The Pragmatic Programmers

3D Game Programming for Kids

Second Edition

Chris Strom

edited by Adaobi Obi Tulton

3D Game Programming for Kids, Second Edition

Create Interactive Worlds with JavaScript

by Chris Strom

Version: P1.0 (July 2018)

About the Pragmatic Bookshelf

The Pragmatic Bookshelf is an agile publishing company. We're here because we want to improve the lives of developers. We do this by creating timely, practical titles, written by programmers for programmers.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at http://pragprog.com.

Our ebooks do not contain any Digital Restrictions Management, and have always been DRM-free. We pioneered the beta book concept, where you can purchase and read a book while it's still being written, and provide feedback to the author to help make a better book for everyone. Free resources for all purchasers include source code downloads (if applicable), errata and discussion forums, all available on the book's home page at pragprog.com. We're here to make your life easier.

New Book Announcements

Want to keep up on our latest titles and announcements, and occasional special offers? Just create an account on pragprog.com (an email address and a password is all it takes) and select the checkbox to receive newsletters. You can also follow us on twitter as @pragprog.

About Ebook Formats

If you buy directly from pragprog.com, you get ebooks in all available formats for one price. You can synch your ebooks amongst all your devices (including iPhone/iPad, Android, laptops, etc.) via Dropbox. You get free updates for the life of the edition. And, of course, you can always come back and re-download your books when needed. Ebooks bought from the Amazon Kindle store are subject to Amazon's polices. Limitations in Amazon's file format may cause ebooks to display differently on different devices. For more information, please see our FAQ at pragprog.com/frequently-asked-questions/ebooks. To learn more about this book and access the free resources, go to https://pragprog.com/book/csjava2, the book's homepage.

Thanks for your continued support,

Andy Hunt
The Pragmatic Programmers

The team that produced this book includes: Andy Hunt (Publisher), Janet Furlow (VP of Operations), Brian MacDonald (Managing Editor), Jacquelyn Carter (Supervising Editor), Adaobi Obi Tulton (Development Editor), Paula Robertson (Copy Editor), Potomac Indexing, LLC (Indexing), Gilson Graphics (Layout)

For customer support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

For Elsa and all the princesses who are going to change the world.

Table of Contents

- Three.js
- Physijs
- Controls
- Noise
- Scoreboard.js
- Shader Particle Engine
- Sounds.js
- Tween.js

Bibliography

Early praise for *3D Game Programming for Kids, Second Edition*

This book helped me expand my programming knowledge and introduced me to 3D gaming concepts, and it was engaging at the same time.

→ Keeley L., age 13

I cracked this book open and had flashbacks to laboriously typing out game programs for the Commodore 64 (Zuider Zee forever!) and was excited at the prospect of sharing that sort of experience with my own kid. This let me dive into JavaScript with the reckless abandon of my long-lost youth and immediately have neat things to show. Finding books that I can learn something from is not terribly hard, but finding a book that I can learn from along with my kid is invaluable.

→ Ron Donoghue
Co-Founder, Evil Hat Productions

I am 11 years old and have been homeschooled for my entire life. I recommend this book as a homeschool programming course and for kids who enjoy programmming. This book is good for experienced programmers and novice programmers who want to do 3D programming right away.

→ Bryson S., age 11

This is a great hands-on book for a kid or even someone with some programming chops, who is interested in making small games. It teaches you how to get started with 3D programming, and it’s really neat that it shows how you can build a decent game in just a little bit of time.

→ Nick McGinness
Software Engineer, Direct Supply

I think this book would teach kids a lot about programming 3D objects. I learned about some new math and programming.

→ Owen, age 10

Chris Strom teaches kids 3D game programming with simple yet powerful explanations and examples. But if you're an adult, you can learn from this book, too. I did!

→ Ron Hale-Evans
author of *Mind Performance Hacks* and *Mindhacker*

This is a bright and breezy trip through basic game coding in 3D, with many useful and practical tips on how to approach programming. I'm confident my kids will learn a lot from it.

→ Paul Callaghan
Former educator, now web developer, and father of three boys

I had a lot of fun with this book. I like how it always hints at what's to come so I would really look forward to the upcoming projects. The author also cracked jokes that kids would understand. I felt as if the author was talking to me in real life, which made the book more enjoyable and made the projects easy to follow. I would recommend this book to anyone who is looking for a fun and easy way to learn code.

→ Cedric H., age 13

Acknowledgments

I am nothing without the strength of my amazing wife, Robin, who helps in ways innumerable. She was the sole editor for the early versions of the book. Despite a full load of her own, she still reads every chapter and gives invaluable notes and suggestions. She helped to run the kid hackathons (OK, she runs them) that aided in development of this book. And oh, yeah—she's an awesome wife and mother.

Also a big thanks to my kids for serving as the primary guinea pigs for this book. Readers can thank them for forcing me to remove the boring stuff and making the fun and interesting stuff more so. Thanks, kiddos.

And, of course, huge thanks to my technical reviewers. It's a tough task to review a book from a kid's perspective, but my reviewers were more than up to the task. In no particular order, they are Ana B., Doug C., Bryson S., Cedric H., Keeley L., Paul Callaghan, Rob Donoghue, Kevin Gisi, Ron Hale-Evans, Brian Hochgurtel, Brian Hogan, Chaim Krause, Nick McGinness, James Sterrett, and Jeremy Sydik.

This book would not exist without the great work of Ricardo Cabello Miguel, affectionately known as "Mr.doob." Ricardo is the primary programmer behind Three.js, the 3D JavaScript library that we use in this book. He also wrote the original implementation of the 3DE Code Editor that we use. This book would be significantly less without his amazing talents. Thanks also to Chandler Prall for his work on the Physijs physics engine, of which we make extensive use.

Last, but not least, many thanks to the folks at The Pragmatic Programmers for believing in the book and helping me realize its full potential. Special thanks to

my editor, Adaobi. A second edition is no easy task, but she kept me focused and helped hone the narrative in ways that were unexpectedly rewarding.

Introduction

Welcome to the world of programming!

I won't lie; it can be a frustrating world (it makes me cry at least once a week). But it's totally worth the pain. You get to make this world do whatever you want. You can share your world with others. You can build things that really make a difference.

This book that you hold in your hands is a great way to get started with programming. Why? Because this book believes the best way to learn programming is by *playing*. Oh, there are a few chapters that describe fundamentals, but we only get to those after having fun. So we jump right into some pretty cool 3D animation in Chapter 1.

This is going to be a blast!

How I Learned to Program (and Why That Matters to You)

When I was a kid, I copied computer game programs out of books. This was a long time ago, so I bought books with nothing but computer code, and typed it into my computer. It took a while, and I had no idea what I was doing at first.

Eventually, I started to recognize things. And then I started to change some of the programs—little things at first—to see what happened. Then I started making bigger changes. Eventually I got pretty good at it and could even write my own programs.

I hope that this book will let you do the same, but with one important difference: I'll explain what's going on so you won't have to guess quite as much.

How YOU Can Learn to Program

Everybody is different. Everybody learns differently.

Because of that, there are at least three different ways that you can learn from this book:

1. Play with cool stuff, then read an occasional chapter on fundamentals.
2. Learn the basics, then make some cool stuff with what you know.
3. Just type the code (like I did when I was a kid).

You choose which works best for you!

If you want to play first (the first option), start with Chapter 1 and then go in order through the rest of the book. You'll mostly work on games and simulations in project chapters—chapters whose titles start with the word "Project"—followed by occasional chapters on fundamental skills. If you're not sure which option is best, choose this one. This is the way I wish I could have learned.

If you're the kind of person who likes to understand fundamentals before building bigger things (the second option), then read the "essentials" chapters first—any chapter without "Project" in the title. There's still plenty of coding in them and 3D fun in some of them. Compared to other programming languages, the core of JavaScript is fairly small. You can learn 80–90 percent of JavaScript's core just by reading those chapters. Being able to use it well is another thing—which is where the "Project" chapters come in!

