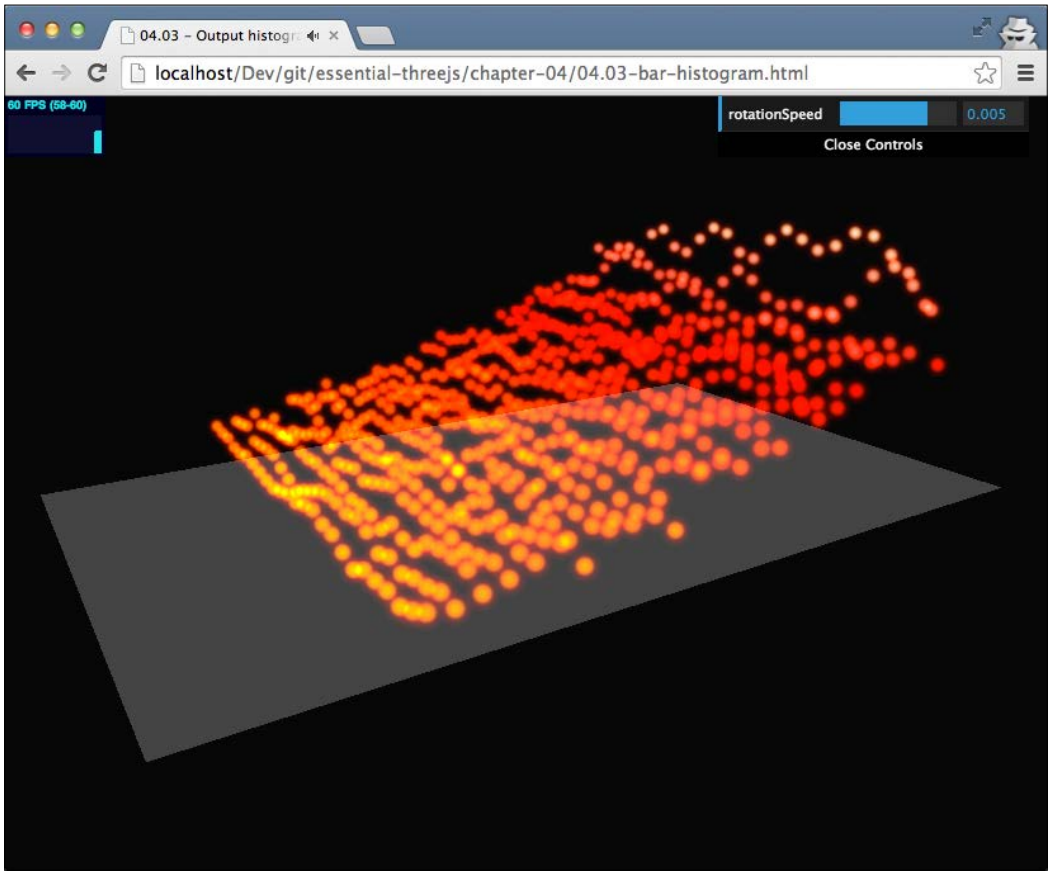
```
var scale = chroma.scale(['orange', 'red', 'white'])
  .domain([0, 255]);
```

In this example, we will create a color scale with three colors. Now, when we use scale 0, we'll get the orange color; with scale 255, we'll get the white color; and with scale 64, we will get something between orange and red colors. The range can be pretty much anything you want. We've used a scale from 0 to 255 because that is the range we receive from the analyzer.

Now, let's look at how we can use this `scale` object together with the information from the `frequencyBinCount` function to color the individual vertices. Consider the following code:

```
var array = new Uint8Array(analyser.frequencyBinCount);
analyser.getByteFrequencyData(array);

var ps = scene.getObjectByName('ps');
var geom = ps.geometry;

for (var i = 0; i < array.length; i++) {
  if (geom.vertices[i]) {
    geom.vertices[i].y = array[i] / 40;
    if (array[i] > max) max = array[i];
    geom.colors[i] = new THREE.Color(scale(array[i]).hex());
  }

}
ps.sortParticles = true;
geom.verticesNeedUpdate = true;
```

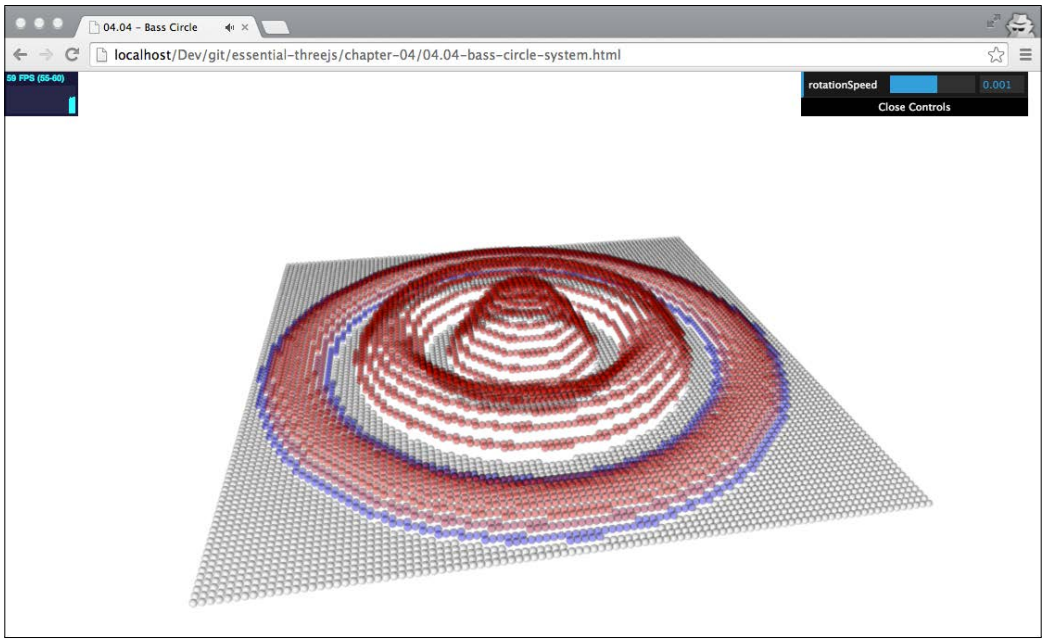
In this piece of code, we once again walk through all the values returned from the `getByteFrequencyData` function. For each of these values, we set the `y` value of the corresponding vertices (`geom.vertices[i]`) and use the `new THREE.Color(scale(array[i].hex())` function to set the color in the `geom.colors[i]` array. The result of this is a particle system whose colors respond to the amplitude of the various frequencies of the audio. The result is shown in the following screenshot:




In the last part of this chapter, we'll look at another way in which you can visualize audio data. We won't go into too much detail, as most topics have already been discussed in the previous sections.

# Combining dynamic colors to create advanced visualizations

In this section, we'll have a very quick look at how you can apply the concepts that you've learned in this chapter to create a more complex visualization. We won't show you all the JavaScript code for this but will just highlight the most important steps. If you want to know the details, you can look at the provided source code. The resulting advanced visualization is shown in the following screenshot:




The visualization you can see in the previous screenshot shows the low, mid, and high tones of the audio that is being played in the form of concentric circles. The low tones are shown in the center of the visualization, the second circle shows the mid tones, and the outermost circle shows the amplitude of the high tones. To create this visualization, we have to perform the following steps:

1. We have to start by setting up the initial particle system.
2. Next, we have to calculate the volumes for the low, mid, and high tones.

3. Then, we need to determine the particles that need to be updated for each tone range.
4. Finally, we need to set the height and color of the individual particles for each tone range.

# Setting up the initial particle system

The particle system that we use for this example is the same one we saw in the previous example. In that section, we saw the `setupParticleSystem()` function that created a simple particle system. In this example, we will use the `setupParticleSystem(100, 100)` function, which creates a particle system with 10,000 particles. The material we will use is described in the following code:

```
var pm = new THREE.ParticleSystemMaterial();
pm.map = THREE.ImageUtils
  .loadTexture("../assets/textures/particles/ball.png");
pm.transparent = true;
pm.opacity = 0.4;
pm.size=0.9;
pm.vertexColors = THREE.VertexColors;
```

With the particle system created, the next step is to calculate the average volumes for each frequency range.

# Calculating volumes for each range

In the previous examples, we've already calculated the average volume.
The main change here is that we want to calculate the volume for a specific range. Consider the following code:

```
var array =  new Uint8Array(analyser.frequencyBinCount);
analyser.getByteFrequencyData(array);

var lowValue = getAverageVolume(array,0,300);
var midValue = getAverageVolume(array,301,600);
var highValue = getAverageVolume(array,601,1000);
```

Here, we've specified the low range as the first 300 frequencies, the mid range as the second 300 frequencies, and the high range as the next 400 frequencies. The function to calculate these volumes is pretty simple. It is described in the following code:

```
function getAverageVolume(array, start, end) {
  var values = 0;
  var average;

  var length = end - start;
  for (var i = start; i < end; i++) {
    values += array[i];
  }

  average = values / length;
  return average;
}
```

This function just adds up all the individual volumes and divides them by the number of values. All we need to do now is select the particles that we want to update and set them to their correct height and color.

# Determining particles that need to be updated and setting the height and color of an individual particle

To determine which particles we need to update is a bit complex. We need to determine the offset from the center, the width of the ring, the spacing used between individual particles, and the size of the particles. To explain the details of this is a bit out of scope. If you want to see the implementation details, look at the source code for the `04.05-bass-circle-system.html` file.

The following lines of code show the steps needed to color and position the particles for the middle tones. First, we select all the particles that should be updated when the volume of the mid range changes.

```
var midParticles = [];
for (var i = 0 ; i < midRings ; i++) {
  midParticles.push(getFallOffParticles
    (centerParticle,(i+1+midFrom)*spacing,(i+midFrom)*spacing));
}
```

The `getFallOffParticles` function determines which particles should be updated based on the provided arguments. In this case, it is the set of particles that represent the middle tones. Besides the particles, we also need to determine the maximum height of the individual particles. The ones in the center of the ring are the highest. The further away from the center of the ring a particle is, the lower its height is. This is described in the following code:

```
for (var i = midRings/2 ; i < midRings ; i++) {
  midOffsets.push(
    Math.sin(Math.PI*(0.5*(i/(midRings/2)))));
}
```

Next, we need to change each particle in the ring to its correct height and color. For this, we've also created a helper function called `renderRing`, which is described in the following code:

```
// render the mid ring
for (var i = 0 ; i < midRings ; i++) {
  renderRing(
    geom,midParticles[i],
    midValue,midOffsets[i],
    midVolumeDownScale);
}

function renderRing(geom, particles
  , value, distanceOffset, volumeDownScale) {

  for (var i = 0; i < particles.length; i++) {
    if (geom.vertices[i]) {
      geom.vertices[particles[i]].y
        = distanceOffset*value/volumeDownScale;
      geom.colors[particles[i]]
        = new THREE.Color(scale(distanceOffset*value).hex());
    }
  }
}
```

In this function, you can see that we've used the same approach as we did in the example in the *Customizing colors of individual particles* section. We use the Chroma.js library to determine the color and position of the pixel based on the volume together with the offset that we calculated earlier. That's it for this last visualization example of this chapter.

# Summary

Particle systems provide a very interesting way to create 3D visualizations. In this chapter, we focused on visualizing audio, but you could use this for many other kinds of 3D visualizations. You can create a `THREE.ParticleSystem` object from any existing geometry. Three.js will then render a particle for each of the vertices of the geometry and use the provided `THREE.ParticleSystemMaterial` object to determine how each particle looks. Besides generating systems from existing geometries, you can also create a custom geometry by instantiating a `THREE.Geometry` object, adding `THREE.Vector3` objects to the vertices array and using that as an input for a `THREE.ParticleSystem` object.

If you use a `THREE.ParticleSystemMaterial` object, the standard way to render particles will be as simple squares using the `color` property from the supplied material. If you want more advanced shapes, you can use a texture for a particle by setting the `map` property of the the `THREE.ParticleSystemMaterial` object.

It's also possible to color each particle individually. For this, you'll have to fill the color array of the geometry with a `THREE.Color` object for each vertex, and you also need to set the `vertexColors` property of the `THREE.ParticleSystemMaterial` object to `true`. The last important property of the `THREE.ParticleSystemMaterial` object is the blending property. You can set this property to the `THREE.AdditiveBlending` object to get a nice-looking glow effect.

Finally, if you need to use color scales for your particles, the easiest way to do this is by using the Chroma.js JavaScript library.

In the next chapter, we'll dive deeper into creating custom geometries in Three.js, using algorithms to create trees, cities, and landscapes.

# 5
# Programmatic Geometries

In this chapter, we're going to look at a different approaches to creating geometries. Instead of using the built-in geometries from Three.js as we did in earlier chapters, we will create the geometries programmatically. This will allow us to create many different-looking types of objects without having to specify the models in detail or model them in a 3D modeling program. We'll cover the following topics in this chapter:

- Creating a geometry from scratch using vertices and faces.
- Coloring the generated geometries; we'll use the `color` property of the created faces.
- Working with textures; we need to configure an additional property besides the faces and the vertices. This is called **UV mapping**.
- Using Perlin noise, you can create natural-looking randomness. We'll use Perlin noise to create a natural-looking terrain.
- Generating textures using random libraries can be done just as geometries using random libraries are created. In this chapter, we will show you how to generate a random building-like texture.
- Using the algorithms of external libraries , we can generate the information needed to create a geometry. We'll use a library that contains an algorithm to generate trees.

We'll start with creating a 3D terrain using `Math.random()`.

# Creating a 3D terrain from scratch

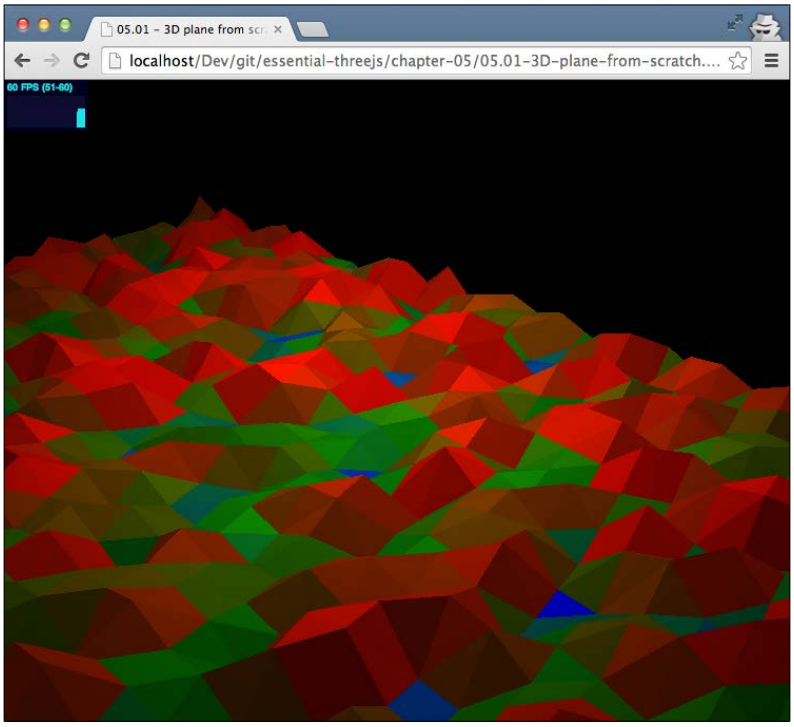
In the first section, we're going to create a 3D terrain from scratch. For this, we need to take the following steps:

- The first thing we need to do is define the position of all vertices that make up the geometry
- When all the vertices have been created, we need to connect them together to create faces
- From this geometry and these faces, we can create a `THREE.Mesh` object to add to the scene with a specific material

In this section, we're going to create a terrain using two different approaches. We're going to start with `Math.random()` to create the terrain, and after that, we're going to use a Perlin noise generator for a more evenly distributed terrain.

# Generating a terrain with Math.random()

For our first example, we're going to create the example you can see in the following screenshot (`05.01-3D-plane-from-scratch.html`):



In this screenshot, you can see a randomly generated terrain where the color of each face is determined by its height. Let's walk through the code to show you how to accomplish this. To make it easier to use, we've put all the relevant code into a single function:

```
function create3DTerrain(width, depth, spacingX, spacingY, height) {
   // create individual vertices
   ...
   // create the faces
   ...
   // create the mesh and add to scene
   ...
}
```

Before we look at the implementation of this function, let's quickly review the arguments it takes:

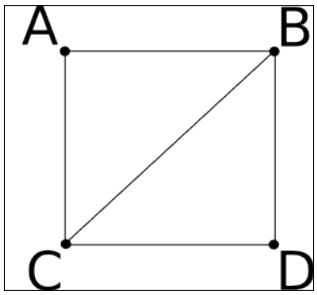
| Argument | Description |
|---|---|
| `width` | This argument refers to the width of the terrain to generate. It defines how many vertices will be created for the width. |
| `depth` | This argument refers to the depth of the terrain to generate. It defines the number of vertices that will be created for the depth. |
| `spacingX` | This argument refers to how many rooms there will be between the individual vertices along the *x* axis. |
| `spacingY` | This argument refers to how many space there will be between the individual vertices along the *z* axis. |
| `height` | This argument defines the maximum height of the terrain to generate. |

Let's start at the beginning and create all the vertices based on the supplied arguments:

```
var geometry = new THREE.Geometry();
for (var z = 0; z < depth; z++) {
  for (var x = 0; x < width; x++) {
    var vertex = new THREE.Vector3(
               x * spacingX,
               Math.random() * height,
               z * spacingZ);
    geometry.vertices.push(vertex);
  }
}
```

What we do here is first create an empty `THREE.Geometry` object, just as we did when we created the `THREE.ParticleSystem` object in the previous chapter. Based on the supplied depth and width, we determine the number and position of the particles we need to create. For the height, which is the `y` property of the `THREE.Vector3`, we use `Math.Random()` to generate a random value, and use that value to calculate the height of this specific vertex. This height is based on the supplied `height` property. The result of this part of the function is that we've got a `THREE.Geometry` object with an array of vertices.

The next step is to use the array position of these vertices to create faces. Three.js uses triangle faces that can be created by instantiating the `THREE.Face3` object. We're going to create faces in the format shown in the following figure:




As you can see in the figure, we are going to create squares by combining two faces. So we need to create a face using the **A**, **B**, and **C** vertices, and another face using the **B**, **C**, and **D** faces. We do this in the following code fragment:

```
for (var z = 0; z < depth - 1; z++) {
  for (var x = 0; x < width - 1; x++) {
    // we need to point to the position in the array
    // a - - b
    // |  /  |
    // c - - d
    var a = x + z * width;
    var b = (x + 1) + (z * width);
    var c = x + ((z + 1) * width);
    var d = (x + 1) + ((z + 1) * width);

    var face1 = new THREE.Face3(b, a, c);
    var face2 = new THREE.Face3(c, d, b);
    geometry.faces.push(face1);
    geometry.faces.push(face2);
  }
}
```

In this small fragment, we walk through all the vertices; based on their position, we have set the `a`, `b`, `c`, and `d` values to a specific value, which points to the correct vertex in the vertices array we filled in earlier. Based on these positions, we create two faces and add them to the `faces` array of `geometry`.

At this point, we've created the two most important parts of our geometry: the vertices and the faces. Before we create a `THREE.Mesh` object and add the geometry to the scene, we need to take a couple of small steps to recreate the example you saw at the beginning of this section. Let's start by looking at how to create the colors. In our example, each face has its own color based on the height of the face. We can specify this by setting the `color` property of the faces we created in the previous code fragment:

```
face1.color = new
      THREE.Color(scale(getHighPoint(geometry, face1)).hex());
face2.color = new
      THREE.Color(scale(getHighPoint(geometry, face2)).hex())
```

To set the color, we once again use `Chroma.js` to help us get the correct color value. The scale we used this time is as follows:

```
var scale = chroma.scale(['blue','green','red'])
              .domain([0,MAX_HEIGHT]);
```

So, when we call `scale()` and supply the height of the face, we get a color back from this range. To calculate the value to supply to the `scale` function, we use the `getHighPoint` function, which determines the highest position of the vertices used in a face:

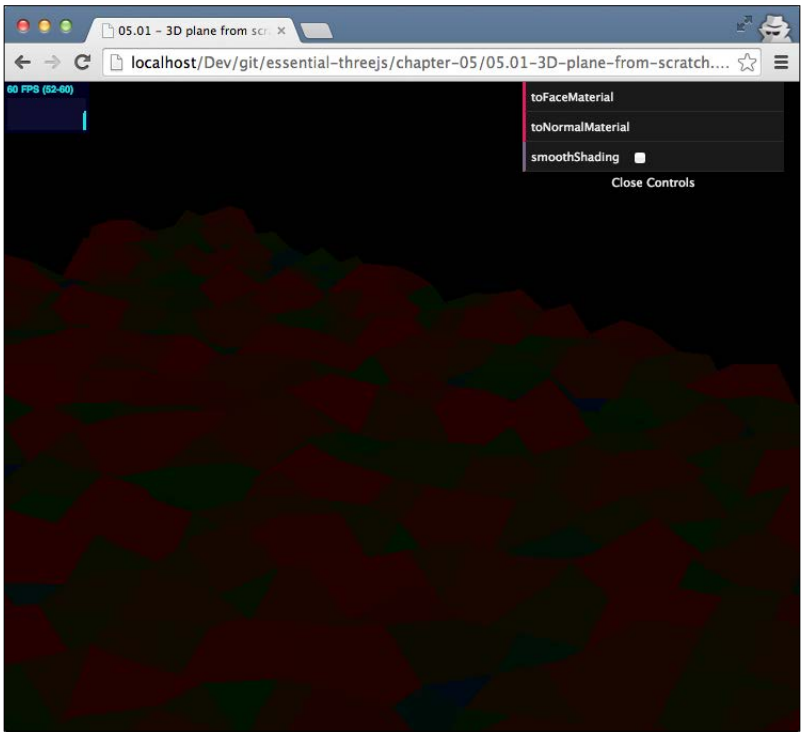
```
function getHighPoint(geometry, face) {

  var v1 = geometry.vertices[face.a].y;
  var v2 = geometry.vertices[face.b].y;
  var v3 = geometry.vertices[face.c].y;

  return Math.max(v1, v2, v3);
}
```

Now, let's create the material, create a `THREE.Mesh` object, and look at the result:

```
var mat = new THREE.MeshPhongMaterial();
mat.vertexColors = THREE.FaceColors;

// create the mesh
```

```
var groundMesh = new THREE.Mesh(geometry, mat);
groundMesh.translateX(-width / 1.5);
groundMesh.translateZ(-depth / 4);
groundMesh.name = 'terrain';

scene.add(groundMesh);
```

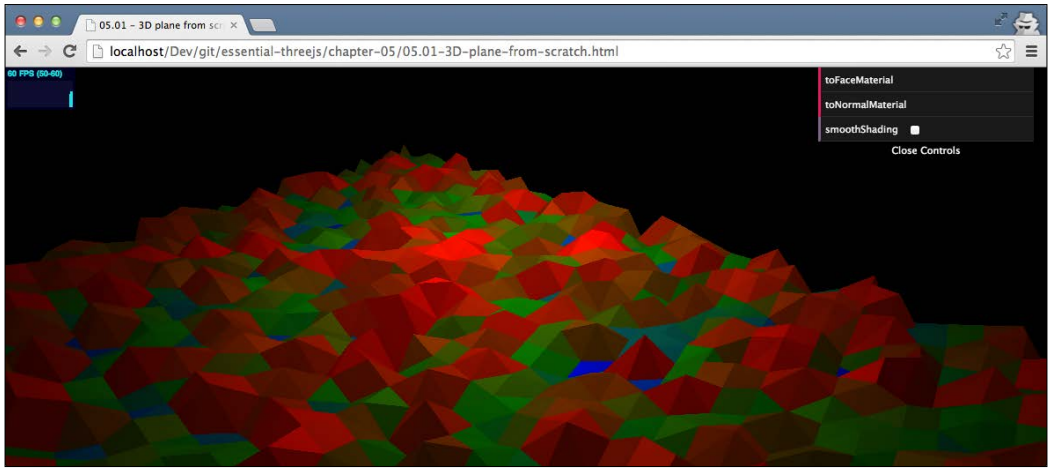
Here, we use `THREE.MeshPhongMaterial`, which can be used to create shiny objects and set the `vertexColors` property to `THREE.FaceColors` to tell the material to use the colors we specified on the faces. Now, we're ready to create the `THREE.Mesh` object and add it to the scene. The result, however, doesn't really look that great.