If you just want to code (the third option), then flip to Appendix 1, *Project Code*. All the code from all the games is there. If you get stuck with some of the coding, then flip to the chapter that the code came from for a deeper explanation. This is how I learned, and I turned out mostly OK!

No matter which option is best for you, there's one rule: *always type the code in by hand*. This is much slower than copying and pasting. You're much more likely to make mistakes typing code in yourself.

Going slow and making mistakes is the point!

Typing code by hand makes you think more about what you're typing. You might believe that you can learn just as well by reading and then copying and pasting, but this is 100 percent not true. Even if you've been programming for 50 years, you should never copy and paste code. Typing gives you time to think and to understand the code you're adding. That is *far* more important than getting it done quickly.

Making mistakes is part of programming. Being able to fix broken code is just as important as being able to program. So make mistakes. It will be hard and frustrating. And it will be worth it.

This means that we've reached the first tip in the book! Especially important tips and guidelines are highlighted throughout the book. Pay attention to them because they can really make a difference as you explore this world.

	Always Type Code / Never Copy and Paste!
	Good programmers always type code by hand—they never copy and paste code. Writing good computer programs is much more important than writing them quickly. Good programming means thinking about the code that you create. It means understanding code. So type it out!

One more thing: if you get stuck, you can always get help!

Getting Help

Every programmer needs help. If you've never programmed in your life, you will need help. If you've been programming for 50 years, you'll still need help.

The two rules for asking for help:

1. Try to figure it out yourself at first.
2. Don't be afraid to ask for help.

Knowing a programming language really well is *not* the most important skill a programmer can have. The most important skill is problem-solving. Knowing everything about a computer language does not mean that you'll create 100-percent problem-free code. It helps, but problems still happen. A lot. So try to figure out problems on your own. Even if you can't solve a problem, you're still improving your problem-solving skills.

Speaking of problem-solving skills, Chapter 2, *Debugging: Fixing Code When Things Go Wrong* is the place to get started. That chapter describes common mistakes, how to solve problems using the code editor and the browser, and how to recover when things go really, really wrong.

	Don't Skip Chapter 2!
	Problem-solving is crazy important, so be sure to read Chapter 2. No matter how you learn, the debugging skills described in Chapter 2 are *must reading*. It might seem like you can skip that stuff, but skipping that chapter is like playing sports without pads or driving without a seat belt. Things might seem fine without the safety of debugging skills. Right up until the point that something goes really wrong.

It is 100 percent OK if you can’t figure out a problem on your own. I am happy to help. This book has its own web page where you can find all the code for this book, as well as the book’s community forum.[1] If you get stuck, post a question to the forum. I almost always answer within 24 hours. Just be sure to tell me how you’ve already tried to solve the problem—otherwise that’s going to be my first question to you.

But please, I want you to do well. Ask questions. Use the forum. Let’s code!

What You Need for This Book

You'll need the Google Chrome web browser and a relatively new computer. That's it!

Not all web browsers can generate the cool 3D-gaming objects that we'll build in this book. To get the most out of the book, you should install the Google Chrome web browser on your computer.[2] Other web browsers will work, but some of the exercises in this book rely on features available only in Google Chrome.

Any modern computer (from the past five years) with Google Chrome installed will be sufficient. To be sure, you can test your computer's capabilities by visiting the Get WebGL site.[3] If that site doesn't tell you that your browser supports WebGL, then you may have to try another computer.

What Is JavaScript?

There are many, many programming languages. Some programmers enjoy arguing over which is the *best*, but the truth is that all languages offer unique and worthwhile things.

In this book we use the JavaScript programming language. We program in JavaScript because it’s the language of the web. It’s the only programming language that all web browsers understand. If you can program in JavaScript, not only can you make the kinds of games that you’ll learn in this book, but you can also program just about every website there is.

What's New in the Second Edition?

This is the second edition of *3D Game Programming for Kids*. The first edition was *awesome*. I've been told that I'm biased, but I don't see it. I'm pretty sure the first edition really was close to perfect.

Right, Chris, if it was perfect, why make a second edition?

Well, first, a lot has happened since the first edition of the book. The programming world is always changing. There's always new stuff coming out. Most of that new stuff won't help us learn, but every now and then, some of it really makes a difference. Three years after the first edition came out, there was enough helpful new stuff that I believed I could make the second edition even better than the first.

The other reason for a second edition is that I've changed a lot since the first edition. As a programmer you can never stop learning. I've been programming for 15 years and am always working to learn as many new things as I can. I've also worked hard to become a better programmer, teacher, and writer. This learning helps me to write even better books!

This is a very different book than the first edition. I removed a few chapters and added some completely new ones. All of the remaining chapters and code have been significantly rewritten. Many of the changes add cool new features (flying controls in Chapter 5, *Functions: Use and Use Again* or fire special effects). The new features are fun, but they're not the main reason for the changes.

The changes are for you. They make the book more fun. They make learning easier. They better help you become a programmer.

So what's new? Pretty much everything. Except for the parts that really were already perfect.

What This Book Is Not

Just to be clear: *We're not going to become experts in JavaScript.*

And…

We're not going to become experts in 3D Game Programming.

We'll cover a ton in this book, but we won't be ready to create the next Facebook when we're done. And we won't be able to recreate MarioKart. Both of those require hundreds of programmers working for hundreds of hours—often with more advanced skills than we will cover in this book.

But we will learn the most important parts of JavaScript. And we'll learn lots of 3D skills. We can do some amazing things with these skills. And we'll be ready to get started on even bigger, even more amazing things.

Let’s Get Started!

Enough introduction—let’s jump into programming!

Footnotes

[1] https://talk.code3Dgames.com/

[2] https://www.google.com/chrome/

[3] http://get.webgl.org/

When you're done with this chapter, you will

- *Be able to write code!*
- *Know how to make 3D shapes*
- *Be able to program simple JavaScript*
- *Make your first animation*

Chapter 1

Project: Creating Simple Shapes

In this chapter, we jump right into writing computer code. There will be plenty of time to describe JavaScript—the programming language we're using. There's also plenty of time to talk about what a programming language is. For now, we program—we write code—without worrying too much about specifics.

The idea is to get familiar with typing in code and to get a flavor of what it's like to program. Believe it or not, just by doing that, you're going to learn a *crazy* amount about programming. You'll take your first steps into building 3D worlds. And you'll even start your first animation.

The most important thing that you're going to do in this chapter is to play. The best programmers play with their code. They experiment. They change code to see what happens. They tweak things to push boundaries. They break things. Really!

So let's get right down to breaking—and creating—together.

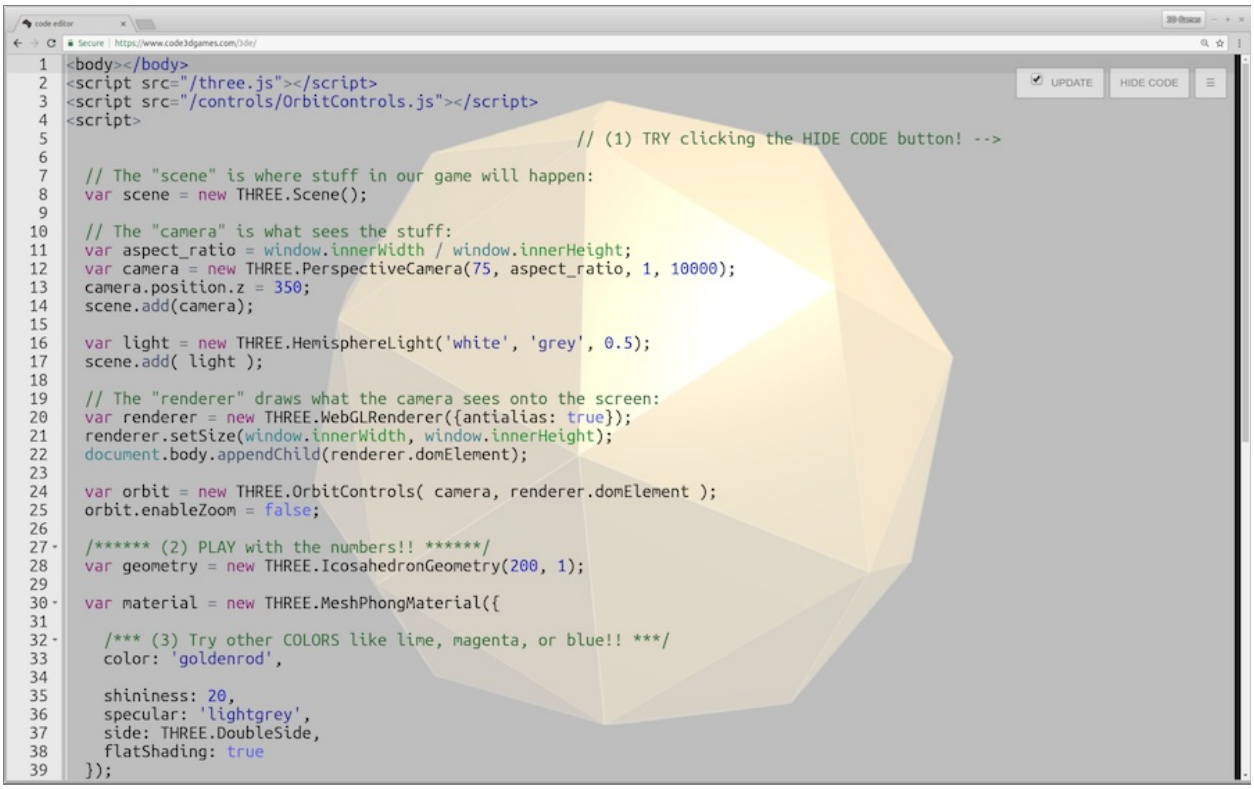
Programming with the 3DE Code Editor

In this book, we use the 3DE Code Editor, or 3DE for short, to do our programming. Some of the many important features of 3DE are:

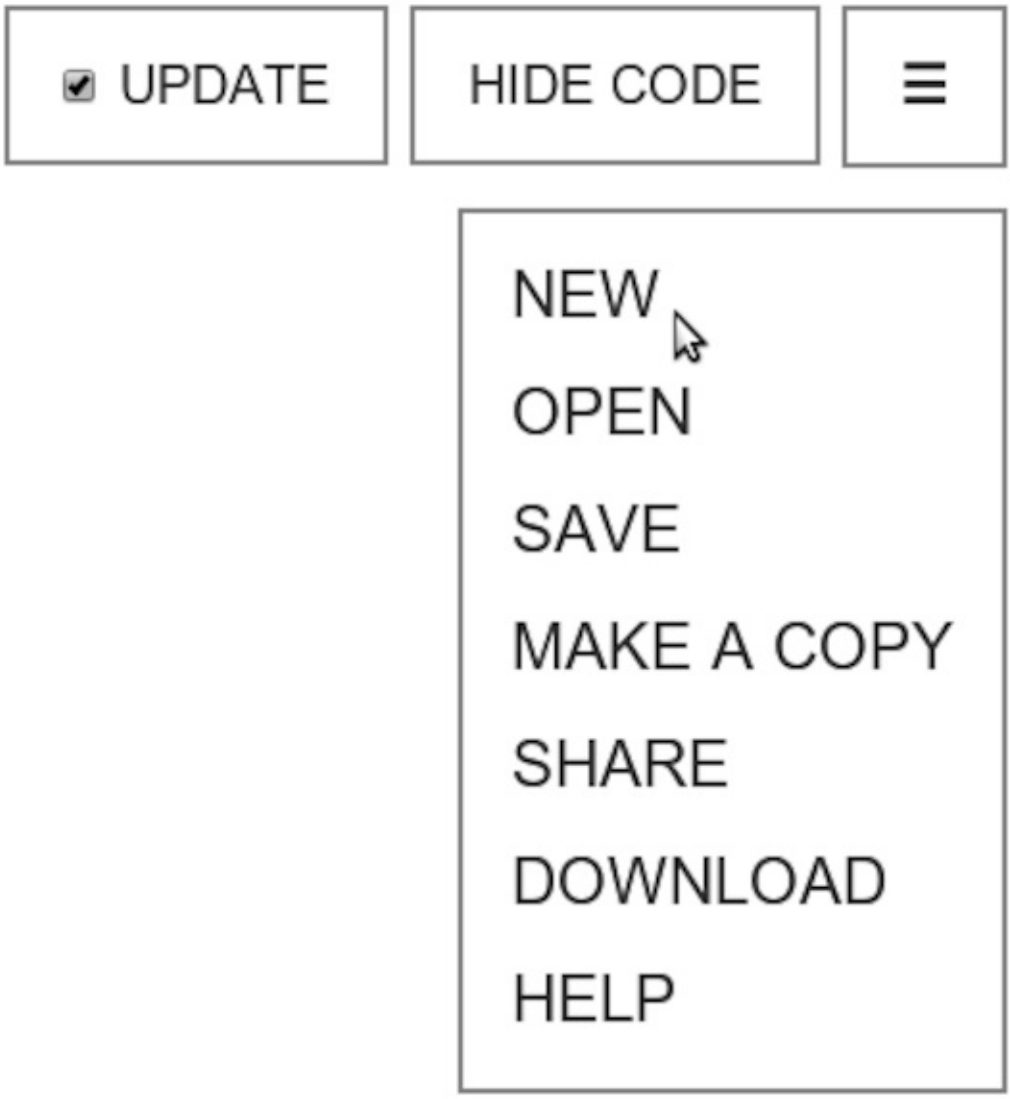
- It runs right in your browser.
- It lets us type in our programming code and see the results immediately.
- Your work is saved automatically every time you type—you can also save from the menu if you want.
- You can download projects for use on other sites (we'll talk about that in Chapter 20, *Getting Code on the Web*).
- You can export all projects for backups or to load them on another computer.
- It runs offline—after the first visit, 3DE is stored in your browser so you can keep working even if you're not connected to the internet!

Remember, you'll need to use the Google Chrome browser for this book. Although most exercises will work in other browsers, it's easiest to stick with a single browser for all our projects—especially when you want to ask questions in the book's forum.[4]

To get started, open the 3DE Code Editor[5] using Chrome. It should look something like this:



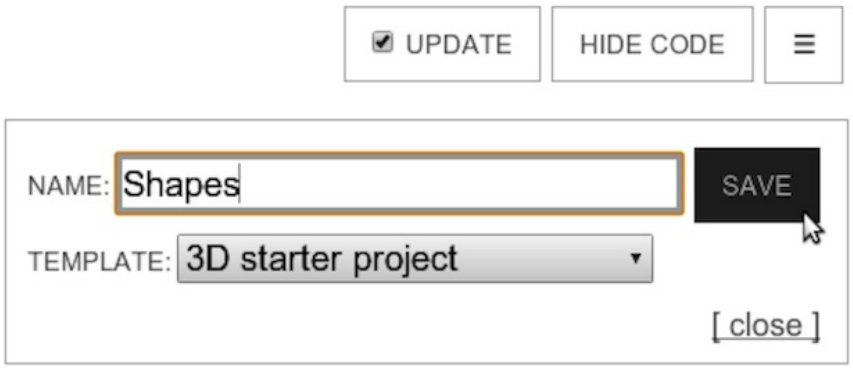
That spinning, multisided thing is a sample of some of the stuff we'll be working on in this book. In this chapter we'll create a new project named **Shapes**.