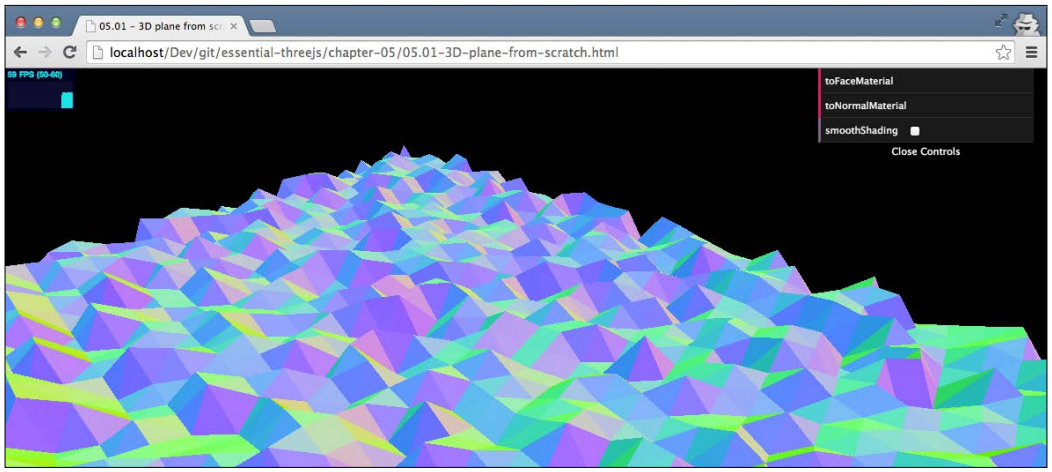

The shapes seem to be there, but the lighting and shading is completely off. The reason for this is that Three.js is still missing one piece of information. What we need to specify is the face normal vector. This is the vector that is perpendicular to the face. This is used by Three.js to determine how lighting applies to this specific face. We could calculate this by hand, but luckily Three.js provides an easier approach for this. Have a look at the following line of code:

```
geometry.computeFaceNormals();
```

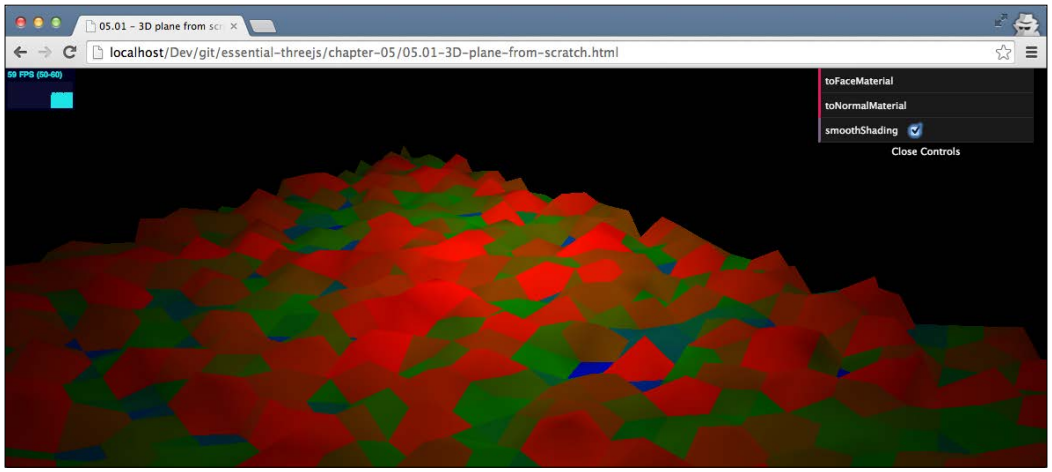
This will compute the normal vectors for all the faces of the geometry. Now, when we render the scene, we get the result we saw at the beginning of the chapter. Have a look at the following screenshot:




If you open the example (`05.01-3D-plane-from-scratch.html`) in your browser, you can find some additional functionality in the menu on the top-right corner. If you click on the **toNormalMaterial** button, the material will change to `THREE.MeshNormalMaterial`. With this material, the color of each face is based on its face normal vector, as shown in the following screenshot:



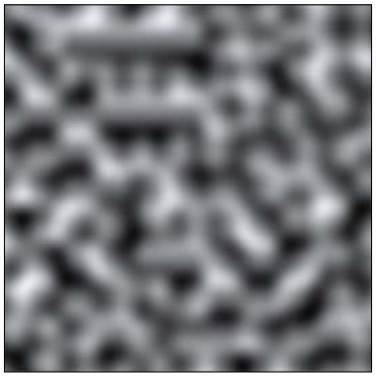
In this example, you can also enable `smoothShading`. With `smoothShading`, you don't see the individual faces, but Three.js combines them to create a smooth geometry. You can easily enable this by setting the `shading` property of the material to `THREE.SmoothShading`. The result looks like the following screenshot:



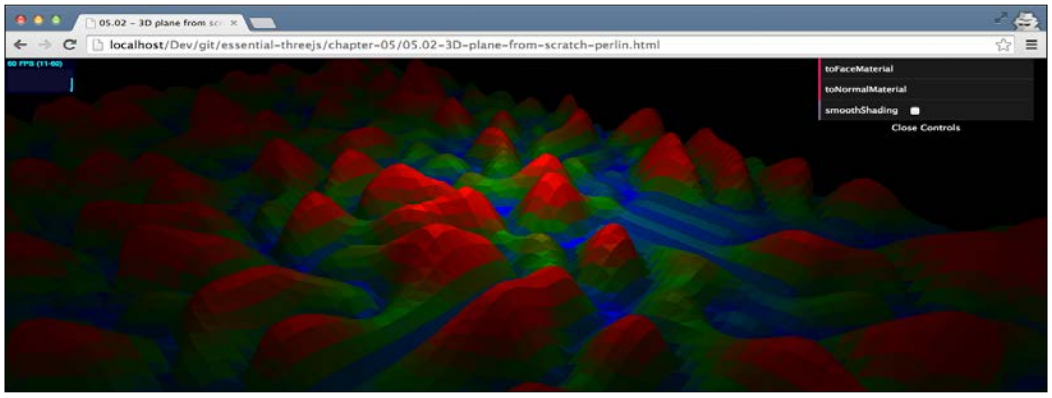

So far, we've created the terrain using the `Math.random()` function. This provides an interesting terrain, but as you can see, it doesn't really look natural. In the next section, we'll show you how you can use a Perlin noise generator for a more natural but random result.

# Generating a terrain with a Perlin noise

A Perlin noise is an algorithm that creates a more natural-looking distribution of values. For instance, the following screenshot is an example of a texture generated using a Perlin noise algorithm:



What you can see here is that even though it appears rather random, the transitions are very smooth and natural looking. If we apply this to our example, we'll get a result (`05.02-3D-plane-from-scratch-perlin.html`) that looks like the following screenshot:




As you can see in the preceding screenshot, the terrain generated is very smooth and looks very natural. To accomplish this, we only have to change the method `create3DTerrain` at a couple of small points and import an additional JavaScript library.

There are a couple of JavaScript Perlin noise libraries available, but I usually use the `perlin.js` library, which you can get from GitHub at the following link:

`https://github.com/josephg/noisejs`

First, include this library at the top of the JavaScript file:

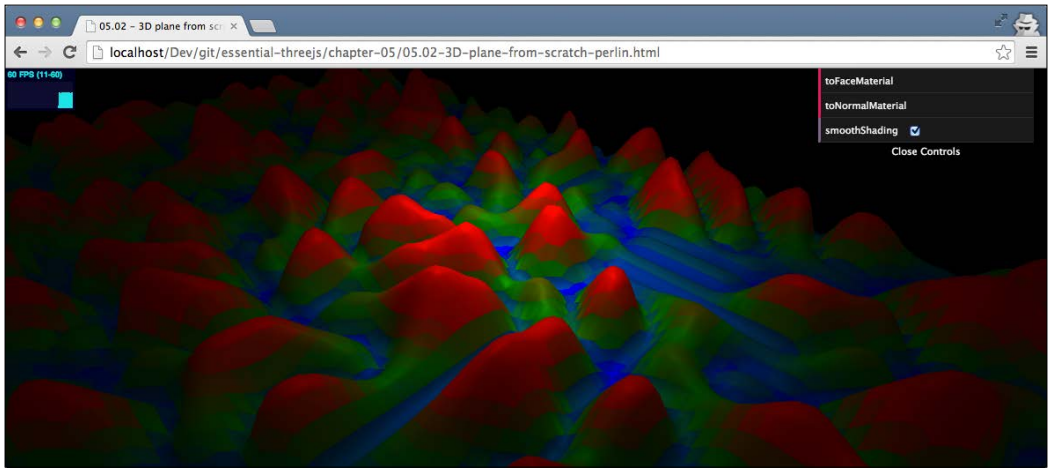
```
<script src="../libs/perlin.js"></script>
```

Now, change the first part of the `create3DTerrain` function to the following code snippet:

```
function create3DTerrain(width, depth, spacingX, spacingZ, height) {
   // seed the perlin generator
  var date = new Date();
  noise.seed(date.getMilliseconds());

  // first create all the individual vertices
  var geometry = new THREE.Geometry();
  for (var z = 0; z < depth; z++) {
      for (var x = 0; x < width; x++) {
```

```
                var yValue = Math.abs(
                      noise.perlin2(x / 10, z / 10) * height * 2);
                      var vertex = new THREE.Vector3(
                                  x * spacingX,
                                  yValue,
                                  z * spacingZ);
                           geometry.vertices.push(vertex);
                     }
               }
          ...
     }
```

Before we can use the `perlin` function, we first need to seed it. In other words, we need to supply the input value that is used to generate random values. In this case, we seed it with the current date in milliseconds. This way, each time we reload the scene, it will show a different landscape. Now we can call the `noise.perlin2()` function to generate the next height value. The arguments of this function define how smoothly the values returned by the `perlin2` function follow each other. If, in each step, the increase is too small, you'll see very little hills and values; if it is too large, you'll only see spikes. Finally, we use `Math.abs` to make sure the returned value is positive, since the `perlin2` function returns values from `-1` to `1`. Especially if you enable `smoothShading`, you can get some beautiful looking terrains, as shown in the following screenshot:



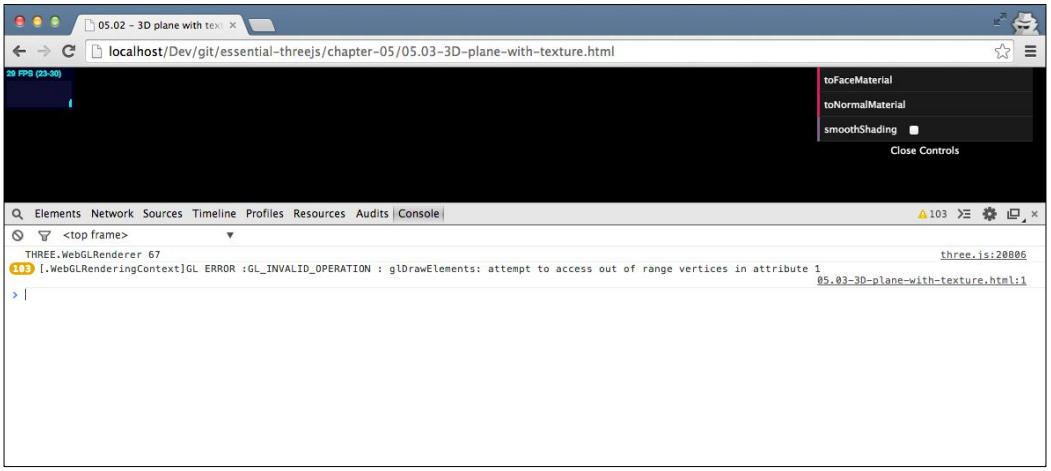

There is one aspect that we haven't tested yet with our custom geometry, and that is textures.

# Adding a texture

We've already seen in the earlier chapters how to configure a texture. So let's do that and see what the result is:

```
var mat = new THREE.MeshPhongMaterial();
mat.map = THREE.ImageUtils
          .loadTexture("../assets/textures/wood_1-1024x1024.png");
```

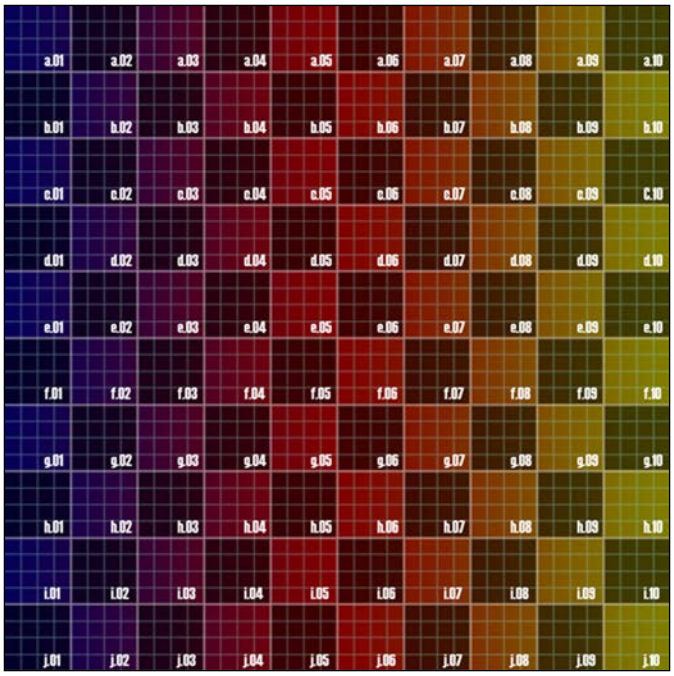
The result is not a texture, but a rather cryptic error message.



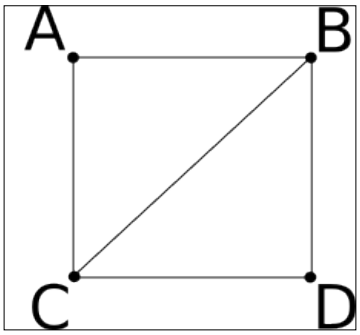

The reason, which isn't actually clear from this error message, is that we're missing one more piece of configuration in the geometry we've created: the UV map.

With a UV map, you define faces of the geometry map with respect to the provided texture. For each vertex of each face, we need to define its position on the texture. The texture, which is practical to use when debugging UV maps, shows the range of the U value (the *x* axis) and the V value (the *y* axis).

As you can see in the following screenshot, both run from **0** to **1**, starting at the top-left corner:




When we discussed faces in this chapter, we looked at the following figure:



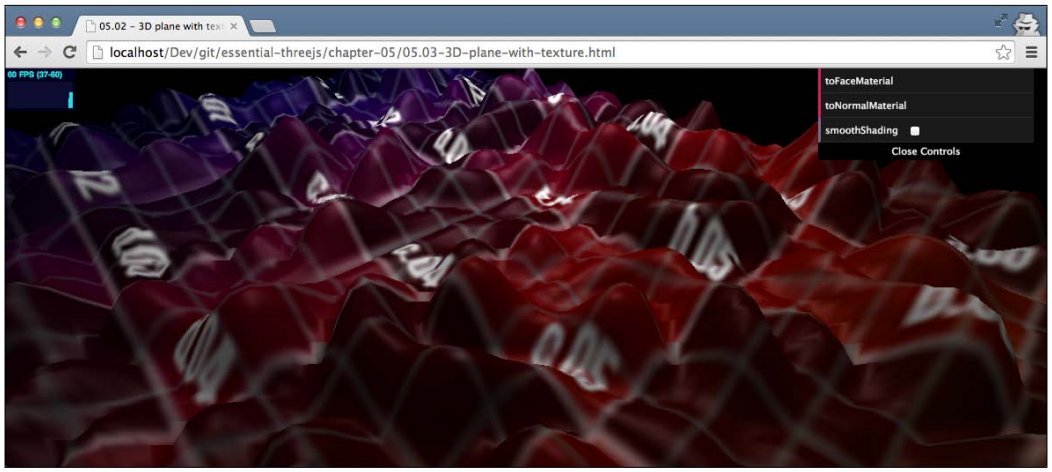

Now let's assume that we want to use this texture on these two faces, and we want to use only its top half. This would mean that vertex **A** would get an UV value of [0,0], vertex **B** would get [1,0], vertex **C** would get [0,0.5], and **D** would get [1,0.5]. Not so difficult, but something we need to add to our geometry if we want to be able to use textures.

We're going to extend the `create3DTerrain` function again to also create the UV mapping we need. To specify these UV mappings, Three.js uses a `THREE.Vector2` object; this object is a 2D vector, so it takes two arguments. This is analogous to the `THREE.Vector3` object we used to specify a point in a 3D space:

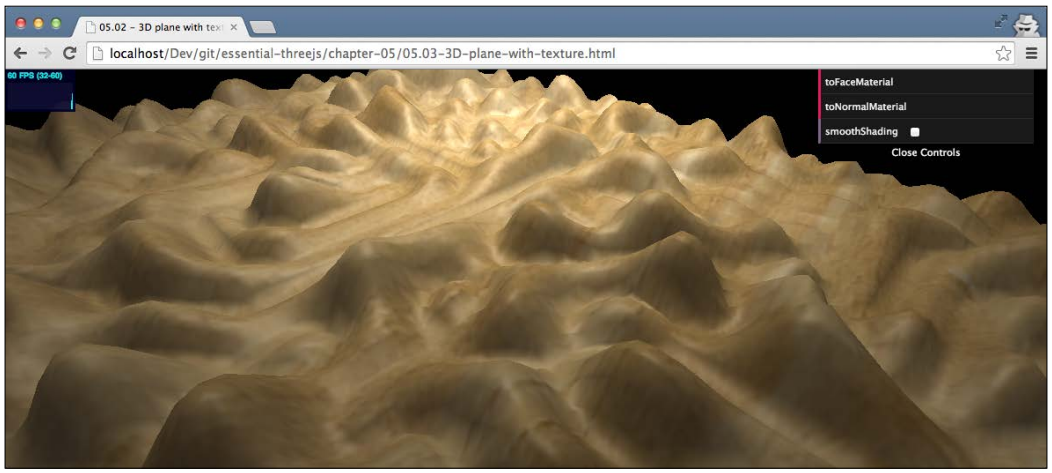
```
for (var z = 0; z < depth - 1; z++) {
    for (var x = 0; x < width - 1; x++) {
        ...

    // define the uvs for the vertices we just created.
          var uva = new THREE.Vector2
                (x / (width - 1), 1 - z / (depth - 1));
          var uvb = new THREE.Vector2
                ((x + 1) / (width - 1), 1 - z / (depth - 1));
          var uvc = new THREE.Vector2
                (x / (width - 1), 1 - (z + 1) / (depth - 1));
          var uvd = new THREE.Vector2
                    ((x + 1) / (width - 1),
                      1 - (z + 1) / (depth - 1));

          var face1 = new THREE.Face3(b, a, c);
          var face2 = new THREE.Face3(c, d, b);

          ...

          geometry.faces.push(face1);
          geometry.faces.push(face2);

          geometry.faceVertexUvs[0].push([uvb, uva, uvc]);
          geometry.faceVertexUvs[0].push([uvc, uvd, uvb]);
  }
}
```

The code to generate the correct UVs isn't that much different from what we saw earlier to calculate the position of the vertices. The only thing we need to keep in mind is that the range for both the U and V values runs from 0 to 1. Note that `geometry.faceVertexUvs` can contain multiple UV mappings. Generally, you set the mapping with index `0` when you want to map a texture. Three.js uses other indices for special purposes. We now configure the debug texture that we saw earlier; you can see that our custom geometry uses the complete texture.

Have a look at the following screenshot




This mapping is based on the UV information we specified. So, if you create a custom geometry, remember to also think about how a texture should be applied to it. Now we can very easily create very interesting-looking terrains. For instance, the following figure was created with a simple wood-like texture (`05.03-3D-plane-with-texture.html`):




As the final step for our custom geometry, we're going to make it easier to create. We're going to extend Three.js so that we can just use new `THREE.Terrain` to create this geometry.

# Creating a JavaScript object with a constructor

As a last step, we're going to make it easier for you to reuse the custom geometry we just created. To better understand this, you need to know a little about how inheritance works in JavaScript. A good tutorial can be found at the `http://martinrinehart.com/frontend-engineering/engineers/javascript/inher/masters/master-inher-crockford.html`.

The goal of this example is to change the `create3DTerrain` function to the following code snippet:

```
function create3DTerrain(width, depth, spacingX, spacingZ, height) {

    var terrain = new THREE.TerrainGeometry(
         width, depth, spacingX, spacingZ, height, scale);

    // setup the material
    ...

    // create the mesh
    ...

    scene.add(groundMesh);
}
```

So, instead of using this function to create the geometry by hand, we just want to make a call to `THREE.TerrainGeometry` just as we would do with the other geometries Three.js provides. The code for this example can be found in this source code: `05.04-Simple-3D-Buildings.html`. To accomplish this, we need to do the following two things:

- First, we have to register this object with Three.js
- After that, we have to make some small changes to our code to work with this new setup

To register this object, we've created the `registerObject` function. Have a look at the following code snippet:

```
function registerObject() {
    THREE.TerrainGeometry = function(
       width, depth, spacingX, spacingZ, height, scale) {
```

```
        THREE.Geometry.call(this);
        ...
    };

    THREE.TerrainGeometry.prototype =
              Object.create(THREE.Geometry.prototype);
}
```

With this setup, we registered the `THREE.TerrainGeometry` object and defined its constructor. In Three.js, all geometry objects extend from the basic `THREE.Geometry` object. To accomplish this, we need to make sure we set the prototype property of our newly created object to `Object.create(THREE.Geometry.prototype)`. The final step is to make sure that whenever we create this object, we always first call `THREE.Geometry.call(this)` to also call the constructor of the `THREE.Geometry` object we extend from.

Once we've got this skeleton set up, we only need a couple of small changes in our code. To get this to work, we changed the following items:

- All references to the geometry variable were replaced by the variable `this`. The reason is that our new object is also a `THREE.Geometry`, so there is no need to use a separate `THREE.Geometry` object.
- We moved the `getHighPoint` function inside the `THREE.TerrainGeometry` object, since it was only used in that context.
- We removed all the `THREE.Mesh` object- and material-related code, since we only wanted to create the geometry.

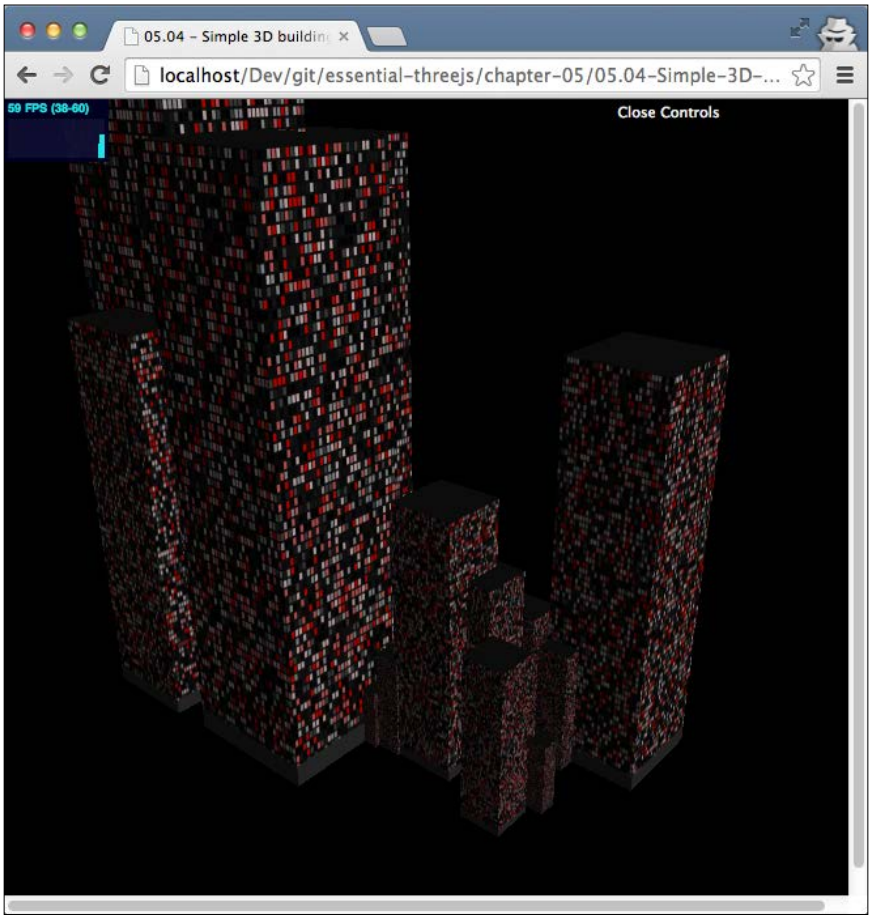
That's it. Now we can create a new, randomly-generated terrain by just calling the following line of code:

```
var terrain = new THREE.TerrainGeometry(width, depth, spacingX,
                  spacingZ, height, scale);
```

In the next section, we'll look at a different approach to procedural object generation, and we'll show you how you can create a procedural city.