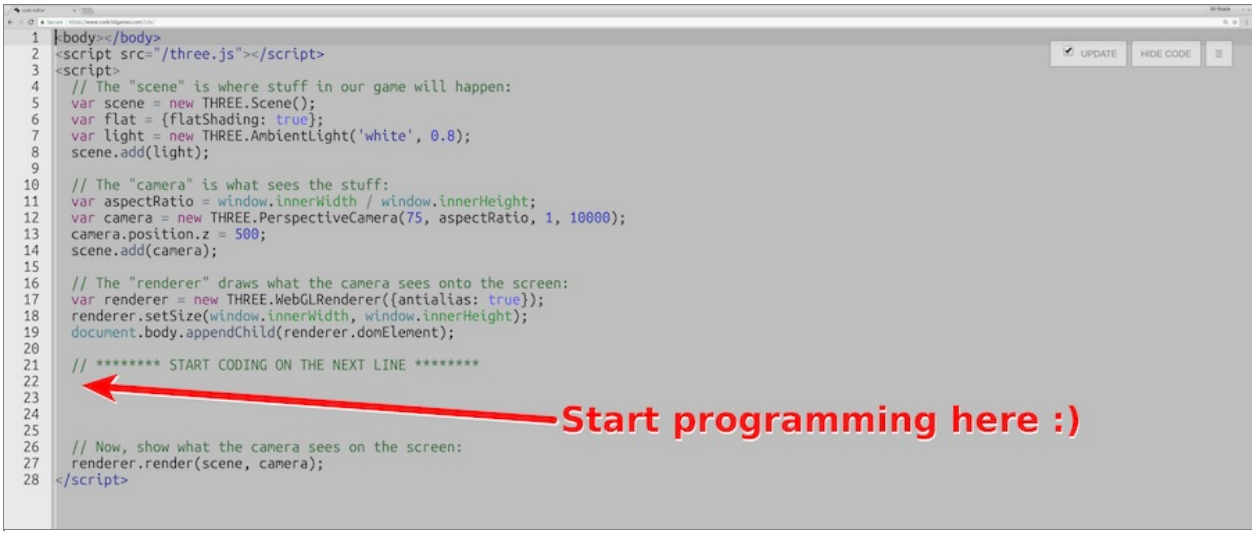
Create Your First Programming Project

To create a new project in the 3DE Code Editor, we click the menu button (the button with three horizontal lines) in the upper-right corner of the window and select NEW from the drop-down.

Type the name of the project, Shapes, in the text field and leave the template set to 3D starter project. Click SAVE as shown in the figure.



When 3DE opens a new 3D project, we see a lot of code is already in the file. We'll talk about all that other code in Chapter 9, *What's All That Other Code?*. For now, let's begin our programming adventure on line 22. Look for the line that says **START CODING ON THE NEXT LINE**.



On line 22, type the following:

```
var shape = new THREE.SphereGeometry(100);
var cover = new THREE.MeshNormalMaterial(flat);
var ball = new THREE.Mesh(shape, cover);
scene.add(ball);
```

When you finish typing that, you should see something cool:

```
20
21  // ******** START CODING ON THE NEXT LINE ********
22  var shape = new THREE.SphereGeometry(100);
23  var cover = new THREE.MeshNormalMaterial(flat);
24  var ball = new THREE.Mesh(shape, cover);
25  scene.add(ball);
26
27
28
29
```

The ball that we typed—the ball that we *programmed*—showed up in 3DE. Congratulations! You just wrote your first JavaScript program!

If that didn't work, double-check that you typed everything exactly as shown. Pay close attention to capitalization as that's a common cause of problems. If that still doesn't work, skip ahead to Chapter 2, *Debugging: Fixing Code When Things Go Wrong* for troubleshooting steps. And if that doesn't work, ask in the online book forum.

Let's take a quick look at the 3D programming we just did. 3D things are built from two parts: the shape and something that covers the shape. The combination of these two things, the shape and its cover, is given a special name in 3D programming: mesh.

Mesh is a fancy word for a 3D thing. Meshes need shapes (sometimes called *geometry*) and something to cover them (sometimes called *materials*). In this chapter we'll look at different shapes. We won't deal with different covers for our shapes until later.

Once we have a mesh, we add it to the scene. The scene is where the magic happens in 3D programming. It's the world in which everything takes place. In this case, it's where our ball is hanging out, waiting for some friends. Let's add some other shapes to the scene so that the ball isn't lonely.

Making Shapes with JavaScript

We've met one shape: the sphere. Lots of shapes are available to 3D programmers. Shapes can be simple, like cubes, pyramids, cones, and spheres. Shapes can also be more complex, like faces or cars. In this book we'll stick with simple shapes. When we build things like trees, we'll combine simple shapes, such as spheres and cylinders, to make them.

So let's explore those shapes…

Creating Spheres

Balls are called *spheres* in geometry and in 3D programming. There are two ways to control spheres in JavaScript.

Size: SphereGeometry(100)

The first way that we can control a sphere is to describe how big it is. When we said `new THREE.SphereGeometry(100)`, we created a ball whose radius was `100`. What happens when you change the radius to `250`?

```
» var shape = new THREE.SphereGeometry(250);
  var cover = new THREE.MeshNormalMaterial(flat);
  var ball = new THREE.Mesh(shape, cover);
  scene.add(ball);
```

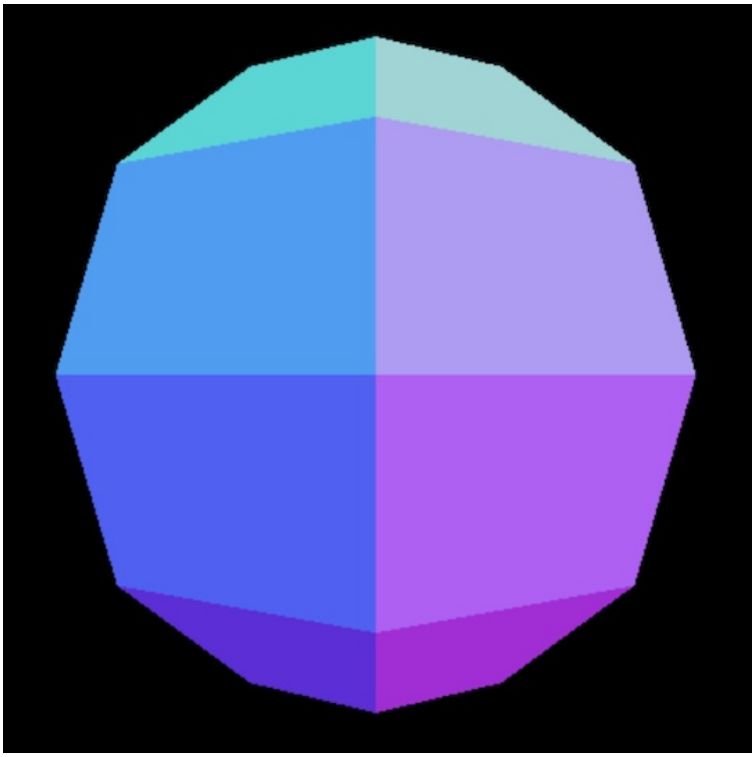
This should make it much bigger:

```
10  // The "camera" is what sees the stuff:
11  var aspectRatio = window.innerWidth / window.innerHeight;
12  var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
13  camera.position.z = 500;
14  scene.add(camera);
15
16  // The "renderer" draws what the camera sees onto the screen:
17  var renderer = new THREE.WebGLRenderer({antialias: true});
18  renderer.setSize(window.innerWidth, window.innerHeight);
19  document.body.appendChild(renderer.domElement);
20
21  // ******** START CODING ON THE NEXT LINE ********
22  var shape = new THREE.SphereGeometry(250);
23  var cover = new THREE.MeshNormalMaterial(flat);
24  var ball = new THREE.Mesh(shape, cover);
25  scene.add(ball);
26
27
28
29
30  // Now, show what the camera sees on the screen:
31  renderer.render(scene, camera);
32 </script>
33
34
35
36
37
38
```

What happens if you change the `250` to `10`? As you probably guessed, it gets much smaller. So that's one way we can control a sphere. What is the other way?