# Creating a city from scratch

So far we've just created a terrain. In this small section, we'll show you how you can get started with creating a complete procedural city. We won't show you all the details here, but enough to get started. The goal for this section is shown in the following screenshot (`05.04-Simple-3D-Buildings.html`):



As you can see in this screenshot, we're going to create a number of skyscrapers that we randomly position. Let's walk through the code to accomplish this. The basic steps are as follows:

```
function createCity(buildingCount, rangeX, rangeY, scale) {
    // create the basic building block
    ...
    // Disable the texture for the roof
    ...
    // For each building create custom texture, set scale and add
}
```

We'll look at each of these steps a bit closer. The first step is to create a basic building using a simple `THREE.CubeGeometry` that we'll scale and texture to look like a building:

```
// create the basic buildingblock
var buildingBlock = new THREE.CubeGeometry(1, 1, 1);
buildingBlock.applyMatrix(
         new THREE.Matrix4().makeTranslation(0, 0.5, 0));
```

You can see that we first created a 1 x 1 x 1 `THREE.CubeGeometry` object, and we translated this geometry along the *y* axis with `0.5`. This last step moved the center of the cube for the *y* axis to the bottom. Now, if we scale the cube along the *y* axis, it will scale directly up, and we won't need to move the mesh afterwards.

In the previous screenshot, you can see that our roof has a different texture than the side walls. The standard way Three.js applies textures on cubes is to apply the texture to all the sides. In one of the earlier sections, you learned about UV mapping, and we'll use that to make sure the roof texture is a single color. We do this by changing the mapping for the two triangles (the fourth and fifth faces) that make up the roof:

```
// setup the texture for the roof
var uvPixel = 0.0;
buildingBlock.faceVertexUvs[0][4][0] =
          new THREE.Vector2(uvPixel, uvPixel);
buildingBlock.faceVertexUvs[0][4][1] =
          new THREE.Vector2(uvPixel, uvPixel);
buildingBlock.faceVertexUvs[0][4][2] =
          new THREE.Vector2(uvPixel, uvPixel);
buildingBlock.faceVertexUvs[0][5][0] =
          new THREE.Vector2(uvPixel, uvPixel);
```

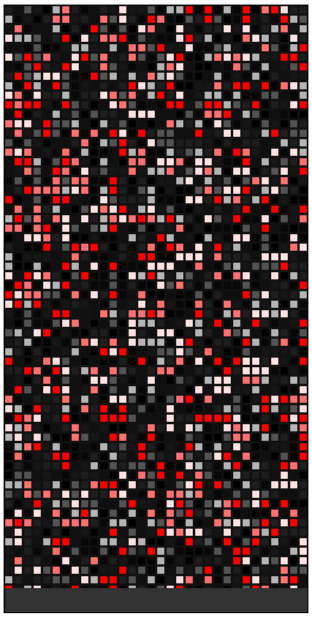
```
buildingBlock.faceVertexUvs[0][5][1] =
          new THREE.Vector2(uvPixel, uvPixel);
buildingBlock.faceVertexUvs[0][5][2] =
          new THREE.Vector2(uvPixel, uvPixel);
```

By pointing all the vertices of the faces to the same position on the texture, we make sure that the roof is of a uniform color. In this case, it is the color of the top-left pixel of the texture. Now that we've got the basic building block ready, we can start scaling and creating the buildings:

```
// create buildings
for (var i = 0; i < buildingCount; i++) {

  // create a custom material for each building
  var material = new THREE.MeshLambertMaterial();
  material.color = new THREE.Color(0xffffff);
  material.map = new THREE.Texture(generateBuildingTexture());

  // reduce bluriness
  material.map.anisotropy = renderer.getMaxAnisotropy();
  material.map.needsUpdate = true;

  // create the mesh
  var building = new THREE.Mesh(buildingBlock, material);
  var scale = ((Math.random() / 1.2) + 0.5) * scale;

  // scale the buildings
  building.scale.x = scale;
  building.scale.z = scale;
  building.scale.y = scale * 4;

  // position the buildings
  building.position.x =
            (Math.random() / 2 * rangeX) - rangeX / 2;
  building.position.z =
            (Math.random() / 2 * rangeY) - rangeY / 2;

  // add to scene
  scene.add(building);
}
```

In the previous code fragment, we first set up a material for each building. We do this because we're going to generate a custom texture for each building using the `generateBuildingTexture()` function, which we'll explain later. Normally, textures are a bit blurry to improve performance and make textures look more natural. In our case, however, we want to have the textures as crisp and sharp as possible. We can do this by setting `anisotropy` of the material to `renderer.getMaxAnisotropy()`. After we've set up the material, we create a `THREE.Mesh` object and randomly set the scale and position before adding it to the scene.

The only thing left to explore is how to create a random texture.
For each building in this example, we create a random texture using the `generateBuildingTexture()` function. This function returns textures that look something like the following screenshot:



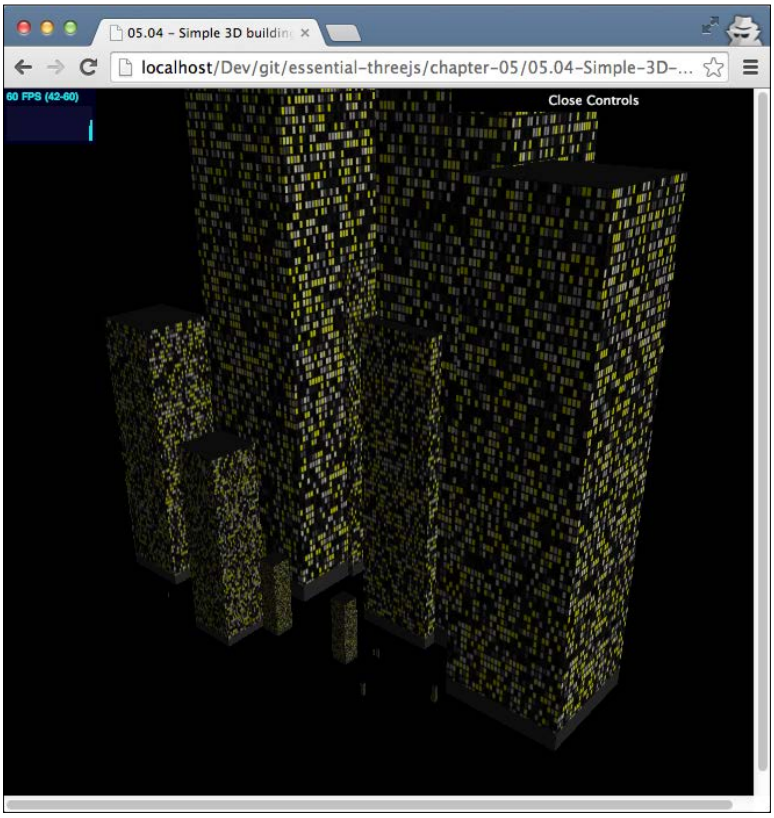
The code to create this canvas is shown in the following code snippet:

```
var scale = chroma.scale(
              ['black','#111111','#222222','white','red'])
              .domain([0,1],10);
function generateBuildingTexture() {
```

```
    var canvas = document.createElement("canvas");
    canvas.width = 256;
    canvas.height = 512;

    var ctx = canvas.getContext("2d");
    ctx.imageSmoothingEnabled = false;
    ctx.webkitImageSmoothingEnabled = false;
    ctx.mozImageSmoothingEnabled = false;

    ctx.fillStyle = '#111111';
    ctx.fillRect(0, 0, 512, 512);

    // fill in the windows
    for (var x = 0; x < 256; x += 8) {
        for (var y = 0; y < 490; y += 8) {
                  ctx.fillStyle = scale(Math.random()).hex();
                  ctx.fillRect(x + 1, y + 1, 6, 6);
      }
    }
    for (var x = 0; x < 256; x += 8) {
        for (var y = 490; y < 512; y += 8) {
                 ctx.fillStyle = '#333333';
                 ctx.fillRect(x + 1, y + 1, 8, 8);
      }
    }

    return canvas;
  }
```

The code isn't that difficult. We create a canvas and fill it with colored rectangles. To determine the color of the rectangle, we use the color scale, which you can see at the top of the code sample. We don't fill the complete texture with rectangles but make sure the lower part is a uniform color. This results in a texture that looks like a building at nighttime, with lots of windows, and a kind of storefront at the bottom (the uniform color).

By playing around with the variables (or the colors), you can create really interesting looking cities.

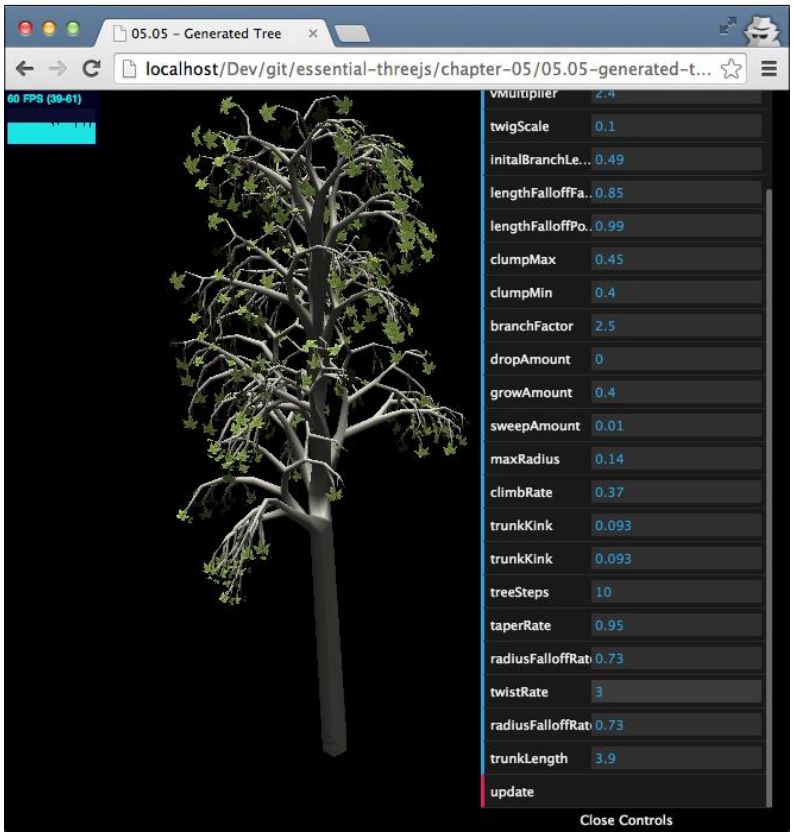



If you want to explore procedural city generation further, there is a great example on how to do this with Three.js at the following URL:
`http://learningthreejs.com/blog/2013/08/02/how-to-do-a-procedural-city-in-100lines/`

Before we move on to the next chapter, we're going to look at another way to quickly get some procedural geometries. In the next section, we'll use a standard library to generate the geometry and convert it into something Three.js can use.

# Creating parametric trees

For the last example, we'll look at how you can use an existing library to easily create parametric geometries. In this case, we'll show you how to create trees like the ones shown in the following screenshot (`05.05-generated-tree`):



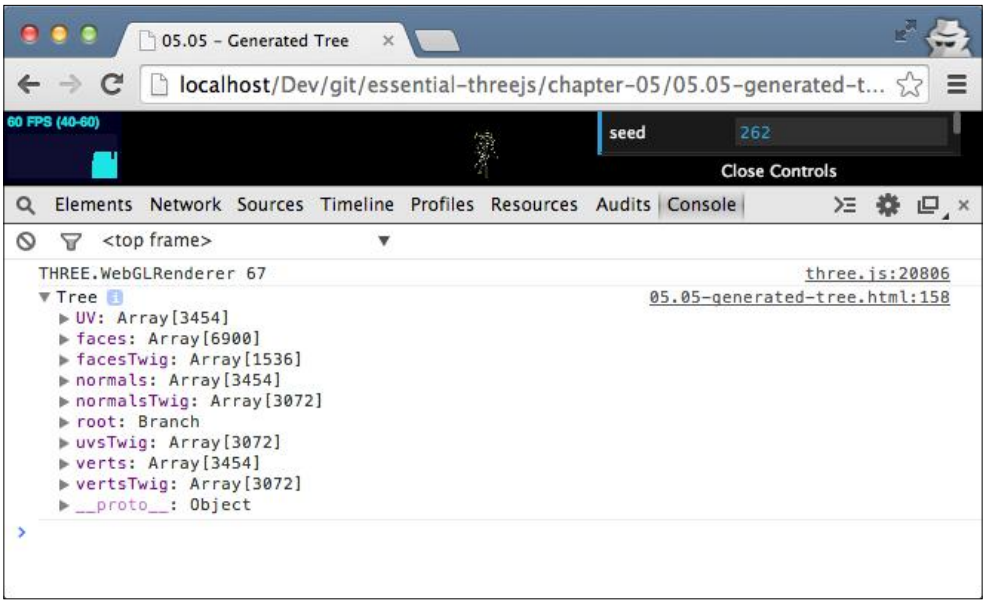

The library we use to create this generated tree is called `proctree.js`, which you can download from GitHub at the following link:

`https://github.com/supereggbert/proctree.js/`

This library allows you to specify a bunch of parameters, which are used to create a tree geometry. To get this to work with Three.js, we have to convert the information provided by this library to Three.js. Let's look at what `proctree.js` generates so that we can decide how to convert it:

```
var myTree = new Tree({
  // configuration
});

console.log(myTree);
```

The following screenshot shows this result in the Chrome console:




As shown in the previous screenshot, the `proctree.js` library creates the following components when generating a tree:

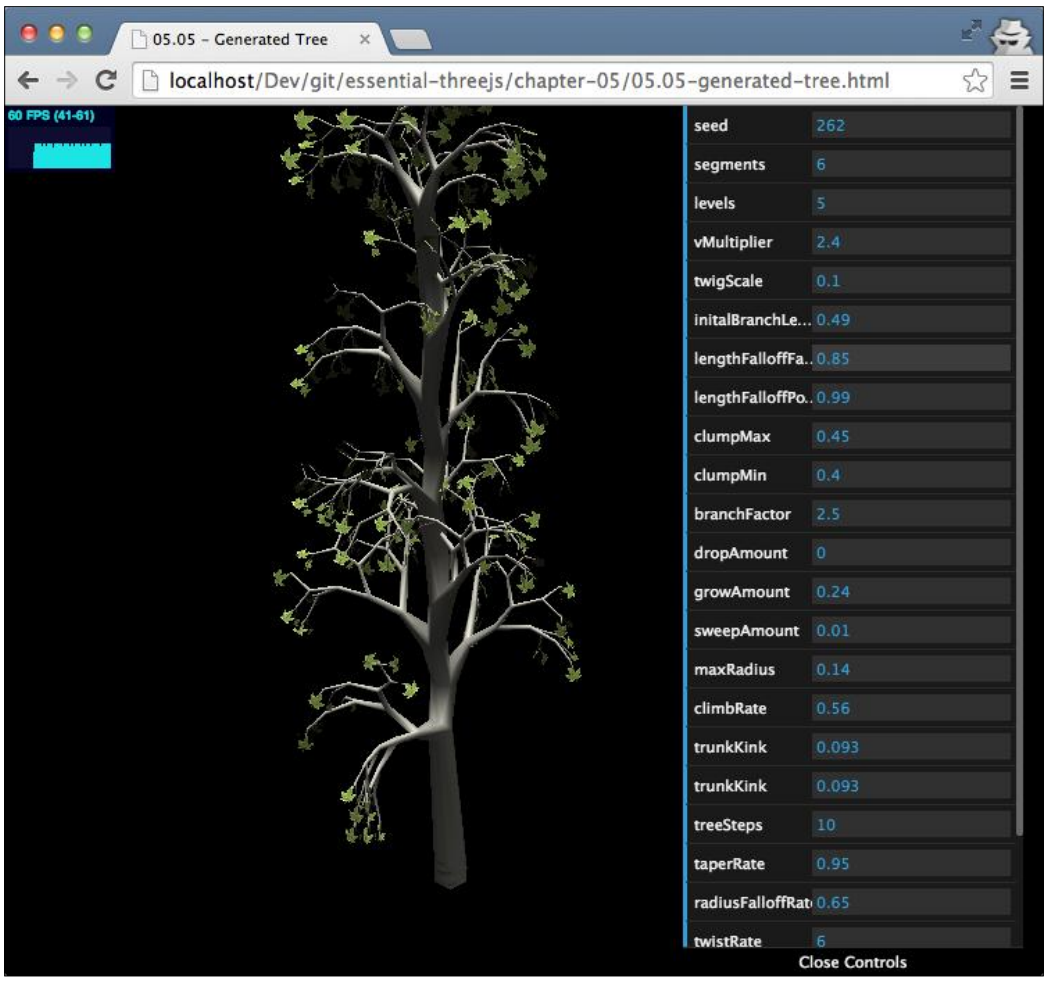
| Property | Description |
|---|---|
| `UV` | This property contains the UV value for each individual vertex of the trunk. |
| `faces` | This property shows an array of faces that together make up the trunk. Each face points to vertices stored in the `verts` property. |
| `facesTwig` | This property shows an array of faces that together make up the leaves of the tree. |

| Property | Description |
|---|---|
| `normals` and `normalsTwig` | These properties contain normals for all of the vertices. |
| `root` | This property contains internal information on how the tree was generated. This property is not needed to create a Three.js geometry. |
| `uvsTwigs` | This property contains the UV values for each individual vertex of the leaves. |
| `verts` | This property represents all the vertices that make up the trunk geometry. |
| `vertsTwig` | This property represents all the vertices that make up the leaves of the tree. |

What we need to do is create a `THREE.Geometry` object and fill the vertices, faces, and UV values based on the properties from the `proctree.js` tree. We'll show you how to do this for the trunk:

```
var trunkGeom = new THREE.Geometry();

 // convert the vertices
myTree.verts.forEach(function(v) {
  trunkGeom.vertices.push(new THREE.Vector3(v[0], v[1], v[2]));
});

 // convert the faces
myTree.faces.forEach(function(f) {
  trunkGeom.faces.push(new THREE.Face3(f[0], f[1], f[2]));
});

// setup uvsTwig
myTree.faces.forEach(function(f) {
  var uva = myTree.UV[f[0]];
  var uvb = myTree.UV[f[1]];
  var uvc = myTree.UV[f[2]];

  var vuva = new THREE.Vector2(uva[0], uva[1]);
  var vuvb = new THREE.Vector2(uvb[0], uvb[1]);
  var vuvc = new THREE.Vector2(uvc[0], uvc[1]);

  trunkGeom.faceVertexUvs[0].push([vuva, vuvb, vuvc]);
});

trunkGeom.computeFaceNormals();
trunkGeom.computeVertexNormals(true);
```

In this code fragment, we start by creating a `THREE.Geometry` object. Next, for each of the vertices in the `myTree.verts` array, we create a `THREE.Vector3`, which we add to the Three.js geometry. We repeat this process for the faces using the `myTree.faces` array. The final step we need to take is set up the UV mapping for each of the faces. In `proctree.js`, the UV values are stored for each vertex, so we need a small conversion before we can set them on `trunkGeom`. Finally, we calculate the face and vertex normals and we're done. Now we can create a `THREE.Mesh` object with our own choice of material and we've got a parametric tree in Three.js.




For instance, the preceding screenshot shows a completely different tree, which is generated using `proctree.js` with slightly different textures.

# Summary

In this chapter, we saw a lot of examples and code you can use to create your own geometries. This can be done either by creating them from scratch or using an external library. To create a geometry, you first have to specify its vertices and its faces. First you create the vertices, and based on these vertices, you can create the faces. When you define the faces, remember that Three.js only supports triangles as face. So when you create a face, use the `THREE.Face3` object.

For correct behavior, we also have to set the normal on the face. Three.js uses this information to determine how to color a face based on the available lighting. The easiest way to do this is to just let Three.js calculate the normal by calling `computeFaceNormals()`. Once the normal is set, you can define different shading styles. If you want to use smooth shading, where you don't see the individual faces, you first have to call `computeVertexNormals(true)`.

There are different ways of creating random geometries. The most straightforward one is using `Math.random()`. This, however, doesn't generate natural-looking terrains. If you want more natural-looking terrains, you can use a Perlin noise generator. If you create a custom geometry and want to apply textures, you also need to specify the UV mapping. This mapping determines which part of a texture should be shown on a face.

So far, we've used `WebGLRenderer` to render our scenes. In the next chapter, we'll use a different renderer, the `CSS3DRenderer`, to transform web pages into 3D.

# 6
# Combining HTML and Three.js with CSS3DRenderer

So far in this book, we've exclusively used the `THREE.WebGLRenderer` object to render our scenes. This specific renderer requires browsers that support WebGL to show a scene. Most modern desktop browsers have good support for WebGL, but mobile devices are currently a bit behind. Three.js offers two alternative renderers that you can use. With CanvasRenderer, you can directly draw on an HTML5 canvas, which is supported on pretty much all the browsers and devices. However, CanvasRenderer has one big disadvantage; its performance isn't that great. With the latest versions of Three.js, we get another option: CSS3DRenderer. With this renderer, Three.js uses CSS3 3D transforms to render a scene. CSS3DRenderer runs well on most devices and even uses hardware acceleration. The disadvantage with CSS3DRenderer is that it doesn't support geometries, materials, and lights.

In the previous chapter, we looked at how you can create geometries from scratch and render them with the `THREE.WebGLRenderer` object. In this chapter, we'll explain how you can use CSS3DRenderer and what features are available. We'll show this through the following examples:

- We'll start by setting up a simple skeleton page that you can use as a starting point for when you want to create a CSS3DRenderer-based visualization.
- Even though CSS3DRenderer doesn't support geometries and meshes, we can still simulate this. We'll show how you can still create geometry-like objects by reusing the information from a `THREE.Geometry` object.

- An interesting aspect is that by using Three.js and CSS3DRenderer, you can easily animate and move HTML elements in 3D. We'll extend the previous example by adding animations.
- As a last example, we'll look at how you can use CSS3DRenderer to simulate a particle system by using CSS3DSprite.

Let's first look at what the most simple CSS3DRenderer-based page looks like.