Not Chunky: SphereGeometry(100, 20, 15)

If you click the Hide Code button in 3DE, you may notice that our sphere isn't *really* a smooth ball:



	You Can Easily Hide or Show the Code
	If you click the Hide Code button in the upper-right corner of the 3DE window, you'll see just the game area and the objects in the game. This is how you'll play games in later chapters. To get your code back, click the Show Code button within the 3DE Code Editor.

Computers can't really make a ball. Instead they fake it by joining a bunch of squares (and sometimes triangles) to make something that looks like a ball. Normally, we'll get enough *chunks*—the squares or triangles making up the surface—so that it's close enough.

Sometimes we want it to look a little smoother. To make it smoother, add some extra numbers to the `SphereGeometry` line:

```
» var shape = new THREE.SphereGeometry(100, 20, 15);

  var cover = new THREE.MeshNormalMaterial(flat);
  var ball = new THREE.Mesh(shape, cover);
  scene.add(ball);
```

The first number is the size, the second number is the number of chunks around the sphere, and the third number is the number of chunks up and down the sphere.

This should make a sphere that is much smoother:

The number of chunks we get without telling `SphereGeometry` to use more may not seem great, but don't change it unless you must. The more chunks that are in a shape, the harder the computer has to work to draw it. It's usually easier for a computer to make things look smooth by choosing a different cover for the shape.

Let's Play!
Play around with the numbers a bit more. You're

already learning quite a bit here, and playing with the numbers is a great way to keep learning!

Just don’t make these numbers too high. Anything much beyond 1000 can lock the browser! Don’t worry too much if the browser freezes or stops responding. You can always fix it with the steps described in *Recovering When 3DE Is Broken*.

When you’re done playing, move the ball out of the way by setting its position:

```
    var shape = new THREE.SphereGeometry(100);
    var cover = new THREE.MeshNormalMaterial(flat);
    var ball = new THREE.Mesh(shape, cover);
    scene.add(ball);
» ball.position.set(-250,250,-250);
```

The three numbers move the ball to the left, up, and back. This frees up space to play with our next shape!

Making Boxes with the Cube Shape

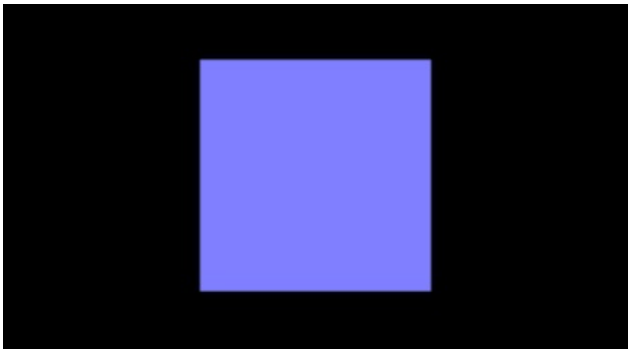
Next we'll make a *cube*, which is another name for a box. In 3D programming, the sides of a cube don't have to be the same size. We can control the width, the height, and the depth.

Size: CubeGeometry(300, 100, 20)

To create a box, we'll write more JavaScript below everything that we used to create our ball. Type the following (maybe add a blank line first):

```
var shape = new THREE.CubeGeometry(100, 100, 100);
var cover = new THREE.MeshNormalMaterial(flat);
var box = new THREE.Mesh(shape, cover);
scene.add(box);
```

If you have everything correct, you should see… a square?



Well, that's boring. Why do we see a square instead of a box? The answer is that our *camera*, our perspective, is looking directly at one side of the box. If we want to see more of the box, we need to move the camera or turn the box. Let's turn the box by rotating it:

```
var shape = new THREE.CubeGeometry(100, 100, 100);
var cover = new THREE.MeshNormalMaterial(flat);
var box = new THREE.Mesh(shape, cover);
scene.add(box);
» box.rotation.set(0.5, 0.5, 0);
```

These three numbers turn the box down, counterclockwise, and left-right. In this case, we rotate `0.5` down and `0.5` to the right:

Setting the box rotation to `(0.5, 0.5, 0)` rotates the cube so we can see that it really is a cube:

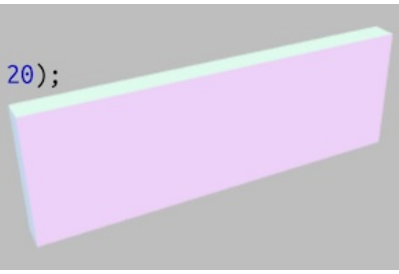
```
var shape = new THREE.CubeGeometry(100, 100, 100);
var cover = new THREE.MeshNormalMaterial();
var box = new THREE.Mesh(shape, cover);
scene.add(box);
box.rotation.set(0.5, 0.5, 0);
```

Our box so far is `100` wide (from left to right), `100` tall (up and down), and `100` deep (front to back). Let's change it so that it's `300` wide, `100` tall, and only `20` deep:

```
» var shape = new THREE.CubeGeometry(300, 100, 20);
  var cover = new THREE.MeshNormalMaterial(flat);
  var box = new THREE.Mesh(shape, cover);
  scene.add(box);
  box.rotation.set(0.5, 0.5, 0);
```

This should produce something like this:

```
var shape = new THREE.CubeGeometry(300, 100, 20);
var cover = new THREE.MeshNormalMaterial();
var box = new THREE.Mesh(shape, cover);
scene.add(box);
box.rotation.set(0.5, 0.5, 0);
```



	Let's Play!
	Take some time to experiment with the numbers inside the `CubeGeometry` and the numbers for the rotation. Both take a little getting used to. And

playing with them is the best way to get there!

Believe it or not, you now know a ton about JavaScript and 3D programming. There is still a lot to learn, of course, but you can already make balls and boxes. You can already move them and turn them. And you only had to write ten lines of JavaScript to do it all—nice!

Let’s move our box out of the way so we can play with more shapes:

```
  var shape = new THREE.CubeGeometry(300, 100, 20);
  var cover = new THREE.MeshNormalMaterial(flat);
  var box = new THREE.Mesh(shape, cover);
  scene.add(box);
  box.rotation.set(0.5, 0.5, 0);
» box.position.set(250, 250, -250);
```

Using Cylinders for All Kinds of Shapes

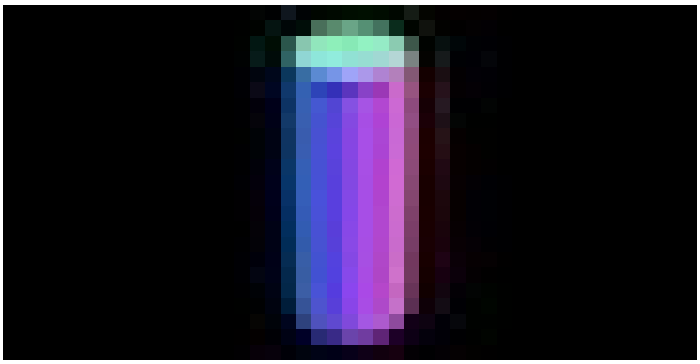
A *cylinder*, which is sometimes also called a tube, is a surprisingly useful shape in 3D programming. Think about it: cylinders can be used as tree trunks, tin cans, wheels… lots of things! But did you know that cylinders can be used to create cones, evergreen trees, and even pyramids? Let's see how!