# Setting up a CSS3DRenderer skeleton

To set up a skeleton page for CSS3DRenderer, we don't really have to change much with regards to the template that we set up in *Chapter 1*, *Get Up and Running with Three.js*. The main thing that we do here is remove the elements that aren't supported, such as lights. The first thing we need to do though is include the `CSS3DRenderer` object as it isn't included in the standard Three.js JavaScript file. This is done using the following line of code:

```
<script src="../libs/CSS3DRenderer.js"></script>
```

Now, let's set up the complete `init()` function just like we did in the first chapter for the `WebGLRenderer` object. It is described in the following code:

```
function init() {

  scene = new THREE.Scene();

  camera = new THREE.PerspectiveCamera(45
    , window.innerWidth / window.innerHeight, 0.1, 1000);
  renderer = new THREE.CSS3DRenderer();
  renderer.setSize(window.innerWidth, window.innerHeight);
  renderer.domElement.style.position = 'absolute';
  renderer.domElement.style.top = 0;

  camera.position.x = 500;
  camera.position.y = 500;
  camera.position.z = 500;
  camera.lookAt(scene.position);

  // create an example object and add to scene
  var cssElement = createCSS3DObject(string);
```

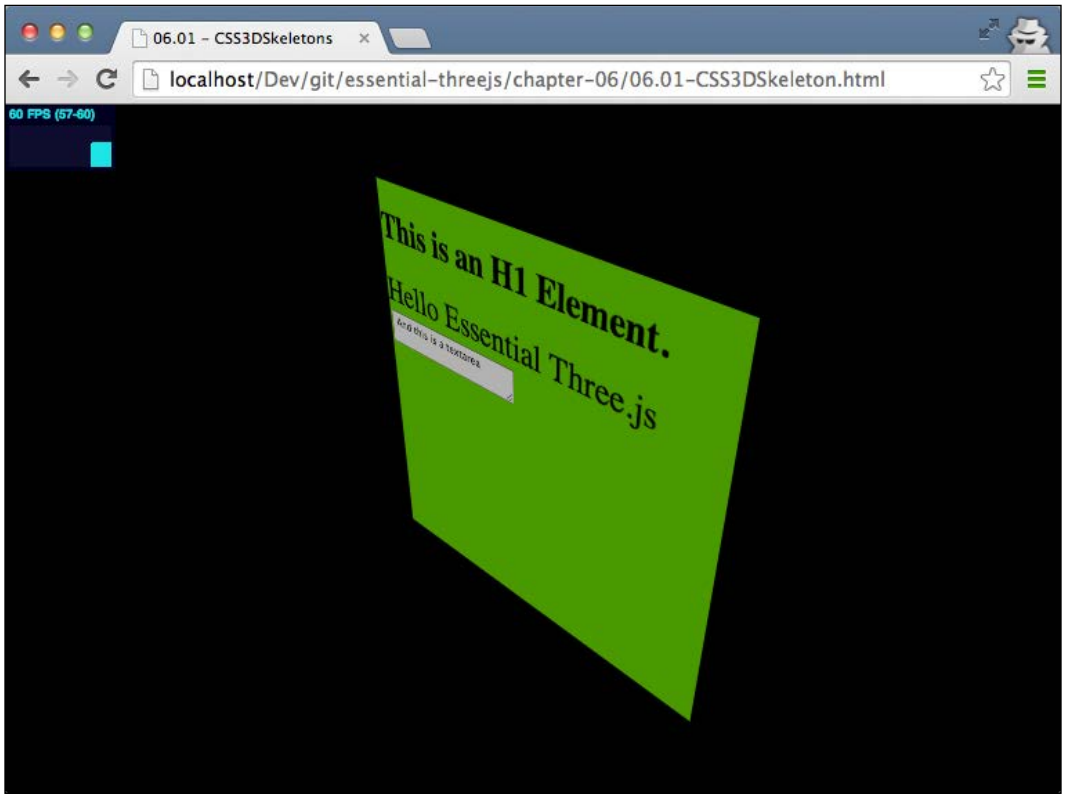
```
    cssElement.position.set(100, 100, 100);
    scene.add(cssElement);

    // add the output of the renderer to the html element
    document.body.appendChild(renderer.domElement);

    render();
}
```

In the previous code fragment, you can see that instead of creating a `THREE.WebGLRenderer` object, we create a `THREE.CSS3DRenderer` object. The rest of the `init()` function hasn't changed much, except that we removed all the light and shadow references. In this `init()` function, we also add an HTML element through the `createCSS3DObject` function. This is described in the following code:

```
function createCSS3DObject(s) {
  // create outerdiv and set inner HTML from supplied string
  var div = document.createElement('div');
  div.innerHTML = s;

  // set some values on the div to style it, standard CSS
  div.style.width = '370px';
  div.style.height = '370px';
  div.style.opacity = 0.7;
  div.style.background =
    new THREE.Color(Math.random() * 0xffffff).getStyle();

  // create a CSS3Dobject and return it.
  var object = new THREE.CSS3DObject(div);
  return object;
}
```

The `createCSS3DObject` function takes a string as an argument and creates a DOM element from it by setting it as the `innerHTML` object for a `div` element. Next, we style the `div` element a little bit and create a `THREE.CSS3DObject`, which we will return. We can wrap any HTML element inside the `THREE.CSS3DObject` and render it through CSS3DRenderer.

When you open the `06.01-CSS3DSkeleton.html` file from the source code, you will see the following screenshot:




The previous screenshot shows the following HTML code rendered with CSS3DRenderer:

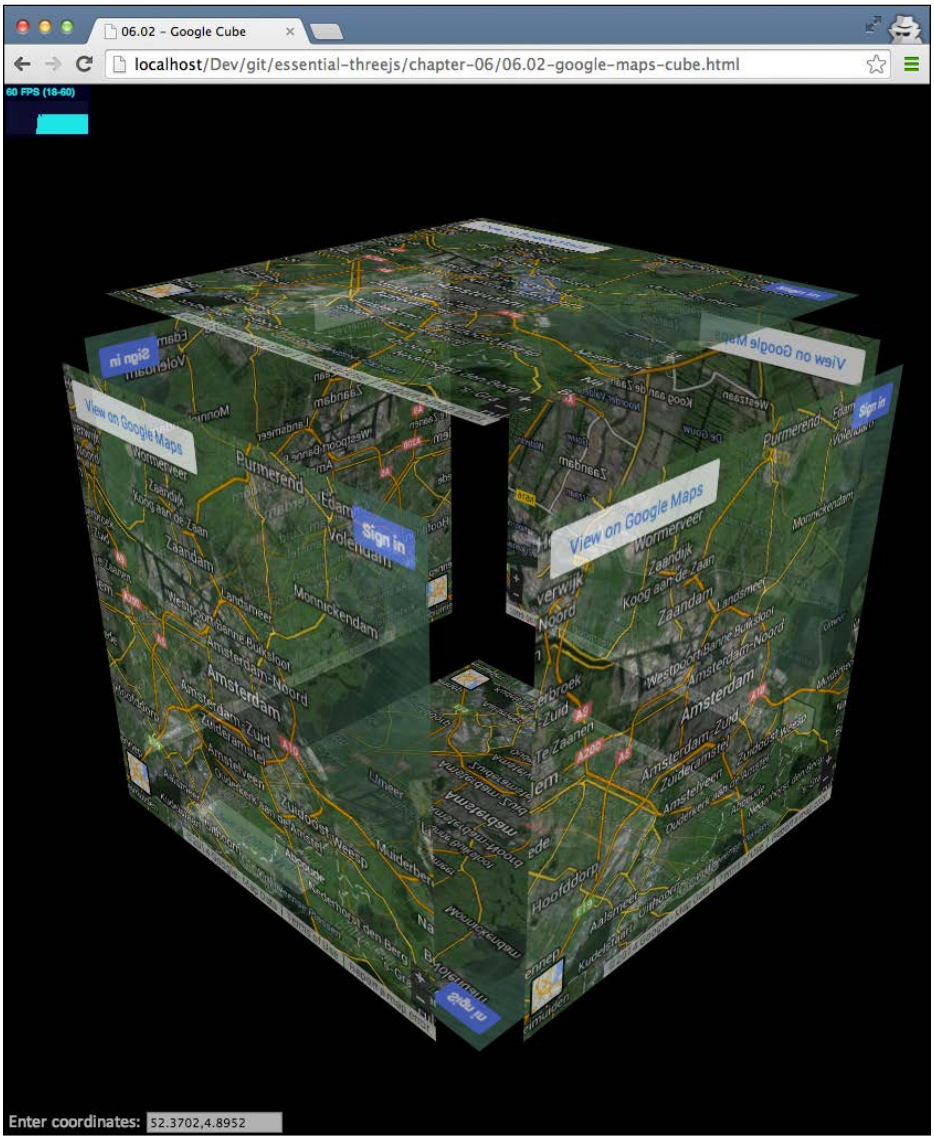
```
var string = '<div>' +
  '<h1>This is an H1 Element.</h1>' +
  '<span class="large">Hello Essential Three.js</span>' +
  '<textarea> And this is a textarea</textarea>' +
'</div>';
```

The interesting thing is that you can still select text, enter values in the `textarea` element, and resize the `textarea` element, just as you would do on a normal page. In the next section, we'll expand on this example by rendering maps on a 3D cube.

# Creating an interactive 3D Google Maps cube

Google Maps provides an API that we can use to show maps of specific areas in our own web pages as an iframe. As iframes are just normal HTML elements, we can transform these with Three.js using CSS3DRenderer.

Let's first look at the example that we're going to create in this section.



In the previous screenshot (the `06.02-google-maps-cube.html` file), you see a 3D cube where each side shows a map from Google Maps. By entering coordinates at the bottom, you can change each side of the cube. To do this, we need to perform the following steps:

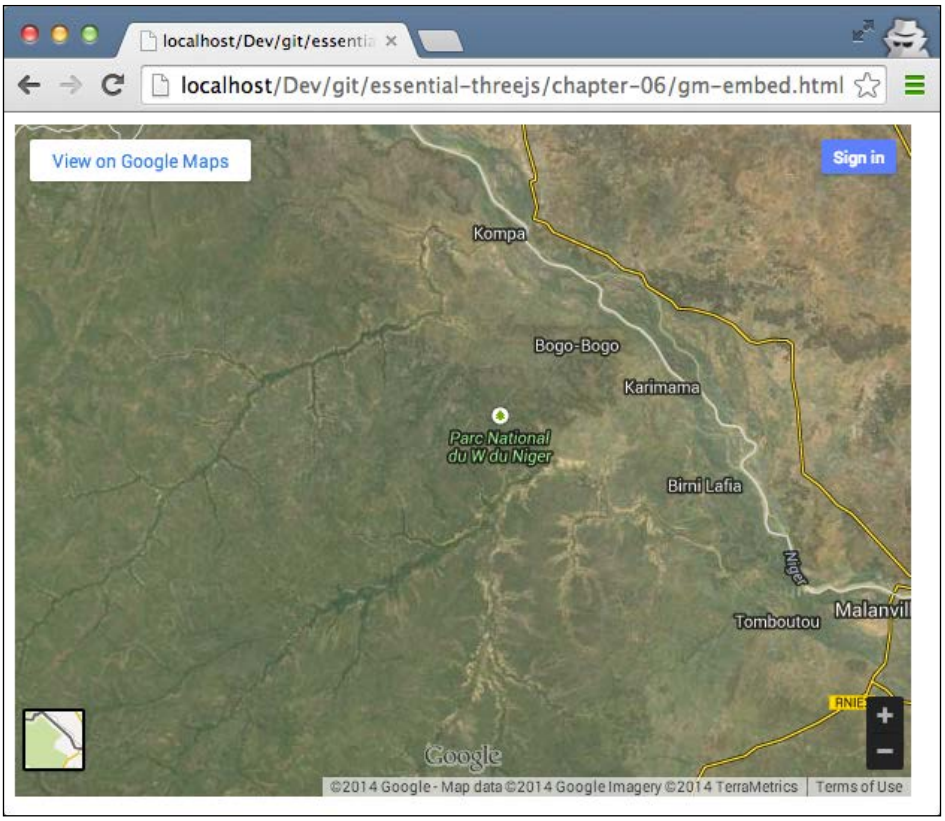
- First, we need to get the HTML code for the maps to be shown. We'll use the Google Maps Embed API for this, which uses an iframe.
- Now, we can create a `THREE.CSS3DObject` from this iframe and add it to the scene.
- The last step we need to perform is position and rotate the `THREE.CSS3DObject` correctly so that the result looks like a cube.

# Displaying a part of Google Maps using HTML

The first step is really easy. Google provides a Google Maps Embed API where you can specify which part of the map needs to be shown through the location parameter. For instance, consider the following iframe:

```
<iframe width="600" height="450" frameborder="0" style="border:0"
  src="https://www.google.com/maps/embed/v1/view?zoom=10&maptype
  =satellite&center=10,2&key=AIzaSyAVUZTKZz1e6hbEwZOT8CmWfoMhe
  gHL7bs"></iframe>
```

If you use the preceding iframe, you'll see a nice looking map directly inside your own page, as shown in the following screenshot:




As an iframe is also a normal HTML element, we can use `THREE.CSS3DObject` to transform and rotate it.

To add an object to the scene, we need to reuse the code, that is, the `createCSS3DObject` function that we saw in the previous example to create a `THREE.CSS3DObject` from a string; this time, we will provide an iframe as the content. Consider the following code:

```
var iframe = '<iframe width="600" height="450" frameborder="0"' +
  ' style="border:0" src="https://www.google.com/maps/embed/' +
  'v1/view?zoom=10&maptype=satellite&center=LOCATION' +
  '&key=AIzaSyAVUZTKZz1e6hbEwZOT8CmWfoMhegHL7bs"></iframe>';
```

Note that we haven't set the center argument yet. We do this so that we can easily reuse this string to show other maps.

# Positioning and rotating the element

To position the maps at the correct location and rotation, we're going to use information from the standard `THREE.CubeGeometry` object. As we've seen in the previous chapter, to create a geometry, we need to define its vertices and its faces. The faces also have an additional property called a normal, which shows how the face is aligned. To create the cube you saw at the beginning of this section, we'll reuse the information from the faces of a standard `THREE.CubeGeometry` object. To make it easier to use, we've created the `createSides(s, geometry)` function, which takes an HTML string and a geometry as its input. The HTML string defines how the side will look, and the geometry will be used to position and rotate the sides. The following code fragment shows this function:

```
function createSides(s, geometry) {

  // iterate over all the sides
  for (var iFace = 0;
    iFace < geometry.faces.length; iFace += 2) {

    // create a new object based on the supplied HTML String
    var side = createCSS3DObject(s);

    // get this face and the next which both make the cube
    var face = geometry.faces[iFace];
    var faceNext = geometry.faces[iFace + 1];

    // reposition the sides using the center of the faces
    var centroid = new THREE.Vector3();
    centroid.copy( geometry.vertices[face.a] )
      .add( geometry.vertices[face.b] )
      .add( geometry.vertices[face.c] )
      .add( geometry.vertices[faceNext.a] )
      .add( geometry.vertices[faceNext.b] )
      .add( geometry.vertices[faceNext.c] )
      .divideScalar( 6 );

    side.position.x = centroid.x;
    side.position.y = centroid.y;
    side.position.z = centroid.z;
    // Calculate and apply the rotation for this side
    ...

    // add to the scene
    scene.add(side);
  }
}
```

In this function, we walk through all the faces, two at a time, of a geometry. We need to combine two faces. We need to do this because Three.js works with triangular faces, and our objects represent rectangles. For every two faces, we create a CSS3DObject with the `createCSS3DObject` function. Once we have CSS3DObject, we need to rotate and position it. The first thing that we do is position the CSS3DObject. For this, we calculate the center of the two faces that make up the side. The result is stored in the `centroid` property. The last step before we add the CSS3D Google Maps element to the scene is rotating the object. The code for this is shown here:

```
// Up contains the normal of our side.
var up = new THREE.Vector3(0, 0, 1);
var normal = geometry.faces[iFace].normal;;

// We calculate the axis on which to rotate by
// selecting the cross of the vectors
var axis = new THREE.Vector3();
axis.crossVectors(up, normal);

// based on the axis, in relation to our normal vector
// we can calculate the angle.
var angle = Math.atan2(axis.length(), up.dot(normal));
axis.normalize();

// now we can use matrix function to rotate the object so
// it is aligned with the normal from the face
var matrix4 = new THREE.Matrix4();
matrix4.makeRotationAxis(axis, angle);

// apply the rotation
side.rotation.setFromRotationMatrix(matrix4);
```

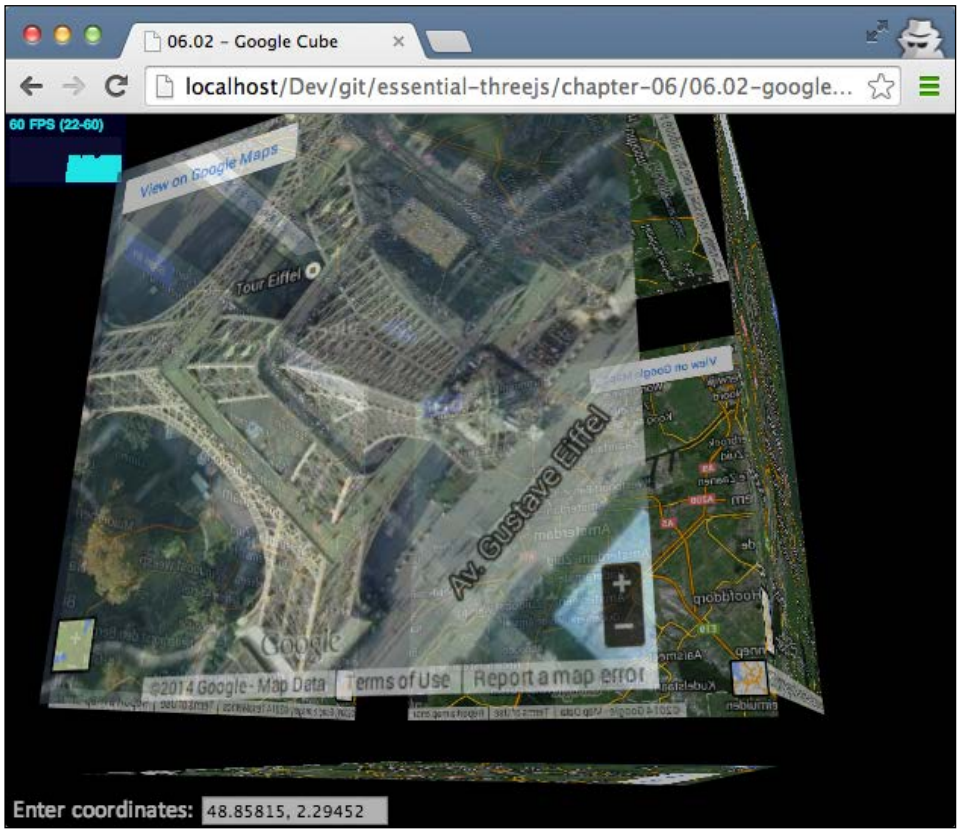
To align our side object with the normal of the face of the provided geometry, we need to perform some matrix calculations. Explaining this in detail here is a bit out of scope, but we'll walk you through the following most important steps:

1. The first thing that we do is determine the cross product of the normal of the face and the normal of our side (up). This gives us the axis around which we're going to rotate. The result of this is a vector that is perpendicular to both the up and the normal vector.
2. To translate to the new position, we don't just need to know the axis around which we're going to rotate, but also the angle (the amount) by which we're going to rotate. We can do this using the `Math.atan2` function.

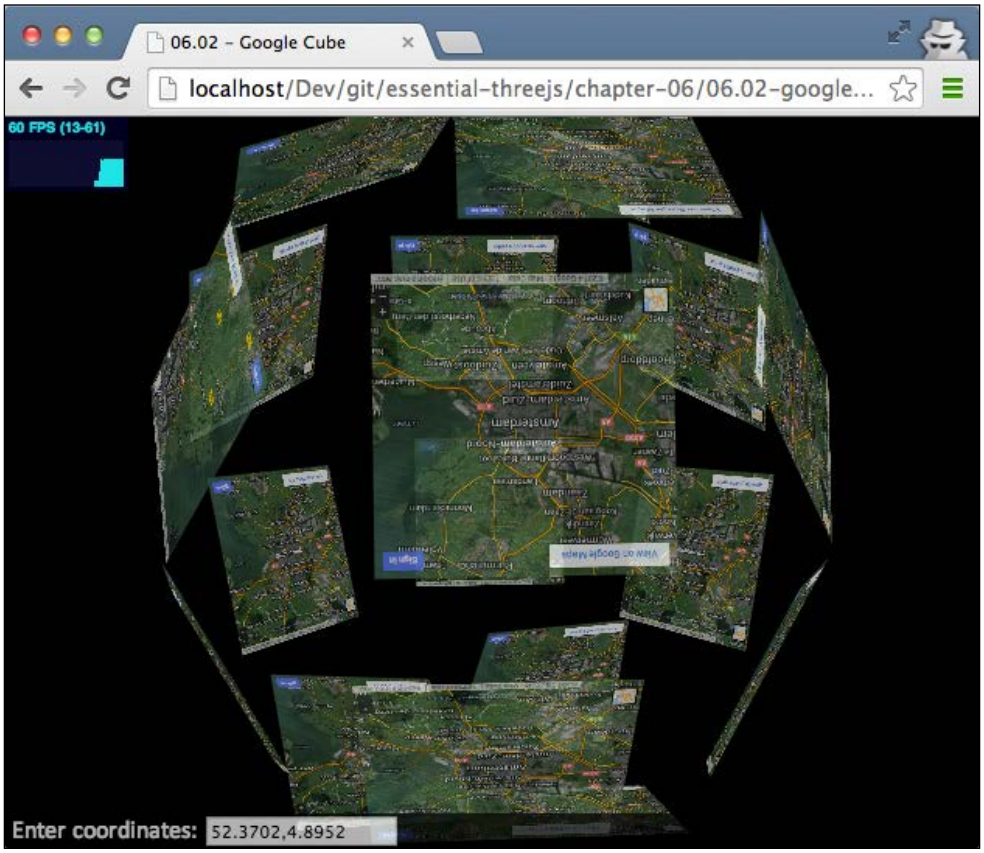
3. At this point, we've got the axis around which to rotate and the angle to rotate. Three.js uses matrix transformations to apply rotations and provides a helper function, `makeRotationAxis`, that we can use to get the correct rotation matrix.
4. All that is left to do is apply the rotation on our CSS3DObject. We do this by using the `setFromRotationMatrix` function on the `rotation` property of the side object.

With all these steps, our CSS3DObjects are automatically positioned and rotated correctly; the following screenshot is displayed as a result of this.

Like with our first example, the maps are fully functional here too. You can zoom in, rotate, and do anything else that Google Maps allows. In this example, we also added a `THREE.OrbitControl` object so that you can use your mouse to move around the scene. Additionally, we added a small menu at the bottom where you can enter the coordinates and update the sides of the cube. The following screenshot, for instance, shows the Eiffel Tower:



As we don't specify the rotation of the sides explicitly, we can easily extend this example to show other shapes, such as a sphere using the `THREE.Sphere` object. This is shown in the following screenshot:



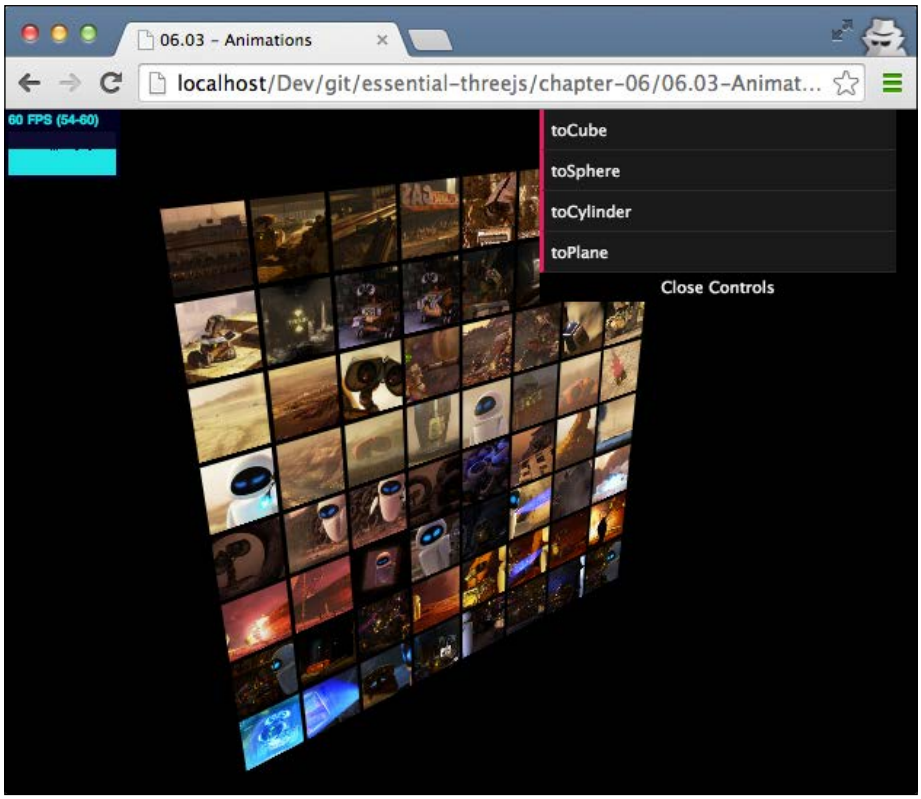

All you need to do to accomplish this is provide a different geometry to the `createSides` function as shown in the following code fragment:

```
createSides(iframe.replace("LOCATION",loc), new
  THREE.SphereGeometry(700,6,4));
```

In the following section, we'll dive a bit deeper into this approach and also add animations through TweenJS.

# Animating HTML elements with TweenJS

In this section, we'll use the same approach as in the previous section to create 3D objects that are built up from HTML elements. The following screenshot shows an impression of what we'll create in this section (the `06.03-Animations.html` file):



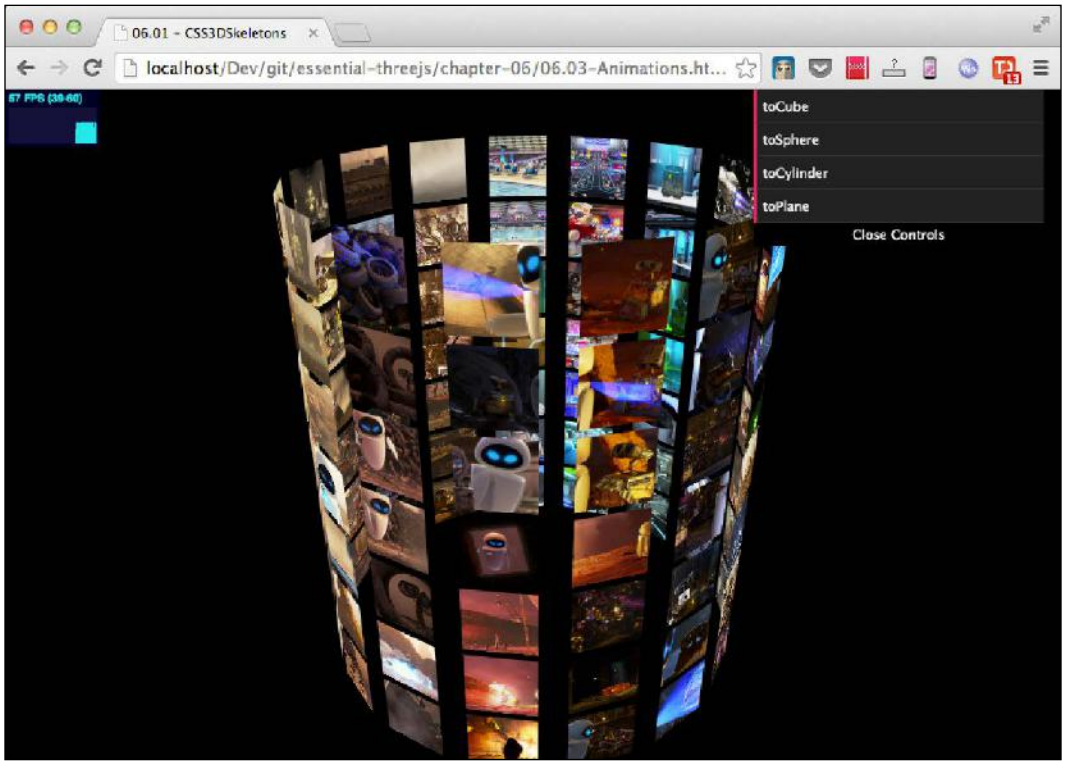

If you run this example, you'll see a rotating 3D object built up from a number of images. Using the menu on the top right-hand side of the window, you can change the shape of the 3D object, and the various components will automatically move to their correct new position and rotation. Let's start with the beginning and look at the HTML element that we use to show the image.

# Using images as the input

Normally, in Three.js, when you want to use an image on a 3D object, you create a `THREE.Texture` object and apply it to a 3D object through a material. In this case, it is much easier. We can just create an `img` element and wrap it in a CSS3DObject. This is described in the following code:

```
function createCSS3DObject(iFace) {
  var div = document.createElement('div');
  var img = document.createElement('img');

  var nrString = ("000" + iFace * 4).substr(-3, 3);
  img.src = '../assets/screens/WALL-E-' + nrString + ".jpg";
  img.width = 140;

  div.appendChild(img);
  div.style.opacity = 0.8;

  var object = new THREE.CSS3DObject(div);

  return object;
}
```

As you can see, in the `createCSS3DObject` function, we created an `img` element that we pointed to a file in the `../assets/screens/` folder. The image is wrapped in a `div` element, which is used to create the CSS3DObject. Positioning and rotating these CSS3DObjects is done in the same way as we saw in the previous chapter. So, when we use a new `THREE.CylinderGeometry(12,12,27,15,7,true)` as an input, we get the following result:



Even though this looks nice, we can make it even more interesting by adding animations for when we update the geometry.

# Setting up the animations

To set up the animation to move from one form to the other, we have to perform the following steps:

1. To animate the HTML elements from their position and rotation in one geometry (for example, the `THREE.PlaneGeometry` object) to another geometry (for example, the `THREE.SphereGeometry` object), we need to store their current position and rotation and the position and rotation they would get in the target geometry.
2. Once we know the current and target positions and rotations, we can use them to set up an animation using TweenJS to slowly change the position and rotation of the elements from the current to the target position and rotation.
3. We also need to take into account the number of elements we have. If the target geometry we're animating has more faces, we need to add images; if there are less faces, we have to ensure that elements are removed.

We've created a single function that takes care of these steps. It is called the `updateStructure()` function. We'll first show you the structure of this method and then dive into the interesting parts. Consider the following code:

```
function updateStructure(geometry, offset) {

  // get the position where we need to move elements
  positionAndRotation = getPositionAndRotation(geometry,offset);

  // setup the tween to move to rotation and position
  // from the current position to target position
  ...

  // walk through the positionAndRotation set and either move or
  // create the elements
  for (var i = 0; i < positionAndRotation.length; i++) {

    if (currentElements.length > i) {
      // we need to move one of the existing ones.
      ...
    }
    else {
      // create a new one, and set it's position of screen
```

```
        ...
      }
    }

    // finally determine the elements that aren't needed anymore
    for (var i = positionAndRotation.length;
      i < currentElements.length; i++) {
      toBeRemovedElements.push(currentElements[i]);
    }
    // and remove them from the scene
    for (var i = 0; i < toBeRemovedElements.length; i++) {
      scene.remove(currentElements.pop());
    }
  }
```

Let's look at some of the details of this function in the following sections.

## Determining the target position and rotation

The implementation of the `getPositionAndRotation` function that is used to determine the new position of the elements works in the same way as we saw in the Google Maps example. The only difference is that this time we don't set the target rotation and position directly on a CSS3DObject, but we return an array that contains the following code structure for each element:

```
{
  pos: THREE.Vector3(),
  rot: THREE.Matrix4()
}
```

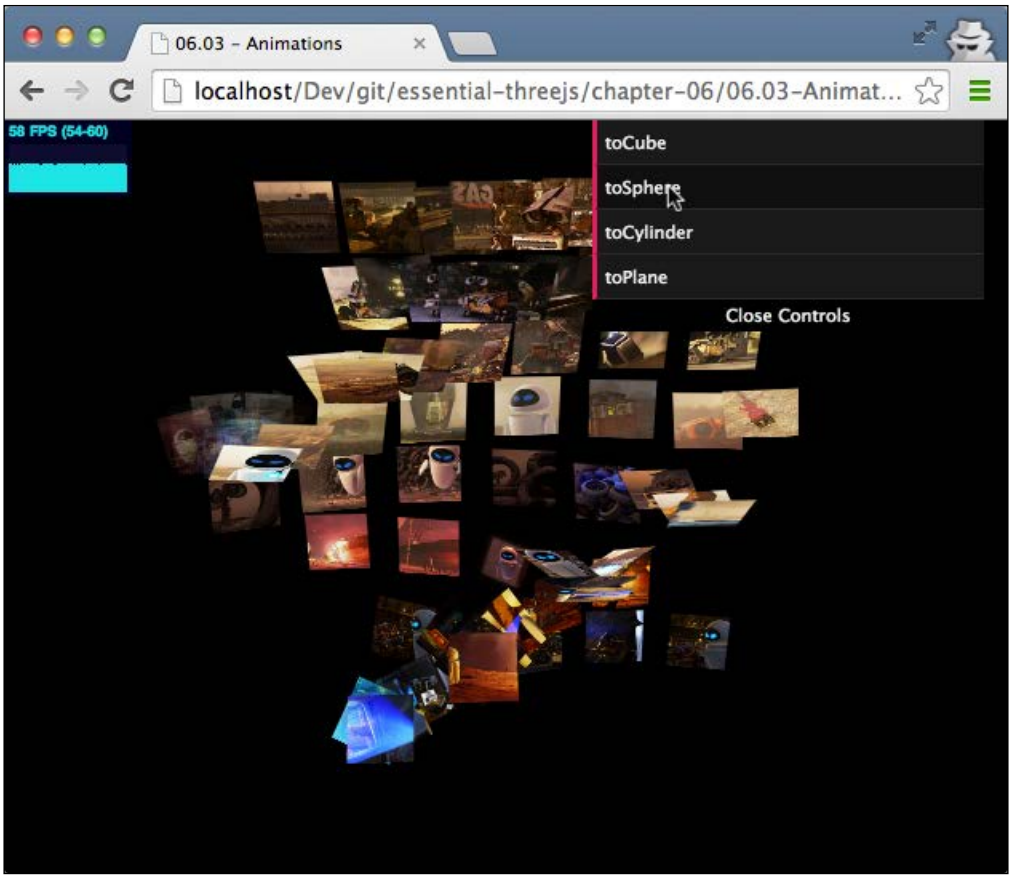
## Configuring TweenJS to run the animation

We once again use TweenJS to implement the animation. Consider the following code:

```
var tweenIn = new TWEEN.Tween({opacity: 0})
  .to({pos: 1.0}, 3000).easing(TWEEN.Easing.Sinusoidal.InOut)
  .onUpdate(function () {
  var toSet = this.pos;
  newlyAddedElements.forEach(function (cssObject) {
    cssObject.element.style.opacity = toSet;
  });

  var i = 0;
  currentElements.forEach(function (cssObject) {
    // get the elements start /target position
    var currentPos = positionAndRotation[i].currentPos;
```

```
        var targetPos = positionAndRotation[i].pos;

        // also get the elements start / end rotations
        var currentRotation =
                      positionAndRotation[i].currentRotation;
        var targetRotation = new THREE.Euler();
        targetRotation.setFromRotationMatrix
          (positionAndRotation[i].rot);

        // use the tween to slowly move the elements
        if (currentPos) {
          cssObject.position.x = currentPos.x +
            (targetPos.x - currentPos.x) * toSet;
          cssObject.position.y = currentPos.y +
            (targetPos.y - currentPos.y) * toSet;
          cssObject.position.z = currentPos.z +
            (targetPos.z - currentPos.z) * toSet;

          cssObject.rotation.x = currentRotation.x +
            (targetRotation.x - currentRotation.x) * toSet;
          cssObject.rotation.y = currentRotation.y +
            (targetRotation.y - currentRotation.y) * toSet;
          cssObject.rotation.z = currentRotation.z +
            (targetRotation.z - currentRotation.z) * toSet;
        }
        i++;
      });
    });
```

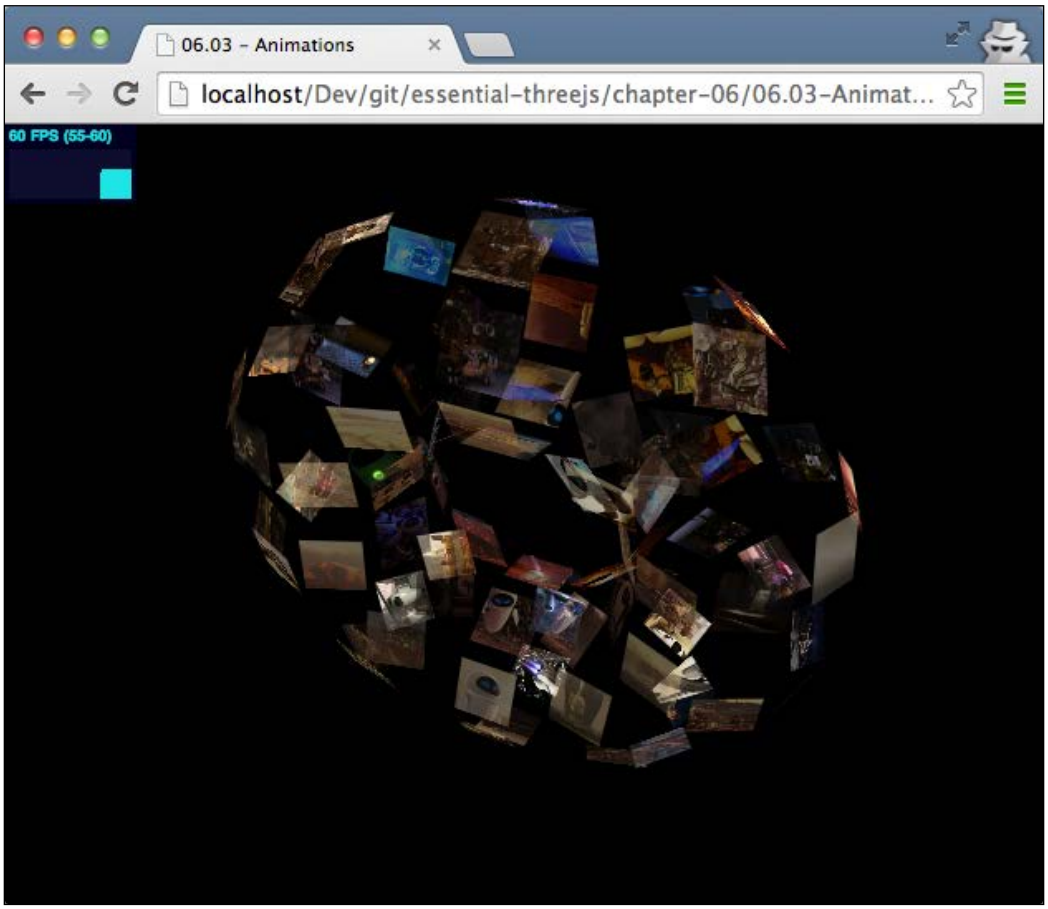
When we run this tween, the `pos` property changes from `0` to `1`. In the `onUpdate` function, we use this property to calculate the new position of the element based on its original position (`currentPos`), the target position (`targetPos`), and how far we're done with the tween (`pos`). We do this for the *x*, *y*, and *z* attributes of the position and do the same with the *x*, *y*, and *z* properties of the rotation. The result is that when we call this function, all the elements will slowly move and rotate to their new position and rotation. This is shown in the following screenshot:




Just like with the previous example, it's really easy to add new shapes. For instance, consider the following code:

```
updateStructure(new THREE.TorusGeometry(20,10,8,10),25);
```

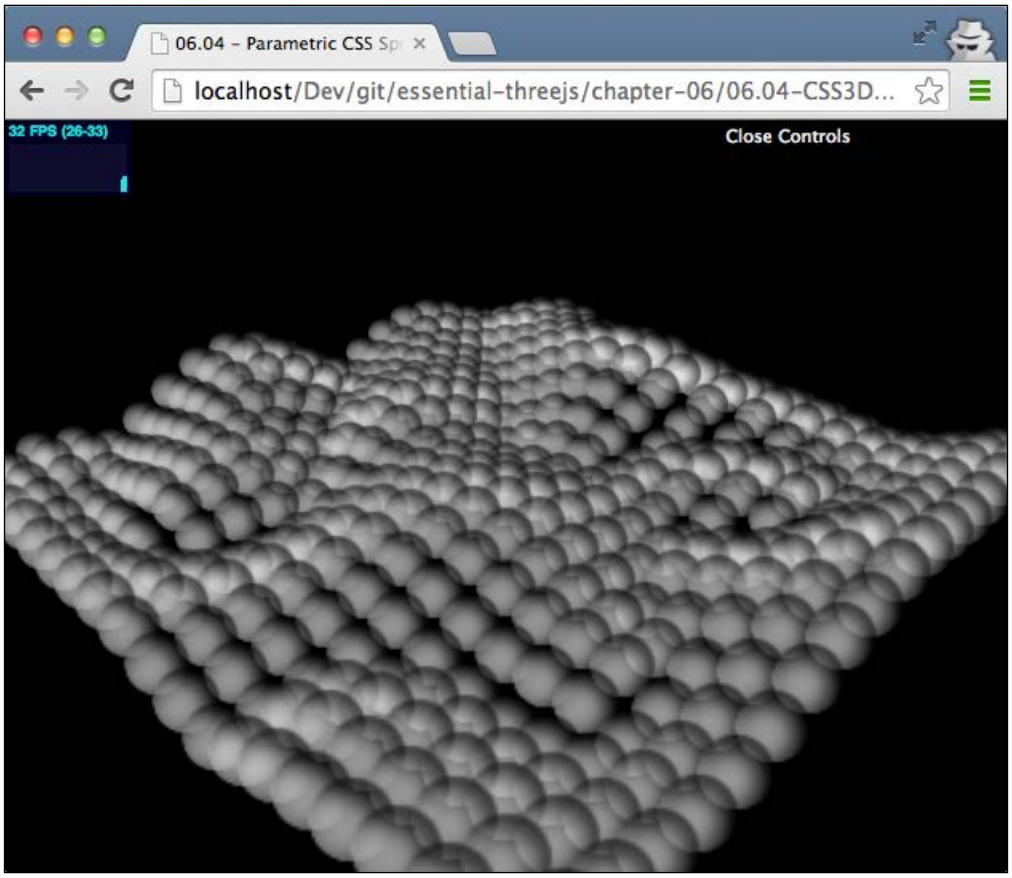
This code results in a torus-like structure, as shown in the following screenshot:




In the last section of this chapter, we'll look at another CSS3DRenderer feature called CSS3DSprite.

# Creating a parametric terrain using CSS sprites

So far, we've shown you examples where we've used `THREE.CSS3DObjects` to show our HTML elements. Three.js, however, provides another way in which we can show and animate HTML elements using a `THREE.CSS3DSprite` object. The main difference is that a CSS3DSprite always faces the camera directly. To demonstrate this, we've extended the example from the previous chapter where we created a random terrain. This time, instead of defining faces, we will render each vertex of the generated terrain geometry as a CSS3DSprite. Basically, what we are doing is creating a particle system using CSS3DSprites. The result will look something similar to what is shown in the following screenshot (the `06.04-CSS3DSprites.html` file):



# Creating a 3D terrain using sprites

To accomplish a 3D terrain, we only have to make a couple of small changes to the `05.02-3D-plane-from-scratch-perlin.html` example from the previous chapter. The first change is in the method we've used to create the 3D terrain. In the previous chapter, we also used this method to set up faces and calculate normals. This time, we only have to randomly determine the position of the vertices. Consider the following code:

```
function create3DTerrain
  (width, depth, spacingX, spacingZ, height) {

  var date = new Date();
  noise.seed(Math.random() * date.getMilliseconds());

  // first create all the individual vertices
  var geometry = new THREE.Geometry();
  for (var z = 0; z < depth; z++) {
    for (var x = 0; x < width; x++) {
      var yValue = Math.abs(
        noise.perlin2(x / 8, z / 8) * height * 2);
      var vertex = new THREE.Vector3(
        x * spacingX, yValue, z * spacingZ);
      geometry.vertices.push(vertex);
    }
  }

  return geometry;
}
```

After we call this method, we'll have a `THREE.Geometry` object whose vertices we'll use to position the `THREE.CSS3DSprite` objects as shown in the following code:

```
var currentGeometry = create3DTerrain(21, 21, 21, 21, 50);
var targetGeometry = create3DTerrain(21, 21,21, 21, 50);
var container = new THREE.Object3D();

// for each vertices add a sprite
geometry.vertices.forEach(function(e) {
  var cssObject = new THREE.CSS3DSprite(createDiv());
  cssObject.position = new THREE.Vector3(e.x, e.y, e.z);
  container.add(cssObject);
});
```

The constructor of the `THREE.CSS3DSprite` object takes an HTML element as its argument. We create this element in the `createDiv` function. This is described in the following code:

```
function createDiv() {

  var img = document.createElement('img');
  img.src = "../assets/textures/particles/ball.png";
  img.height = 29;
  img.style.opacity = 0.6;

  return img;
}
```

This time, we just create a simple HTML `img` element. The result of these steps is a random terrain where on each vertex, the `ball.png` image will be shown.

# Animating the terrain with TweenJS

In the examples in this book, we've already used TweenJS a couple of times to animate transitions between colors, positions, and rotations. This time, we'll use TweenJS to animate the transition from one randomly generated terrain (`currentGeometry`) to another (`targetGeometry`). Let's look at how we transition from `currentGeometry` to `targetGeometry`. This is described in the following code:

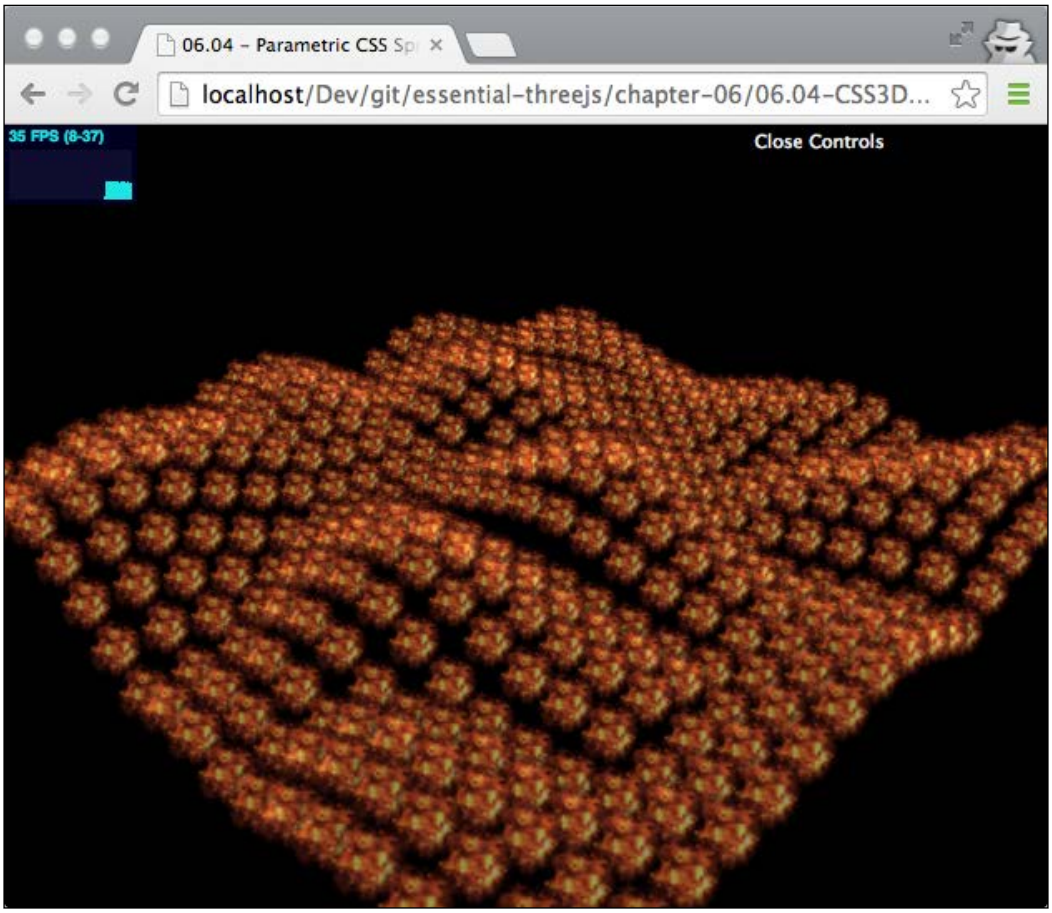
```
var tweenIn = new TWEEN.Tween({pos: 0})
  .to({pos: 1}, 1000).easing(TWEEN.Easing.Sinusoidal.InOut)
  .onUpdate(function () {
  var target = this.pos;
  var i = 0;
  container.children.forEach(function (e) {
    e.position.y = currentGeometry.vertices[i].
    y - (currentGeometry.vertices[i].
    y - targetGeometry.vertices[i].y) * target;
    i++;
  });
}).onComplete(function () {
  currentGeometry = targetGeometry.clone();
  targetGeometry = create3DTerrain(21, 21, 21, 21, 50);
  this.pos = 0;
});
```

What happens in this tween is that for each of our created CSS3DSprite objects, we slowly change the `y` position from its original position to the `y` position of the relevant vertex in the `targetGeometry` object. At the end of this tween, all the CSS3DSprites will have moved to their new position. In the `onComplete` function of the tween, which is called when the transition ends, we set the `currentGeometry` object to a clone of the `targetGeometry` object, generate a new `targetGeometry` object, and reset the tween by setting `pos` back to `0`.