Size: CylinderGeometry(20, 20, 100)

Below the box code, type in the following to create a cylinder (a blank line first can make your code easier to read):

```
var shape = new THREE.CylinderGeometry(20, 20, 100);
var cover = new THREE.MeshNormalMaterial(flat);
var tube = new THREE.Mesh(shape, cover);
scene.add(tube);
```

If you rotate the tube a little (you remember how to do that from the last section, right?), then you might see something like this:



If you were not able to figure out how to rotate the tube, don't worry. Just type this line after the line with `scene.add(tube)`:

```
tube.rotation.set(0.5, 0, 0);
```

When making a cylinder, the first two numbers describe how big the top and bottom of the cylinder are. The last number is how tall the cylinder is. So our cylinder has a top and bottom that are `20` in size and `100` in height.

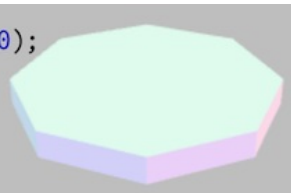
Let's Play!

Play with those numbers and see what you can create! If you change the first two numbers to `100` and the last number to `20`, what happens? What happens if you make the top `1`, the bottom `100`, and the height `100`?

What did you find?

A flat cylinder is a disc:

```
var shape = new THREE.CylinderGeometry(100, 100, 20);
var cover = new THREE.MeshNormalMaterial(flat);
var tube = new THREE.Mesh(shape, cover);
scene.add(tube);
tube.rotation.set(0.5, 0, 0);
```



And a cylinder that has either the top or bottom with a size of `1` is a cone:

```
var shape = new THREE.CylinderGeometry(1, 100, 100);
var cover = new THREE.MeshNormalMaterial(flat);
var tube = new THREE.Mesh(shape, cover);
scene.add(tube);
tube.rotation.set(0.5, 0, 0);
```

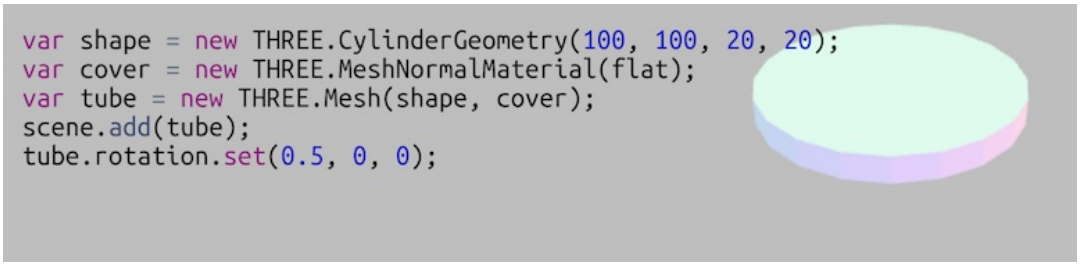
It should be clear that you can do a lot with cylinders, but we haven't seen everything yet. We have one trick left.

Pyramids: CylinderGeometry(1, 100, 100, 4)

Did you notice that cylinders look chunky like the spheres? It should be no surprise then, that you can control the chunkiness of cylinders. If you set the number of chunks to `20`, for instance, with the disc, like this:

```
» var shape = new THREE.CylinderGeometry(100, 100, 20, 20);
  var cover = new THREE.MeshNormalMaterial(flat);
  var tube = new THREE.Mesh(shape, cover);
  scene.add(tube);
  tube.rotation.set(0.5, 0, 0);
```

Then you should see something like this:



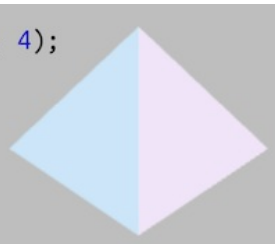
Just as with spheres, you should use lots of chunks like that only when you really, really need to.

Can you think how you might turn this into a pyramid? You have all of the clues that you need.

Let's Play!
 Play with different numbers and see what you can create!

Were you able to figure it out? Don't worry if you weren't. The way we'll do it is actually pretty sneaky. The answer is that you need to *decrease* the number of chunks that you use to make a cone. If you set the top to `1`, the bottom to `100`, the height to `100`, and the number of chunks to `4`, then you'll get this:

```
var shape = new THREE.CylinderGeometry(1, 100, 100, 4);
var cover = new THREE.MeshNormalMaterial(flat);
var tube = new THREE.Mesh(shape, cover);
scene.add(tube);
tube.rotation.set(0.5, 0, 0);
```



It might seem like a cheat to do something like this to create a pyramid, but this brings us to a very important tip with any programming:

Cheat Whenever Possible
 You shouldn't cheat in real life, but in programming—especially in 3D programming—you should always look for easier ways of doing things. Even if there is a *usual* way to do something, there may be a *better* way to do it.

You're doing great so far. Move the tube out of the center like we did with the cube and the sphere:

```
tube.position.set(250, -250, -250);
```

Now let's move on to the last two shapes of this chapter.

Building Flat Surfaces with Planes

A *plane* is a flat surface. Planes are especially useful for the ground, but they can also be handy to mark doors and edges in our games.

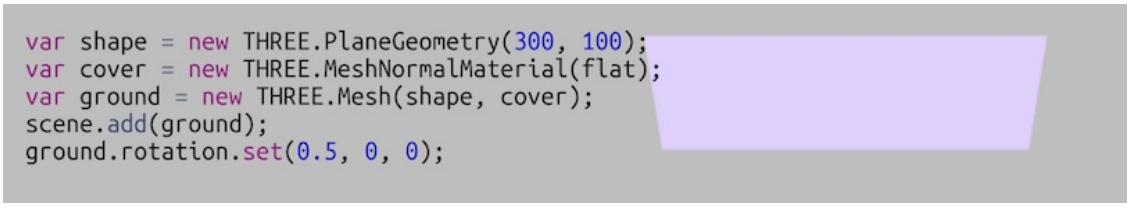
PlaneGeometry(100, 100)

Since planes are just flat squares, they are much simpler than the other objects we've seen so far. Below your cylinder code (and a blank line), type the following:

```
var shape = new THREE.PlaneGeometry(100, 100);
var cover = new THREE.MeshNormalMaterial(flat);
var ground = new THREE.Mesh(shape, cover);
scene.add(ground);
ground.rotation.set(0.5, 0, 0);
```

Don't forget the rotation on the last line. Planes are so thin that you might not see them when looking at them sideways.

The numbers when building a plane are the width and depth. A plane that is `300` wide and `100` deep might look like this:



That's pretty much all there is to know about planes. Move our plane out of the way:

```
    var shape = new THREE.PlaneGeometry(300, 100);
    var cover = new THREE.MeshNormalMaterial(flat);
    var ground = new THREE.Mesh(shape, cover);
    scene.add(ground);
»   ground.position.set(-250, -250, -250);
```

Now let's move on to the greatest shape in the world.

Rendering Donuts (Not the Kind You Eat) with Torus

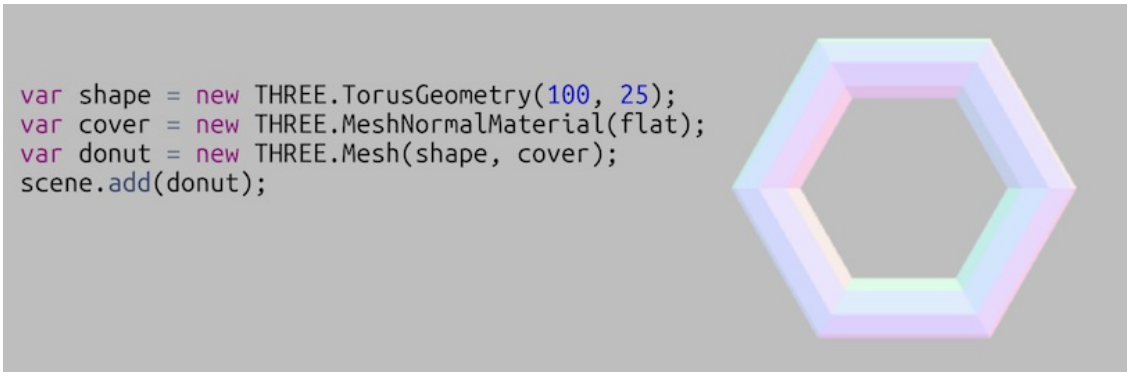
In 3D-programming-speak, a donut shape is called a *torus*. The simplest torus that we can create needs us to assign two values: one for the distance from the center to the outside edge, and the other for the thickness of the tube.