All we need to do now is chain the tween to itself and we have our animation. This is described in the following code:

```
tweenIn.chain(tweenIn);
tweenIn.start();
```

Also, by just changing the image you want to load, you can quickly create interesting effects, as shown in the following screenshot:



# Summary

In this chapter, we focused on the features of CSS3DRenderer. As you've seen, there are lots of interesting things you can do with CSS3DRenderer that will not only look great, but also deliver good performance on a wide range of mobile and desktop devices.

When you work with CSS3DRenderer, you do have to remember that this renderer doesn't support the wide range of materials, geometries, and objects that the other Three.js renderers support. The only objects that CSS3DRenderer supports are CSS3DObject and CSS3DSprite. With CSS3DRenderer, you can render each HTML element. All you have to do is wrap it in a CSS3DObject or CSS3DSprite.

To render geometries with CSS3DRenderer, you can't use the geometries that Three.js uses out of the box. What you can do, as shown in this chapter, is use the information from a geometry's object and use that to render HTML elements in one of the shapes provided by Three.js.

It's also possible to simulate a particle system with CSS3DRenderer. In this case, you should use a CSS3DSprite. This object will always be aligned with the camera.

In the next and final chapter, we'll look at how you can use Three.js together with Blender. You use Blender for animation and modeling, and use Three.js for real-time rendering in the browser.

# 7
# Loading and Animating External Models Using Blender

Three.js works really well for showing and animating models directly in the browser, either using WebGL or, as you've seen in the previous chapter, using CSS3D. With Three.js, you get a large set of standard geometries that you can use in your scene, and as you've seen in *Chapter 5*, *Programmatic Geometries*, it's also easy to create custom geometries from scratch. Three.js, however, isn't a 3D modeling tool. If you want to create more complex geometries, you'll need a different kind of tool. There are many commercial and very expensive 3D modeling tools available, but luckily, there is also a great open source 3D modeling application. This application is called Blender (`http://www.blender.org`). Using Blender, you can easily model complex 3D objects and even assign animations to these models. In this chapter, we'll explain how you can combine the modeling functionality from Blender with the rendering functionality provided by Three.js.

The following topics will be explained in this chapter:

- We'll start by installing Blender and configuring the Three.js plugin so that models and animations can be exported to Three.js.
- Next, we'll explain how to accomplish a simple model in Blender, export it using the Three.js plugin, and finally, load the model and show it in the browser.

- Three.js supports two different kinds of animations: bone-based animations and morph-based animations. These same kinds are also supported in Blender. In this chapter, we'll show you how to set up such an animation in Blender and animate it in your browser through Three.js.
- For both the bone-based and morph-based animation, we'll show you how to play an animation defined in Blender using Three.js.

The first thing we'll do is set up Blender and configure the Three.js plugin.

# Installing Blender and the Three.js plugin

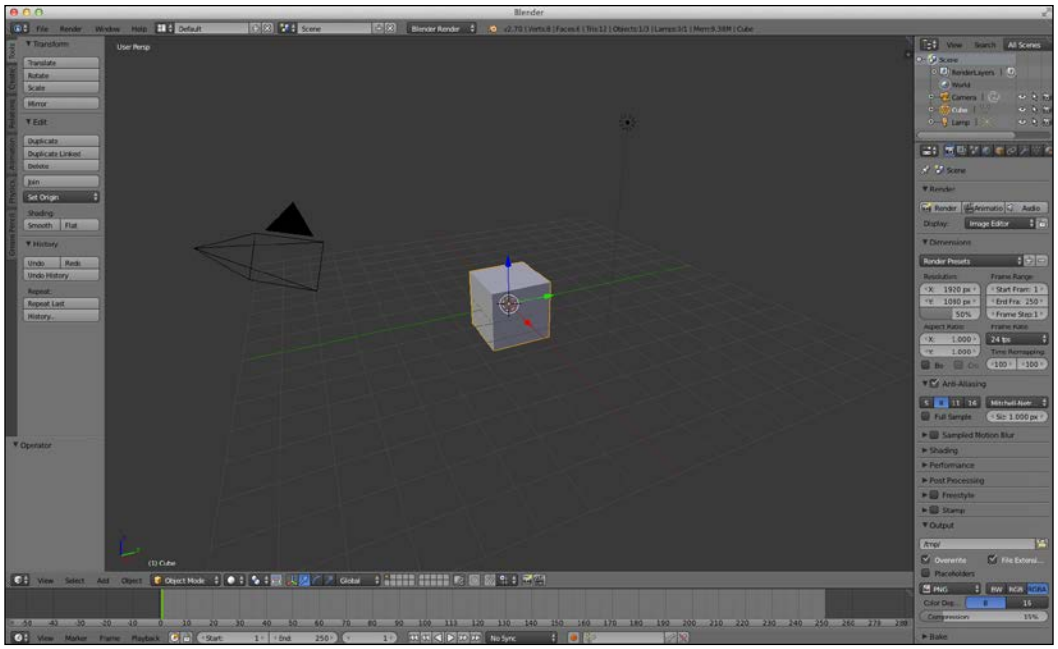
The first thing that we need to do is download Blender. The examples in this chapter were done with Blender 2.70a, but they should also work with newer versions. In this section, we'll perform the following steps:

1. First, we'll show you where you can download Blender and how to install it.
2. Then, we'll show you how to install the Three.js plugin in Blender.
3. Finally, we'll explain how to enable this plugin so that you can use it directly from the Blender menus.

Let's start with the first step and install Blender.

## Downloading and installing Blender

You can download your platform-specific version of Blender from `http://www.blender.org/download/`. This page also lists the installation instructions for your specific platform. Note that if you don't have permissions to install Blender on your local machine, there is also a portable version available that doesn't require an installation. It is available at `http://portableapps.com/apps/graphics_pictures/blender_portable`. After you install Blender, start it up to see whether everything is working. You should be presented with a screen that somewhat resembles the following screenshot:



You'll see a single cube in the middle of an empty scene. Now, we need to install the Three.js plugin, which will allow us to import and export models in the Three.js format. Note that Three.js also supports a large number of other 3D formats, but for best results, you should use its own JavaScript-based format.

# Installing the Three.js plugin

To install the Three.js plugin, we first need to get the latest version of it. We've added the required files to the source code of this book. You can find this plugin, which is a folder named `io_mesh_threejs`, in the `assets/plugins` directory. To install this plugin in Blender, we need to copy the complete `io_mesh_threejs` directory to the location where Blender stores its plugins. Depending on your platform, these are stored at different locations. For Windows, copy the `io_mesh_threejs` directory to the `C:\Users\USERNAME\AppData\Roaming\Blender Foundation\Blender\2.70a\scripts\addons` location. For OS X, depending on where you extracted the zip file, copy the `io_mesh_threejs` directory to the `/location/of/extracted/zip/blender.app/Contents/MacOS/2.6X/scripts/addons` location. Finally, for Linux, copy the `io_mesh_threejs` directory to the `/home/USERNAME/.config/blender/2.70a/scripts/addons` location.

If you've installed it on Ubuntu using the `apt-get` command, you should copy the `io_mesh_three.js` directory to the `/usr/lib/blender/scripts/addons` location.

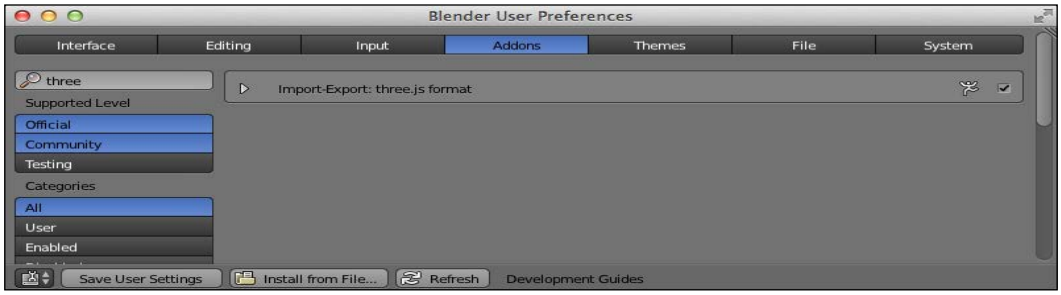
Now, all that is left to do is enable the Three.js plugin from Blender.

# Enabling the Three.js plugin

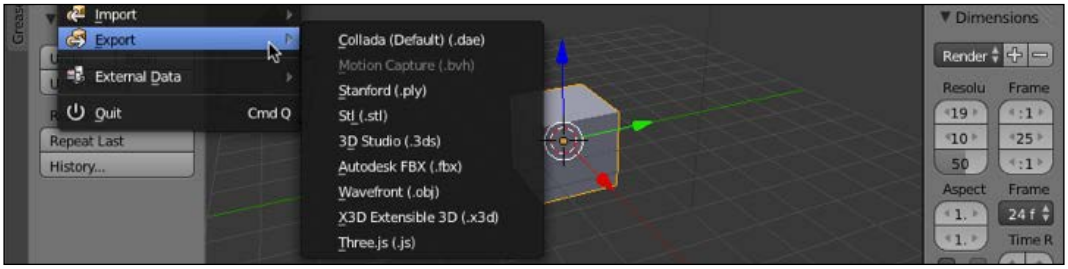
To enable the plugin, restart Blender and open the **Blender User Preferences** window through **File | User Preferences…**. From this window, select the **Addons** tab; this will show all the available plugins, as shown in the following screenshot:


In the search box in the top left-hand side of the window, type `three`. This filters the plugins and if the `io_mesh_threejs` directory is copied to the correct location, we will see a window as shown in the following screenshot:

Here, you can see that the Three.js plugin is found, but as it is gray in color, it isn't enabled yet. You can now enable the plugin by checking the checkbox to the far right, as shown in the following screenshot:




Now, the Three.js plugin is enabled and we can export Blender models to Three.js. Before you go on, be sure to click on the **Save User Settings** button so that Blender stores these changes. To test whether everything is configured and enabled correctly, close this window and open the **Export** menu via the **File** option. It should show a Three.js export option, as shown in the following screenshot:



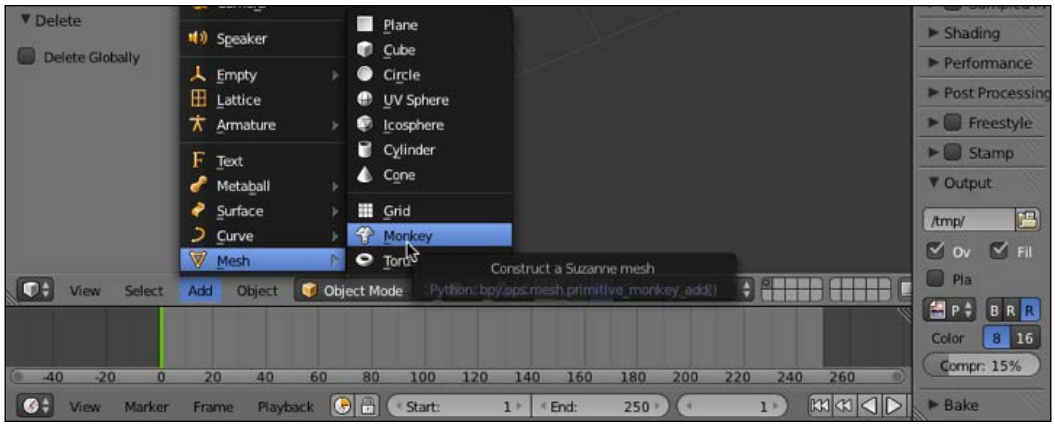

With the export option available, let's look at how we can use this plugin.

# Exporting a model from Blender and showing it in Three.js

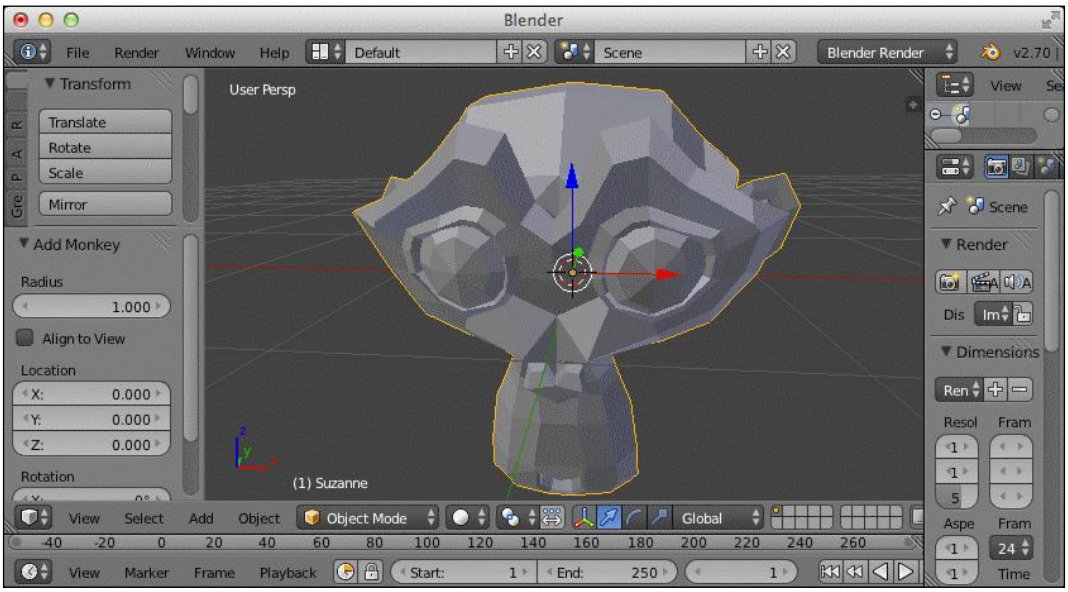
Let's define a model of a 3D geometric shape in Blender, export it using the plugin we just installed, and render it in Three.js. You can also find the Blender model used in this example in the `assets/models` folder from the source code of this book.

# Exporting the model

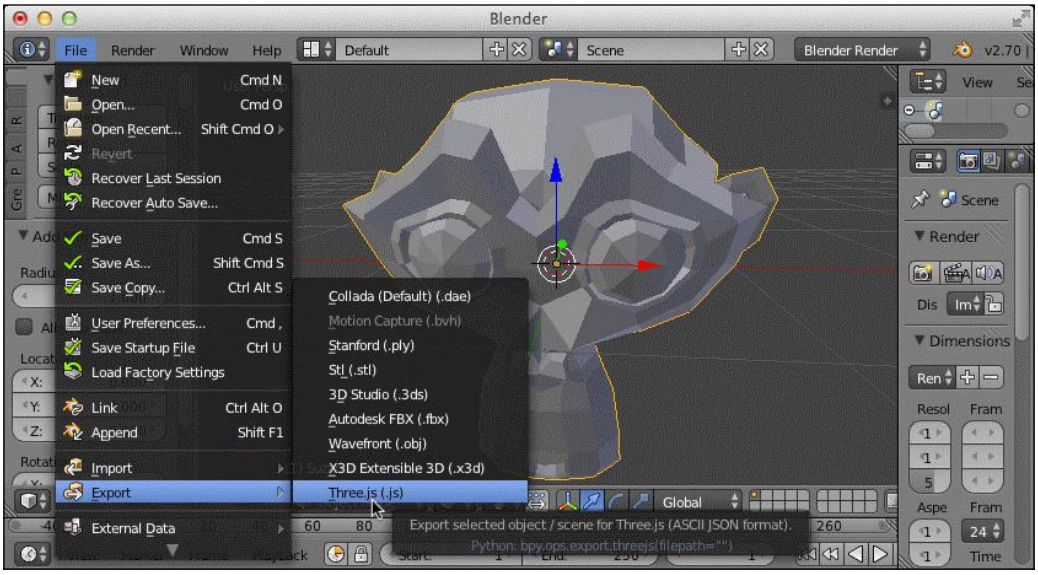
For this first example, we're just going to use one of the standard geometries that are provided by Blender. First off though, let's remove the cube that we see. If you've opened Blender, the cube will be selected, so just delete it by pressing *x* and clicking on **Delete** in the pop up menu. Next, click on **Add** (at the bottom of the 3D window) and select **Monkey** via the **Mesh** option's submenu. This is shown in the following screenshot:




Doing this will create a geometry that looks like a monkey, as shown in the following screenshot:



Now that we've got a model in Blender, we can use the plugin we installed at the beginning of this chapter to export this model to Three.js. To do this, go to the **Export | Three.js** option from the top menu, as shown in the following screenshot:



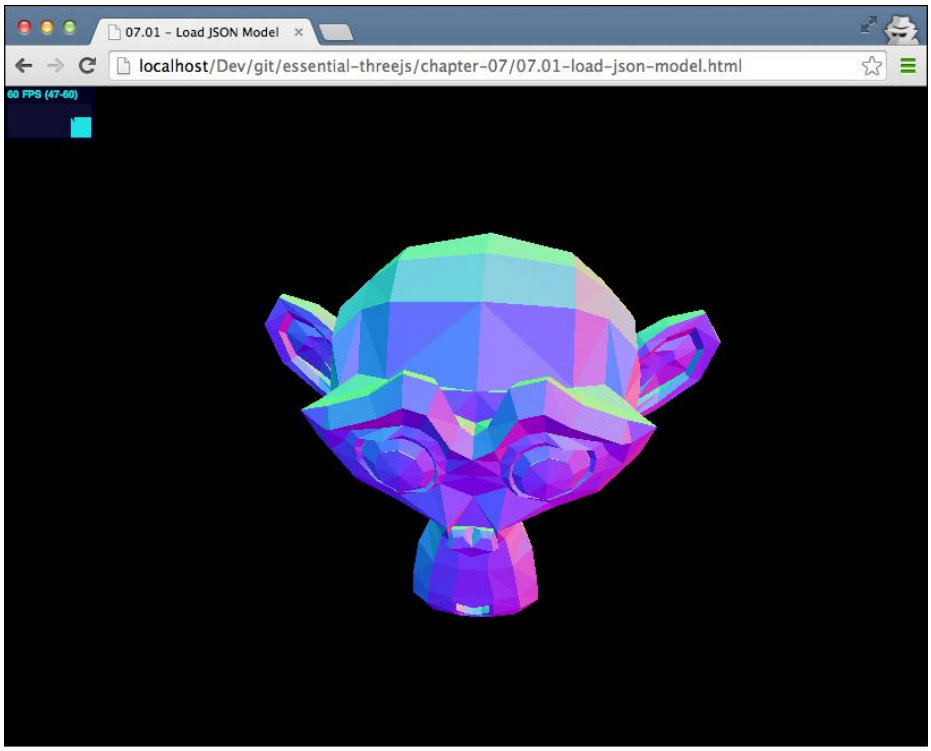

This opens up the export dialog box where you can convert the current model to Three.js. For this example, we will export the monkey to the `assets/models/exported` directory located inside the source code of this book. You can specify additional Three.js properties, but for now, the defaults will suffice. This is shown in the following screenshot:

# Loading the model and showing it in Three.js

Now that we've got our model, all we have to do is load it in Three.js and display it on the screen. Doing this is very easy and only requires the following JavaScript code:

```
function loadModel() {
  var loader = new THREE.JSONLoader();
  loader.load("../assets/models/exported/monkey.js",
    function(model) {
    var material = new THREE.MeshNormalMaterial();

    var mesh = new THREE.Mesh(model, material);
    mesh.translateY(-0.5);
    mesh.scale = new THREE.Vector3(3, 3, 3);

    scene.add(mesh);
  });
}
```

The previous code fragment uses the `THREE.JSONLoader.load()` function to load the model we just exported from Blender. The first argument to the `loader.load()` function specifies the model that we want to load, and the second argument is a callback function that will be called once the model is loaded. For this example, in our callback function, we move the geometry down a bit using the `translateY` function and scaling the model along the *x*, *y*, and *z* axis by setting the `scale` property. We need to specify this callback function as Three.js loads models asynchronously. In the callback function, we create the material we want to use for the model and use the provided model to create a `THREE.Mesh` object, which we will add to the scene. The result is shown in the following screenshot:



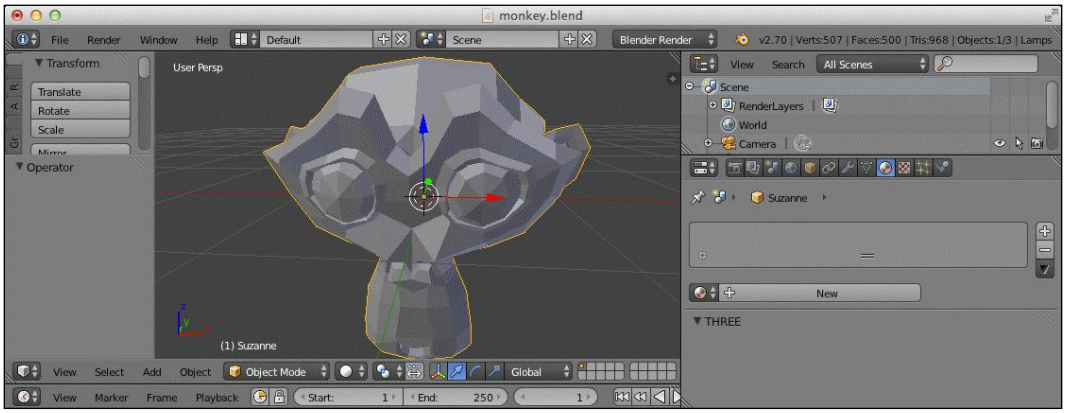

In this example, we defined the model in Blender and the material in Three.js. Blender, of course, also supports materials and textures. In the following section, we'll show you how you can use materials defined in Blender for your own models.

# Using Blender's predefined materials in Three.js

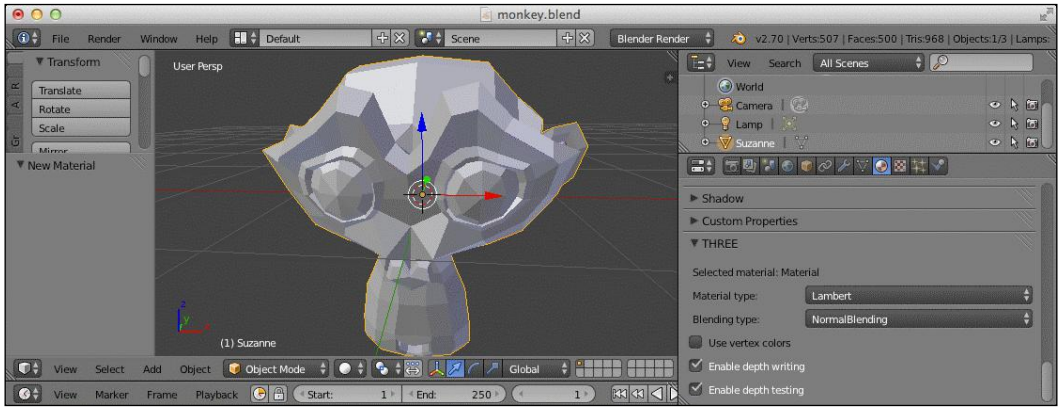
Blender has excellent support for a large range of different materials and provides easy-to-use tools to map and apply textures. Using the Three.js plugin, we can also export the materials used in Blender to Three.js. Let's continue with the monkey we left off in the previous section. The first step we need to take is set up the material.

# Setting up a Blender material

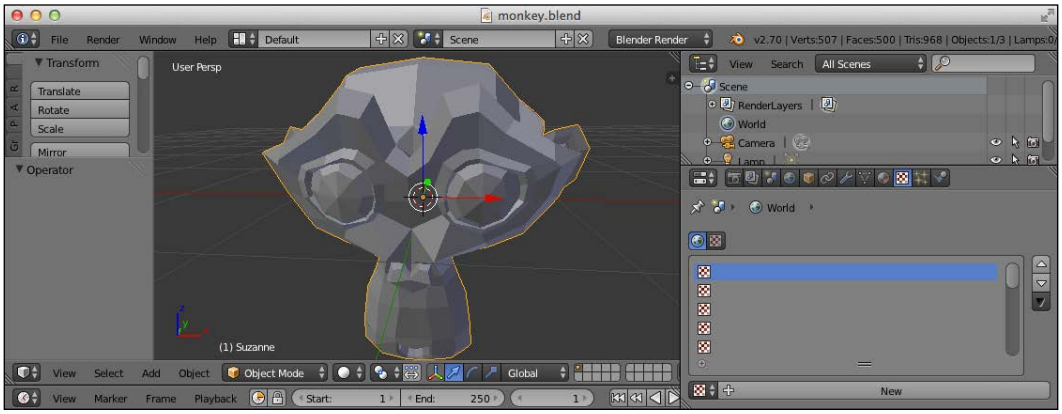
On the right-hand side of the screen, open the material tab, as can be seen in the following screenshot, and click on **New**:



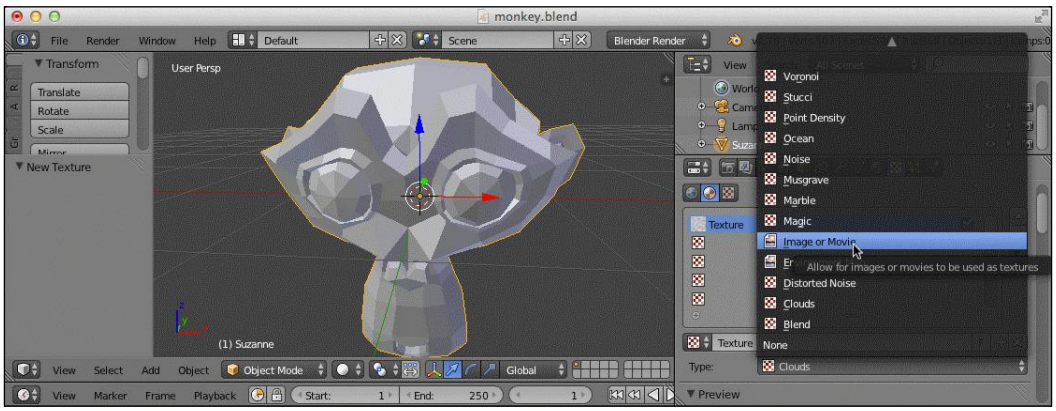

This will create a new material and assign it to our monkey geometry. If you scroll down in the material tab after the material has been created, you can set some Three.js-specific properties, such as the material type to be used and the type of blending you expect. This is shown in the following screenshot:



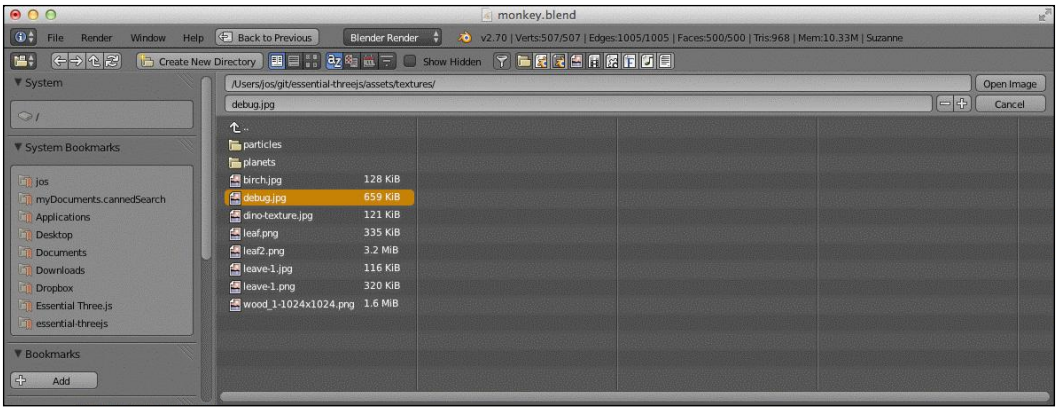
For our example, we'll just keep **Material type** as **Lambert** and **Blending type** as **NormalBlending**. Now, click on the texture tab so we can define the texture that we want to use on our monkey. This is shown in the following screenshot:




In this tab, click on **New**. This will create the texture for this material. Next, click on the **Type** field and select **Image or Movie** as the texture type. This is shown in the following screenshot:



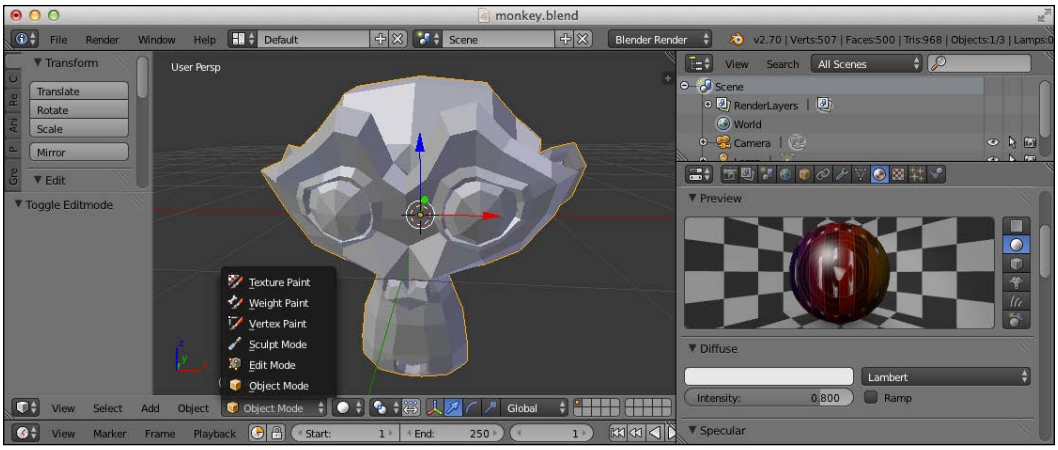
Next, scroll down a bit and click on the **Image** tab. There, click on the **Open** button. This will open a file selection window from where you can select the image that should be applied as a texture. In our example, we've used the `debug.png` image. This is shown in the following screenshot:



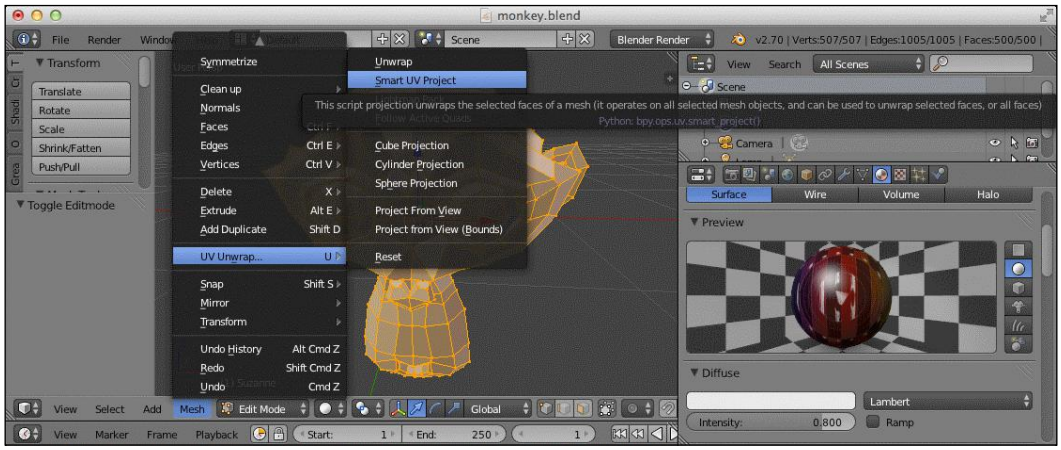

Now that we've set up the material, we need to perform an additional step in Blender before we can use this material in Three.js. As you might remember from *Chapter 5*, *Programmatic Geometries*, when we want to apply a texture to a geometry, we also need to define a UV mapping. This defines how individual faces map to a certain part of the texture.

# Setting up UV mapping in Blender

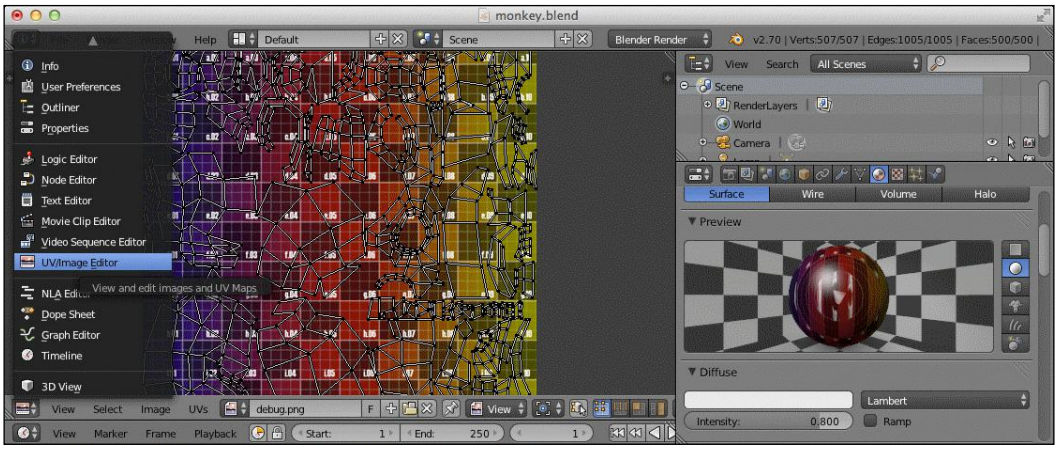
Luckily, we don't have to set up UV mapping by hand in Blender. For this example, we'll just let Blender determine the best way to map our monkey to the texture. We can do this by telling Blender to unwrap our geometry and project it on the texture. To do this, we first have to select the edit mode. You can do this by clicking on the **Object Mode** button at the bottom of the screen or by just hitting the *Tab* key. This is shown in the following screenshot:




From this mode, navigate to **Mesh | UV Unwrap** and click on the **Smart UV Project** option. This is shown in the following screenshot:



In the window that pops up, just click on **OK** to accept the default settings and you're done. To see how Blender unwrapped our geometry, you can open the **UV/Image Editor** view from the editor icon in the bottom left-hand side of the window. This is shown in the following screenshot:



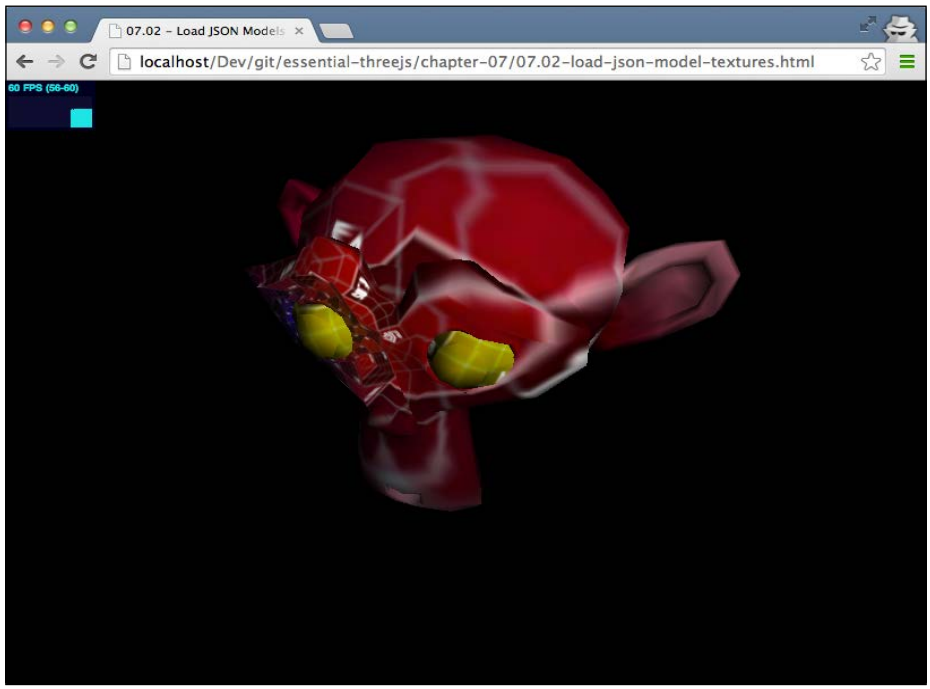

Each black dot you see in this square represents a vertex of the model, and you can see how each face is mapped to the texture. Now that we've set up UV mapping, we can export the model to Three.js and render it.

# Exporting and rendering in Three.js

Exporting this model is done in the same manner as we've done in the previous example. Loading the model isn't that much different, except that we have to specify a couple of additional parameters. Consider the following code:

```
function loadModel() {
  var loader = new THREE.JSONLoader();
  loader.load("../assets/models/exported/monkey-mat.js",
    function(model, material) {

    var mesh = new THREE.Mesh(model, material[0]);
    mesh.scale = new THREE.Vector3(3, 3, 3);
    scene.add(mesh);

  }, "../assets/textures/");
}
```

The first change is in the callback function we specify. Instead of just defining the model argument, we also specify a material argument. The provided material argument contains an array of the materials we specified in Blender. So, to create a mesh using the material from Blender, we just use the `new THREE.Mesh(model, material[0])` function. For this example, we have also defined an image to be used as the texture. We need to provide the `THREE.JSONLoader()` function with information about where it can find the images used by the materials. This is done by providing the location as the final argument to the `loader.load()` function. In this case, we specify `"../assets/textures/"` as the location. Now, when we view this scene in the browser, you will see something like the following screenshot:



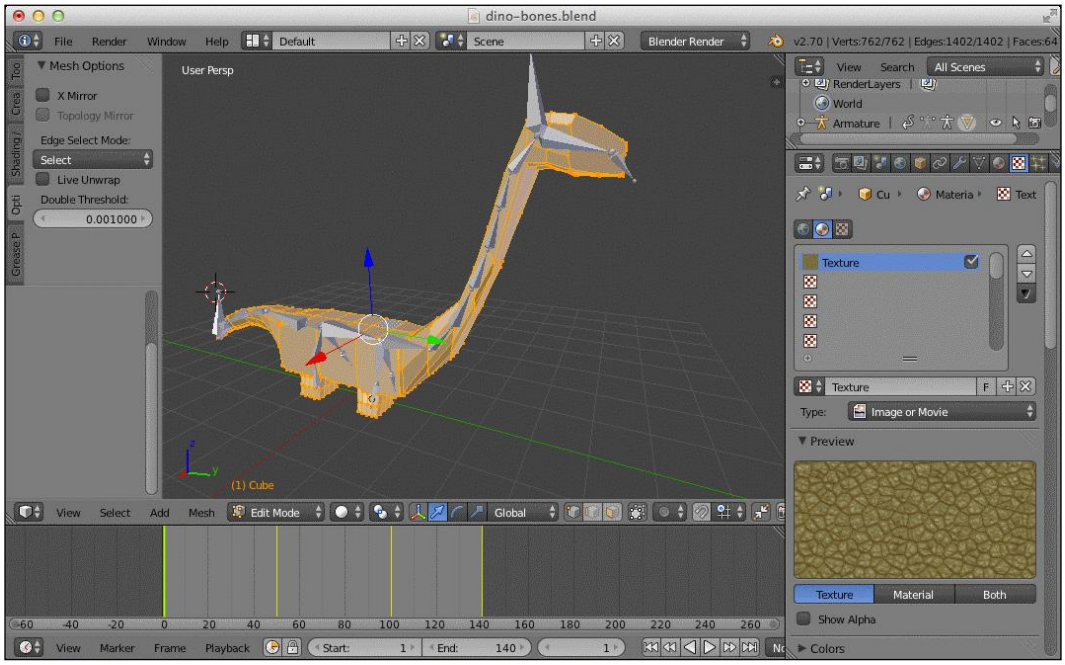

You can see the monkey's head with the texture and material we defined in Blender. That's it for loading and exporting static models. In the next section, we'll show you how to run animations created from Blender.

# Working with skeletal-based animations in Three.js

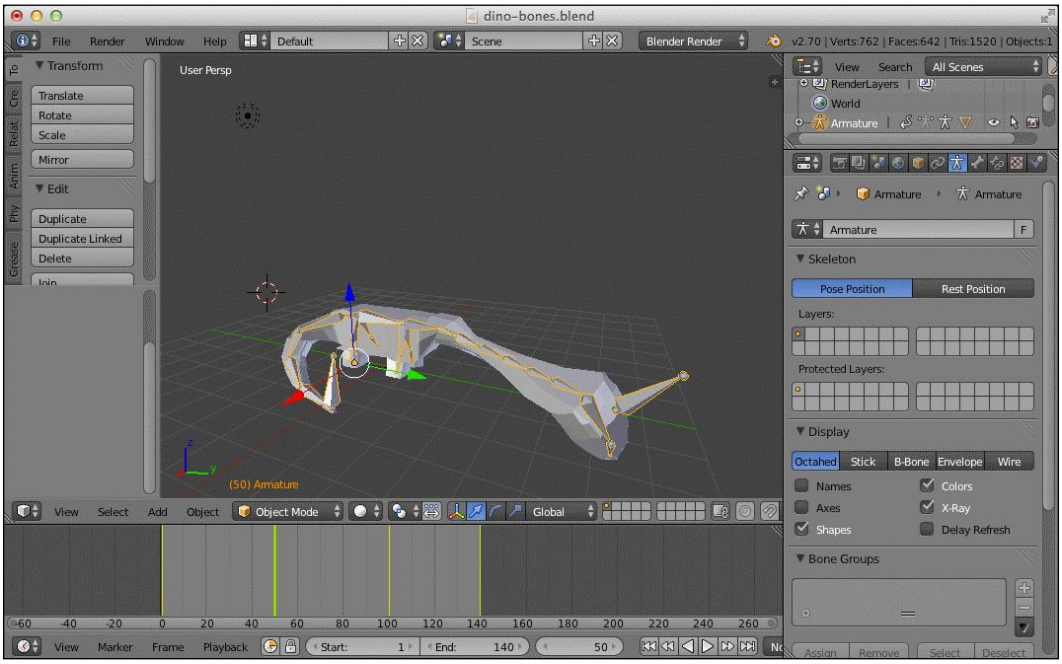
In this section, we're going to look a bit closer at how to show animations defined in Blender from Three.js. We start by looking at skeletal-based animations, where we animate a model by moving its bones around. Setting up the bones and the associated animations in Blender, however, is a bit out of the scope of this book. A good introduction on skeletal animations can be found at the Blender wiki at `http://wiki.blender.org/index.php/Doc:2.6/Manual/Your_First_Animation/2.Animating_the_Gingerbread_Man`, or you can follow some tutorials on YouTube.

# Exploring the model and exporting it to Three.js

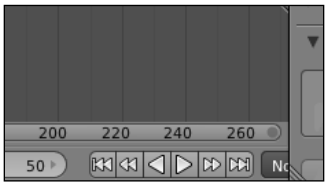
To see the animation and model we're working with, load the `dino-bones.blend` file, which you can find in the `assets/models/blender` directory of the source code. Once you open up this blender file, you'll be shown a very roughly modeled dinosaur. This is shown in the following screenshot:



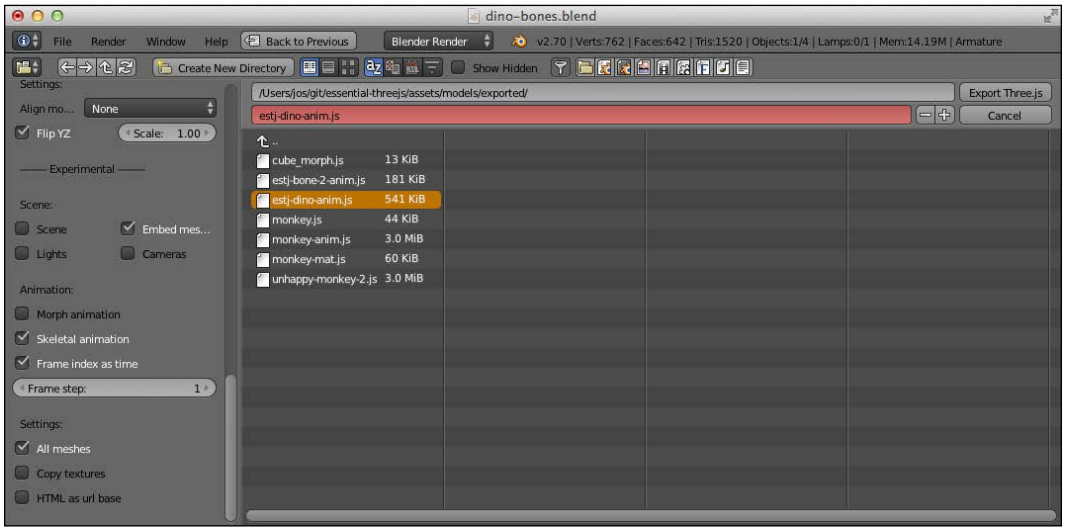
In this screenshot, you can see the shape of the dinosaur and also the skeleton it is made of. For this example, we've connected the model to the bones you can see. The result is that whenever we move the bones (in the pose mode in Blender), the model deforms as well. We've used this to create a number of poses for this dinosaur. One of the poses is shown in the following screenshot:




In this pose, the head of the dinosaur is at the bottom and its tail is curved. By combining various poses and adding each pose as a key for the animation, we can create a skeletal-based animation. You can view our sample animation by clicking on the play button at the bottom of the screen. This is shown in the following screenshot:



Exporting this model together with the animation can be done directly through Blender by using the Three.js plugin we installed. This time, however, we need to make sure that the export is configured correctly. First, make sure the model is in its rest mode (change **Frame step** option to **1**) and then select **Three.js** through **File | Export**. In the window that opens, scroll the right scroll bar down until you see Three.js-specific properties. Make sure that the properties for the animation are set as shown in the following screenshot:




This will make sure that the bones and the animation we defined in Blender are exported and can be used in Three.js. Now, all that is left to do is to load this model in Three.js and run the animation.

# Loading and animating the model in Three.js

Let's look at how the `loadModel` function needs to change to support these animations. Consider the following code:

```
function loadModel() {
  var loader = new THREE.JSONLoader();
  loader.load("../assets/models/exported/estj-dino-anim.js",
    function(model, loadedMat) {
      loadedMat[0].skinning = true;
      THREE.AnimationHandler.add(model.animations[0]);
      animmesh = new
        THREE.SkinnedMesh(model, loadedMat[0]);

      animmesh.translateY(-2);
```

```
        var animation =
          new THREE.Animation(animmesh, "ArmatureAction");
        animation.play();

        scene.add(animmesh);
    }, "../assets/textures/");
}
```

We've added a couple of things to this function. The first thing we do is set the `loadedMat[0].skinning` property to `true`. This makes sure that Three.js knows we're working with a skinned model. If this property is not set to `true`, everything will seem OK, but you'll see no animations. The next step is to inform Three.js about the animations that are available in our model. You can use the `THREE.AnimationHandler.add` function for this purpose. In this example, we only have a single animation defined, so we add this animation (`model.animations[0]`) to the animation handler.