TorusGeometry(100, 25)

Type the following below your plane code:

```
var shape = new THREE.TorusGeometry(100, 25);
var cover = new THREE.MeshNormalMaterial(flat);
var donut = new THREE.Mesh(shape, cover);
scene.add(donut);
```

You should see a very chunky donut:



By now you probably know how to make the donut less chunky.

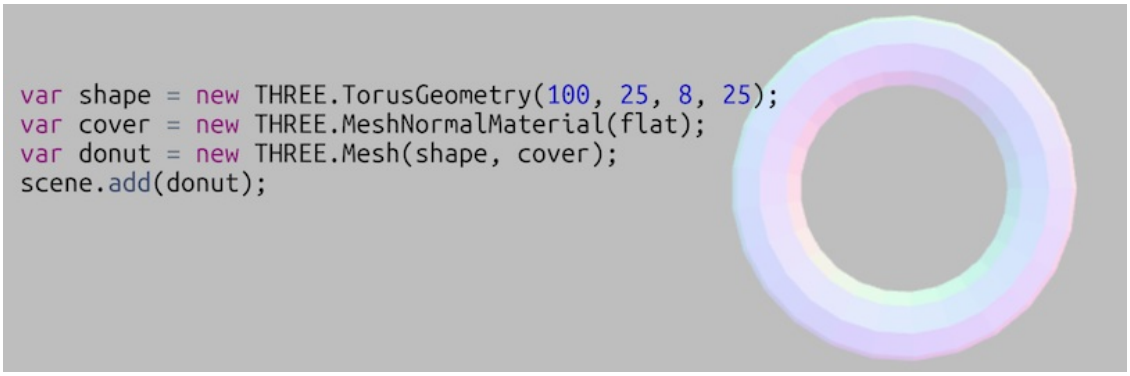
TorusGeometry(100, 25, 8, 25)

Like the sphere, the donut shape is built from chunks. The chunks can be made bigger or smaller around the inner tube, which we can set by including a third number when defining the `TorusGeometry`. We can also include a fourth number to adjust the size of the chunks around the outside of the donut. Try experimenting with numbers like the following and see what happens.

```
var shape = new THREE.TorusGeometry(100, 25, 8, 25);
var cover = new THREE.MeshNormalMaterial(flat);
var donut = new THREE.Mesh(shape, cover);
```

```
scene.add(donut);
```

Now *that* is a good-looking donut:



TorusGeometry(100, 25, 8, 25, 3.14)

We can play one other trick with donuts. Try adding another number, `3.14`, to the `TorusGeometry` shape:

```
var shape = new THREE.TorusGeometry(100, 25, 8, 25, 3.14);
var cover = new THREE.MeshNormalMaterial(flat);
var donut = new THREE.Mesh(shape, cover);
scene.add(donut);
```

That should make a half-eaten donut! Does `3.14` seem like a weird number? It might, depending on how much math you've learned. We'll talk more about it in Chapter 7, *A Closer Look at JavaScript Fundamentals*.

We now have five numbers that control the donut. The first two are the size of the donut and the size of the tube. The next two control the chunkiness. The last controls how much of the donut to draw.

Animating the Shapes

Before we finish our first programming session, let's do something cool. Let's make all of our shapes spin around like crazy.

In 3DE, add the following code after all of the shapes:

```
var clock = new THREE.Clock();

function animate() {
  requestAnimationFrame(animate);
  var t = clock.getElapsedTime();

  ball.rotation.set(t, 2*t, 0);
  box.rotation.set(t, 2*t, 0);
  tube.rotation.set(t, 2*t, 0);
  ground.rotation.set(t, 2*t, 0);
  donut.rotation.set(t, 2*t, 0);

  renderer.render(scene, camera);
}

animate();
```

Don't worry about what everything means in that code. For now, it's enough to know that at specific time intervals, we're changing the shape's rotation. After each change, we tell the renderer—the thing that draws the scene on our computer screens—to redraw the current shapes in their updated rotations.

	If 3DE Locks Up
	When doing animations and other sophisticated programming, it's possible to completely lock up the 3DE Code Editor. This is not a big deal. If 3DE stops responding, you'll need to undo whatever change you made last. Instructions on how to do that are in *Recovering When 3DE Is Broken*.

The Code So Far

To make things a little easier, the completed version of this project is included as part of *Code: Creating Simple Shapes*. Use that code to double-check your work as you go through the exercises, but try not to copy and paste code into 3DE. It's impossible to learn and understand programming unless you code it yourself.

What’s Next

Whoa! That was pretty wild. We learned a ton and we’re just getting started!

Already we know how to code projects in the 3DE Code Editor. We know how to make a lot of different shapes. We even know how to move and spin things with JavaScript. And best of all, it took us only fifteen lines of code to create a pretty cool animation after making our shapes. That’s a good start.

Now that we have a taste of 3D programming, in the next chapter we’ll talk about programming in web browsers.

Footnotes

[4] https://talk.code3Dgames.com/

[5] http://code3Dgames.com/3de

When you're done with this chapter, you will

- *Recognize errors from the code editor*
- *Know how to look for errors in the browser's JavaScript console*
- *Be able to fix projects when a program freezes*

Chapter 2

Debugging: Fixing Code When Things Go Wrong

Programming is a great skill to have. Being able to fix broken programs is an even better skill to have.

Code breaks. All the time. Sometimes the people you work with make mistakes—no matter how good they are. Sometimes we make mistakes—no matter how careful we are. Mistakes happen and code breaks. And it doesn't matter who broke it. What really matters is who is going to fix it. *Hint:* it's you!

So in this chapter, we'll look at some strategies to help us fix things—to help us debug broken code. Sometimes our code editor can help. Other times, we can use the browser's JavaScript console. Sometimes we even break the entire browser. We'll figure out how to fix that, too.

	Programming Can Be Overwhelming
	At times, programming can make you want to throw your computer against the wall (don't). When programming, keep these two facts in mind: • There will be things that you don't know—this is OK. • Your programs are going to break—this is OK. Just remember that everyone struggles with this, and

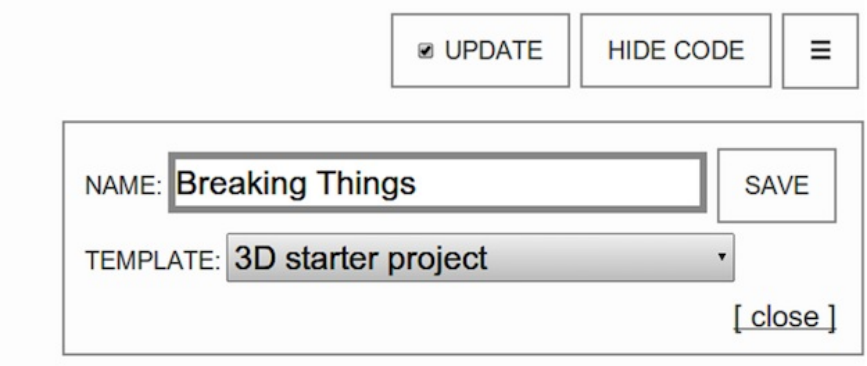
you’ll be just fine.

Getting Started

	Know the Code Editor
	We're still using the 3DE Code Editor that we used in Chapter 1, *Project: Creating Simple Shapes*. If you haven't already gotten started with 3DE, go back to that chapter and familiarize yourself with the editor.

Start a New Project

Any work that you've already done in 3DE is automatically saved. So we can create a new project for this chapter and won't lose the cool shapes from the first chapter. Click the menu button and then choose **NEW** from the menu: Let's call the new project **Breaking Things**.