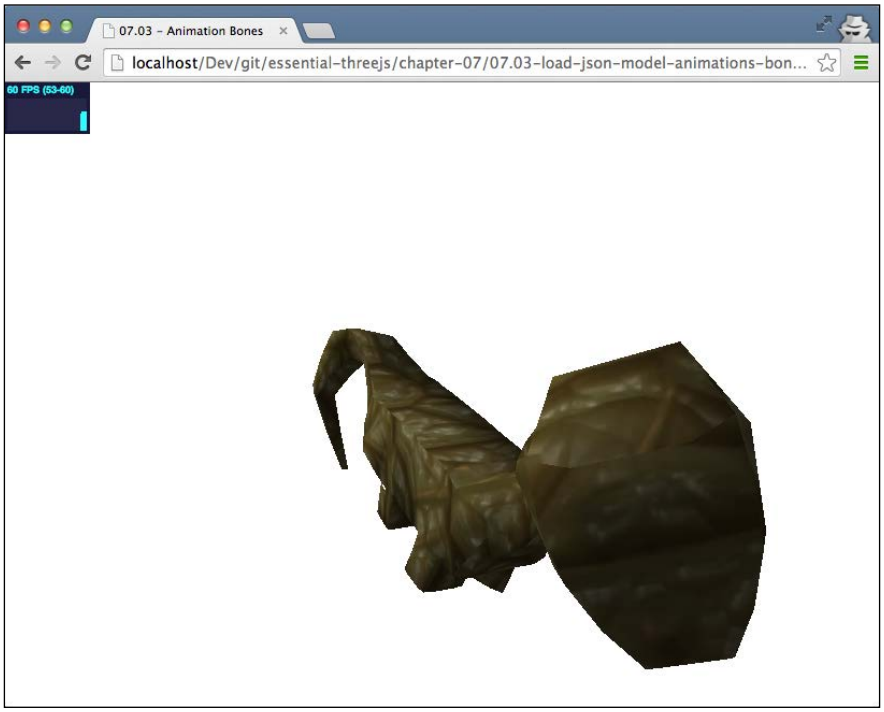
Now, we can just create our mesh. For these kinds of models (also called skinned models), we have to use a special mesh called the `THREE.SkinnedMesh` object. Creating this mesh works in the same way as for a normal `THREE.Mesh` object; you specify the model and the material you want to use. Now, we can set up the animation we want to play and call the `play` function on this object. In this case, our animation is called `ArmatureAction`. This name is used in Blender for a skeletal-based animation. If you want to know the names of all the animations available, the easiest way is to just output the `model.animations` array to the JavaScript console of your browser.

If you run the demo with this setup, you still won't see a dancing dinosaur. For this, we need to make a small change to the `render` function, as shown in the following code:

```
var clock = new THREE.Clock();
function render() {

  ...
  var delta = clock.getDelta();
  if (animmesh) {

    THREE.AnimationHandler.update(delta);
  }
  ...
}
```

We need to define a `THREE.Clock` object that keeps track of the time passed between each rendering. Three.js uses this information to determine how to render our model, using the `THREE.AnimationHandler.update` function, according to the animation we have defined.

Now, when you open this example in your browser (the `07.03-load-json-model-animations-bones.html` file), you'll see our model from Blender following the poses we defined for our animation, as shown in the following screenshot:



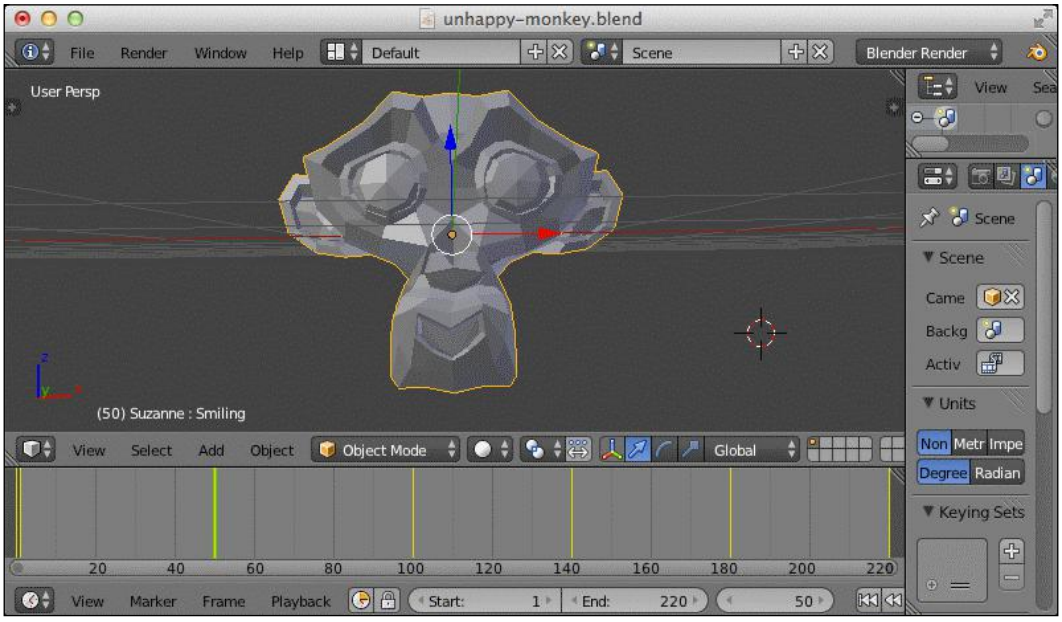

It's worth mentioning that with Three.js, you can also move the bones of the model programmatically, but that's a bit beyond the scope of this book. Besides skeletal-based animations, Three.js also supports another often-used animation method called morph-based animation.

# Working with morph-based animations in Three.js

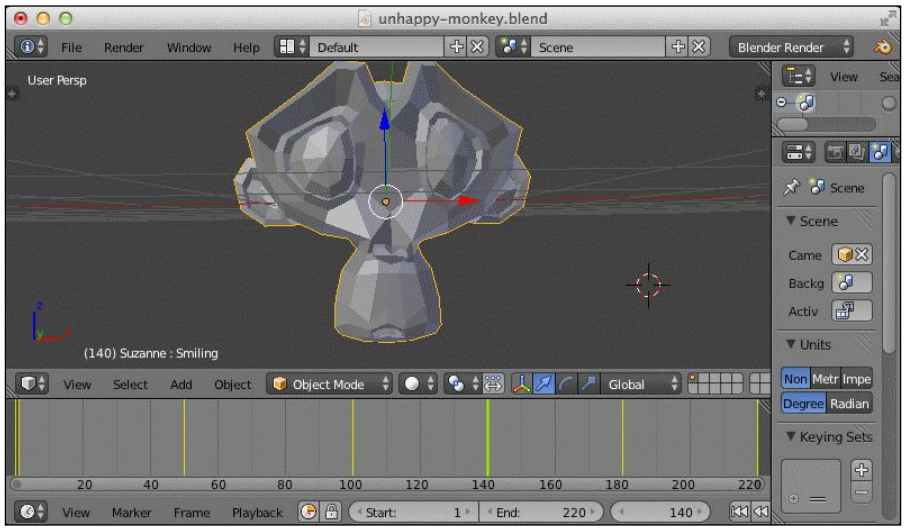
With skeletal-based animations, we move the model around based on the position and rotation of its individual bones. This provides you with a lot of flexibility but becomes very complex for detailed animations such as facial expressions, which aren't bone-based. With morph-based animations, you don't work with bones but simply define the various states of the model (by moving the vertices around), and you let Blender (or Three.js) fill in the missing steps between these model states. Morph-based animations, for example, are often used to model facial expressions.

# Exploring the model and exporting it to Three.js

In this example, we're going to create a morph-based animation where the monkey we saw in the beginning is made to smile at first and appear sad later. Once again, explaining how to set this up in Blender is a bit out of the scope of this book, but the related information can be easily found on YouTube or on the Blender wiki at `http://wiki.blender.org/index.php/Doc:2.6/Manual/Animation/Techs/Shape/Shape_Keys`. To see how it was set up in Blender, we've provided the Blender source code file in the `assets/models/blender` folder. If you load the `unhappy-monkey.blend` file, you will see the animation we've set up. In the following screenshot, you see the monkey smiling:

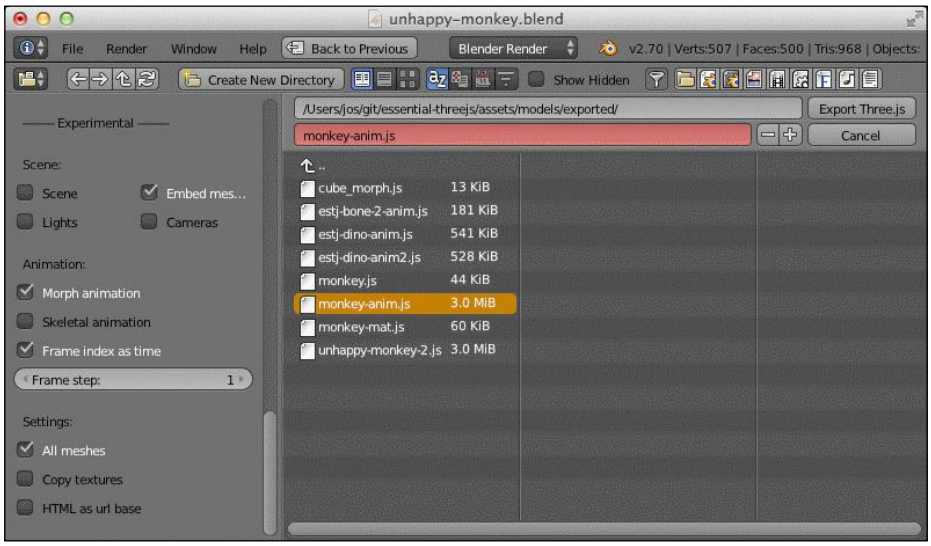


In the next screenshot, the monkey is looking a bit sad:




What you can see here is that we haven't defined a skeleton. What we're doing here is just repositioning the individual vertices to change the facial expression of our model.

To export this model, we once again have to make sure that the exported Three.js is configured correctly. So, click on **Three.js** through **File | Export** and make sure the **Morph animation** box is ticked. This is shown in the following screenshot:




This will result in a Three.js model that we can load from our JavaScript code.

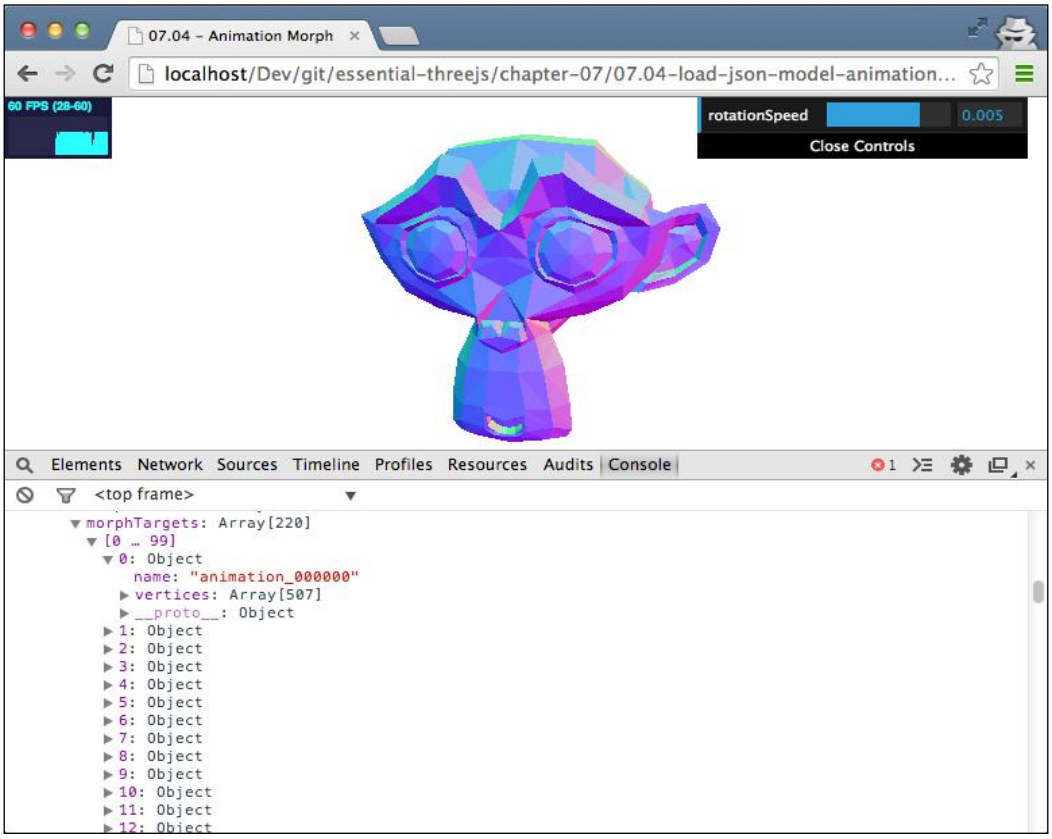
# Loading and animating the model in Three.js

Before we can animate the model, we first have to load the model. Loading the model is pretty much the same as we've seen for a bone-based model. This is described in the following code:

```
function loadModel() {
  var loader = new THREE.JSONLoader();
  loader.load("../assets/models/exported/unhappy-monkey-2.js",
    function(model, loadedMat) {

      var mat = new THREE.MeshNormalMaterial({
        color: 0xff0000
      });
      mat.morphTargets = true;
      mat.shading = THREE.SmoothShading;

      animmesh = new THREE.MorphAnimMesh(model, mat);

      animmesh.parseAnimations();
      animmesh.playAnimation("animation");
      animmesh.duration = 10;

      scene.add(animmesh);

  );
}
```

In this example, we haven't defined a material for our monkey. So, the first thing that we do is create our own material. For a skeletal-based animation, we had to set the `skinning` property of the material to `true`. In this case, we have to set the `morphTargets` property of the material to `true` to tell Three.js that this model uses morph-based animations. Next, we can create our mesh. This time we have to use the special `THREE.MorphAnimMesh` object. The resulting mesh contains all the functions we need to play our animation. First, we need to determine the available animations. We do this by calling the `parseAnimations()` function on the mesh we just created.

This will look through the morphs defined in the model and try and determine the animation names. It does this by looking at the `morphTargets` property, which is shown in the following screenshot:

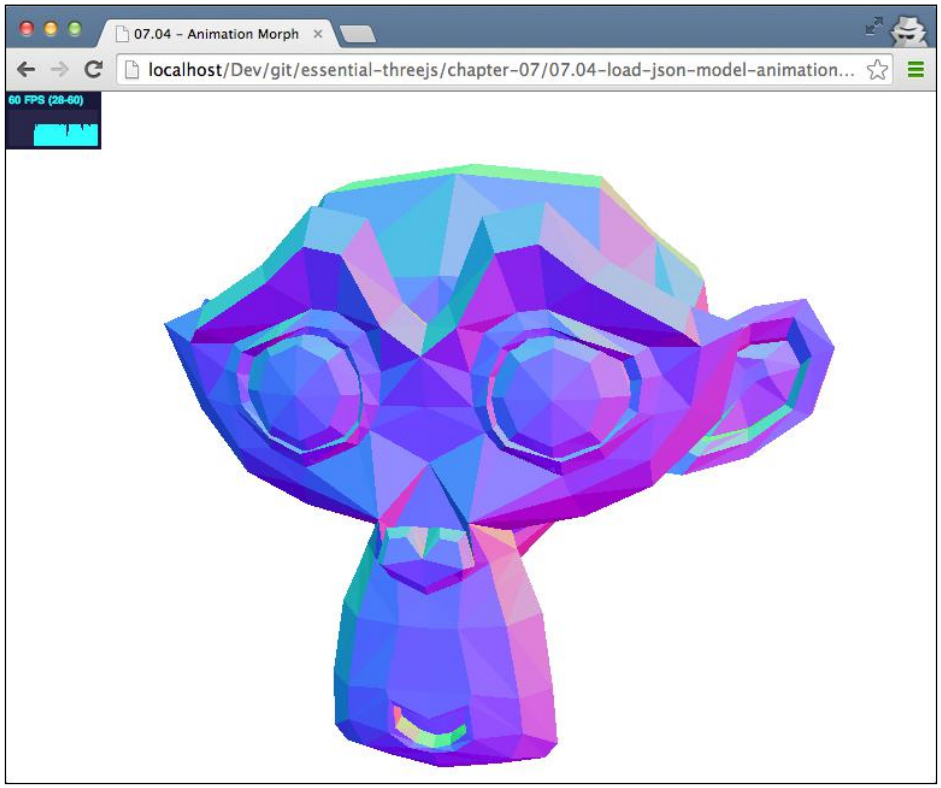



The result from the `parseAnimations()` function will give us an animation called `animation`. Next, we tell the mesh to play this animation, using the `playAnimation()` function. The last step that we need to do is set up the duration of the animation. With the duration, you can control the playback speed.

For a morph-based animation, we also need to make a small change to the `render` function and add the following lines of code:

```
var delta = clock.getDelta();
if (animmesh) {
  animmesh.updateAnimation(delta);
}
```

The reason we need to do this is the same as for a bone-based animation. Three.js needs to know how much time has passed since the last render. It uses this information to determine how the model looks for the next rendering. When you combine all of this, you get a smiling and sad-looking monkey (the `07.04-load-json-model-animations-morph.html` file). This is shown in the following screenshot:




The easiest way to get used to the different kinds of animations is to play around with Blender-based examples and see how changes in Blender affect the rendering in Three.js. As you've seen, being able to import external models is a very powerful feature of Three.js.

# Summary

In this chapter, we focused on how you can integrate Three.js with Blender. We've only shown you the tip of the iceberg, as Blender is a very powerful and sometimes complex 3D modeling application.

As you've seen in this book so far, Three.js is a really powerful way of rendering 3D graphics in your browser, but it lacks in modeling functionality. Blender is a great tool to use in combination with Three.js. With Blender, you can model your geometries and visualize them in real time in a browser using Three.js. Blender also provides a great way to define materials and apply textures; additionally, if you're struggling to define a UV map for your model, using the functionality that Blender provides could be of great help.

Besides creating models, you can also use Blender to create animations, which you can run using Three.js. Three.js supports two different kinds of animations: skeletal-based animations and morph-based animations. With skeletal-based animations, you create a skeleton that defines how the object will move around. This is great for creating characters and defining natural movements. For more detailed and fine-grained animations, such as facial expressions, morph-based animations come in handy. With morph-based animations, you define various positions of all the vertices in a model and slowly move (morph) from one state to the other. To use a skeletal-based animation in Three.js, you have to use the `THREE.SkinnedMesh` object, and if you want to use morph-based animations, you have to use the `THREE.MorphAnimMesh` object. If you've set up your animation and don't see any errors in the console and still the animation isn't running, remember to check whether you've set the `skinning` property or the `morphTargets` property of the material you're using to `true`. Only then will the animation run.

This is the end of the final chapter. In this book, we've tried to show you the most important parts of Three.js to give you a basic understanding of how you can use this great 3D library to create 3D visualizations and render beautiful models.

You've learned about textures, lighting, materials, models, particle systems, and much more. Still, however, there is a fairly large part of Three.js that we haven't covered in this book. There are, for instance, more kinds of lights available than the ones we've seen, a large number of out-of-the-box geometries provided by Three.js that you can use, advanced post-processing filters provided by Three.js that you can apply to your scene, and even more materials that you can use to style your models.

If you want to know more about these features, there are a couple of places where you can look. Packt Publishing provides a comprehensive Three.js reference book, *Learning Three.js: The JavaScript 3D Library for WebGL*, *Jos Dirksen*, which provides a detailed reference of all the Three.js features; also, you can of course have a look at the Three.js website, which also provides lots of information and examples.

I hope you liked the information and examples from this book. Feel free to use and expand on these examples and create beautiful 3D visualizations yourself!

# Index

## Symbols



## A



## B



## C

## D



## E



## F



## G

## H



## I



## J



## K



## L



## M

## N



## O



## P

## R



## S



## T

## U



## V



## W


## Thank you for buying
# Three.js Essentials

## About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

## About Packt Open Source

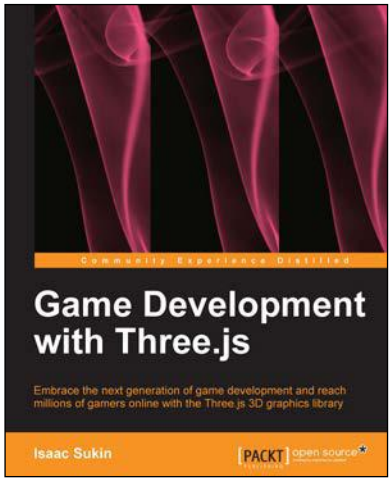
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

## Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



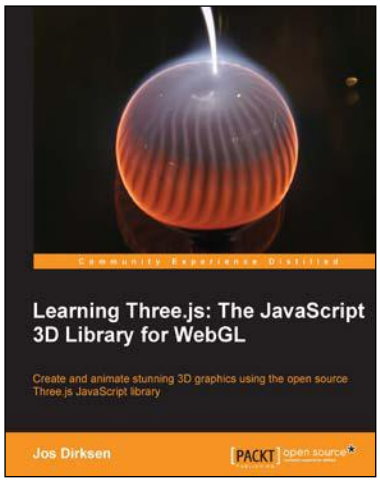

## Game Development with Three.js

ISBN: 978-1-78216-853-9 Paperback: 118 pages

Embrace the next generation of game development and reach millions of gamers online with the Three.js 3D graphics library

1. Develop immersive 3D games that anyone can play on the Internet.
2. Learn Three.js from a gaming perspective, including everything you need to build beautiful and high-performance worlds.
3. A step-by-step guide filled with game-focused examples and tips.

---



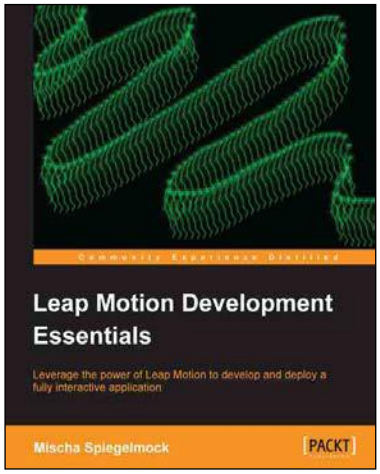

## Learning Three.js: The JavaScript 3D Library for WebGL

ISBN: 978-1-78216-628-3 Paperback: 402 pages

Create and animate stunning 3D graphics using the open source Three.js JavaScript library

1. Create and animate beautiful 3D graphics directly in the browser using JavaScript without the need to learn WebGL.
2. Learn how to enhance your 3D graphics with light sources, shadows, and advanced materials and textures.
3. Each subject is explained using extensive examples that you can directly use and adapt for your own purposes.

[PACKT] open source
community experience distilled
PUBLISHING



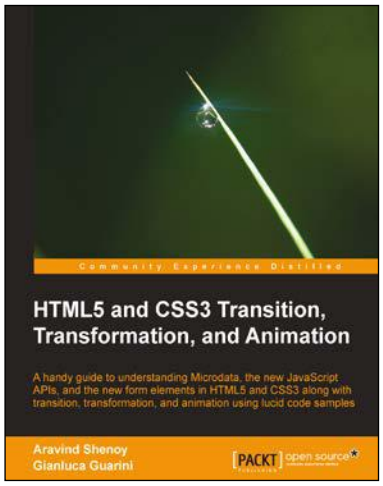

## Leap Motion Development Essentials

ISBN: 978-1-84969-772-9 Paperback: 106 pages

Leverage the power of Leap Motion to develop and deploy a fully interactive application

1. Comprehensive and thorough coverage of many SDK features.
2. Intelligent usage of gesture interfaces.
3. In-depth, functional examples of API usage explained in detail.

---




## HTML5 and CSS3 Transition, Transformation, and Animation

ISBN: 978-1-84951-994-6 Paperback: 136 pages

A handy guide to understanding Microdata, the new JavaScript APIs, and the new form elements in HTML5 and CSS3 along with transition, transformation, and animation using lucid code samples

1. Discover the semantics of HTML5 and Microdata.
2. Understand the concept of the CSS3 Flexible Box model.
3. Explore the main features of HTML5 such as canvas, offline web application, geolocation, audio and video elements, and web storage.
4. Master the tools and utilities in HTML5 and CSS3.