Be sure to leave the template set to **3D starter project**.

Let's start by breaking simple things that our code editor can find for us.

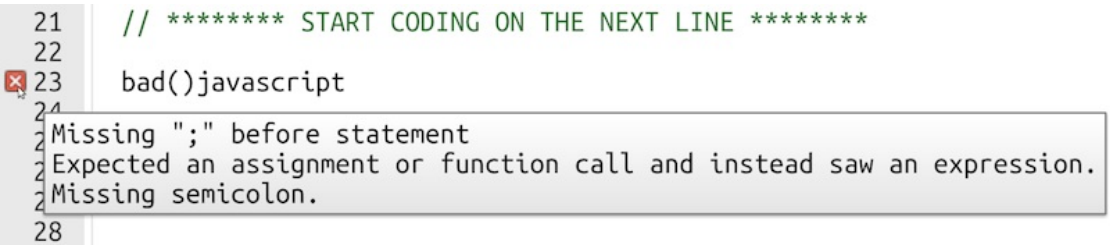
Debugging in 3DE: The Red X

A red X next to your code means 3DE sees a problem that will stop your code from running. Let's write some really bad JavaScript to see this action. Enter the following line below START CODING ON THE NEXT LINE.

```
bad()javascript
```

That's some bad JavaScript!

Are you wondering why? It's bad because you should never have a word that comes right after parentheses in JavaScript. If you write code like this, 3DE will show a red X next to the problem line, indicating that the line has to be fixed. Move the mouse pointer over the red X to display the actual error message as shown in the following figure.



Our code editor really did not like that JavaScript! We'll see what all of that means when we discuss Chapter 5, *Functions: Use and Use Again* and Chapter 7, *A Closer Look at JavaScript Fundamentals*. Just try to remember that this is how the code editor tells us that there are words after the parentheses when there shouldn't be.

Some things to check in your code when you see a red X:

- Did you forget a semicolon? Missing other JavaScript punctuation like parentheses, brackets, or quotes? Look around.
- If you don't see a problem on the red X line, look at the previous line as well. If you don't see it there, look at the line below. 3DE can't always tell where the problem begins and may be off by one or two lines.

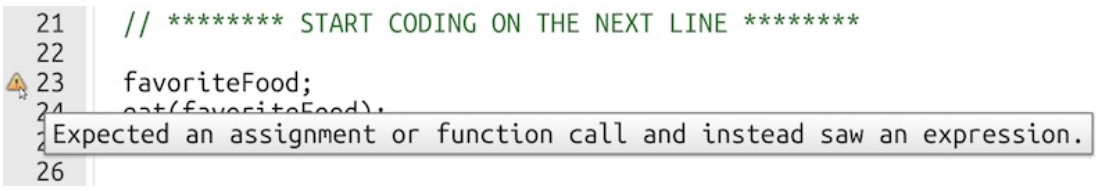
Debugging in 3DE: The Yellow Triangle

Unlike a red X, a yellow triangle that shows up to the left of your code is not a showstopper. Your code will probably run even if lines in your code are marked with yellow triangles, but it may not run *correctly*. It's best to get rid of those triangles as they come up.

Let's see this in action by writing some more bad JavaScript (but not *too* bad). First, remove the `badjavascript` line from the previous section. Then add the following lines:

```
favoriteFood;
eat(favoriteFood);
```

In this case, 3DE will tell us via the yellow triangle that the `food` line is not doing anything.



To fix the problem, we can change the `food` line into an assignment, which sets—or *assigns*—a value.

```
» var favoriteFood = 'Cookie';
eat(favoriteFood);
```

3DE should accept the new `favoriteFood` line and no longer display any errors. However, even though 3DE may not report any more issues, something is still wrong with this code. To figure out what that is, we're going to need the web programmer's best friend, the JavaScript console.

Opening and Closing the JavaScript Console

If code is not working as expected, one of the first places to check is the browser's JavaScript console. The quickest way to do that is with special keyboard combinations that only JavaScript programmers like us know!

On PCs and Chromebooks, `Ctrl+Shift+J` (holding down the `Ctrl`, `Shift`, and `J` keys at the same time) will open and close the JavaScript console.

If you're using an Apple computer, you can use `⌘+Option+J` to open and close the console.

Don't worry if you see tons of warnings and errors the first time you open the JavaScript console. It keeps a log of events that happen on a web page or in the 3DE Code Editor. If the messages are too much, you can clear the log with the button that has a circle with a line through it, then click the `UPDATE` button in 3DE to show only the current log information.

The same key combination that opens the JavaScript console will close it (but leave it open in this chapter).

Debugging in the Console

Once you open up the console, you'll see an error message that `eat` is not defined.

```
▶Uncaught ReferenceError: eat is not defined                    code.html:24
    at code.html:24
```

No matter how the error is formatted, the browser sees bad code and reports it in the JavaScript console. On line 24 of our program, we told the browser to run the `eat` *function*. The only problem is that we never told the browser *how* to do that!

Error Formatting May Change

Don't be surprised if your console error doesn't look exactly as shown. The report format can change a lot—even for the same code. Sometimes it'll add another number to the error message. Like '24:3' instead of just '24'.

```
Uncaught ReferenceError: eat is not defined                    code.html:24
    at code.html:24:3
```

Sometimes it'll add extra lines.

```
▶Uncaught ReferenceError: eat is not defined                    code.html:24
    at code.html:24
    at _.html:10
```

Sometimes it'll start the errors with "VM" followed by strange numbers.

```
▶Uncaught ReferenceError: eat is not defined                    code.html:24
    at VM141 code.html:24
    at VM139 _.html:10
```

There's not much rhyme or reason to the differences. Pay the most attention to the error message itself ('Uncaught ReferenceError: eat is not defined') and the first number after 'code.html' ('24').

We'll talk more about functions in Chapter 5, *Functions: Use and Use Again*. For now, it's enough to know that a function is a way to write code that can be

run again and again.

To resolve this problem, let's give our JavaScript program an `eat()` function.

```
var favoriteFood = 'Cookie';
eat(favoriteFood);

» function eat(food) {
»   console.log(food);
» }
```

At this point, 3DE should still have no red or yellow triangles. And now no errors should show in the JavaScript console. We fixed the errors by adding that `eat()` function.

Thanks to what's inside the `eat()` function, the JavaScript console should have the word "Cookie." We might eventually want the `eat()` function to create a monster that eats whatever food we send to the function. For now, all that `eat()` does is send the food to `console.log()`. As you might guess, `console.log()` takes whatever you send and prints it out in the JavaScript console.

	The console.log() Utility Is Super Helpful
	Browsers have some pretty sophisticated debugging tools in them. Sometimes they can be useful—they're worth learning some day, just not for this book. Often the most useful debugging tool is `console.log()`. When code gets complex, it's easy to get confused by values. If you're ever unsure of a value, send it to `console.log()`, then check the results in the JavaScript console.

You can send more than one thing to `console.log()` if you put a comma in between things.

```
function eat(food) {
```

```
»   console.log(food, '!!! Nom. Nom. Nom');
}
```

Thanks to that `console.log()` statement, the message, “Cookie !!! Nom. Nom. Nom” should be in the console.

Why Some Errors Only Happen in Running Code

Before a program is run, computers read code and convert it into an internal structure. This process is called *compiling* a program.

Code editors can compile code first to see if it’ll run. If it won’t compile, they can flag these *compile time* errors with red and yellow triangles. Code editors are good for finding these compile time errors.

Other errors won’t occur until the code actually runs. The code compiles OK, but might be missing something (like a function) that can’t be seen until the code runs. These are *runtime* errors. The JavaScript console helps us fix runtime errors.

Before we wrap up this chapter, let’s look at some 3D programming errors that you’re likely to run into.

Common 3D Programming Errors

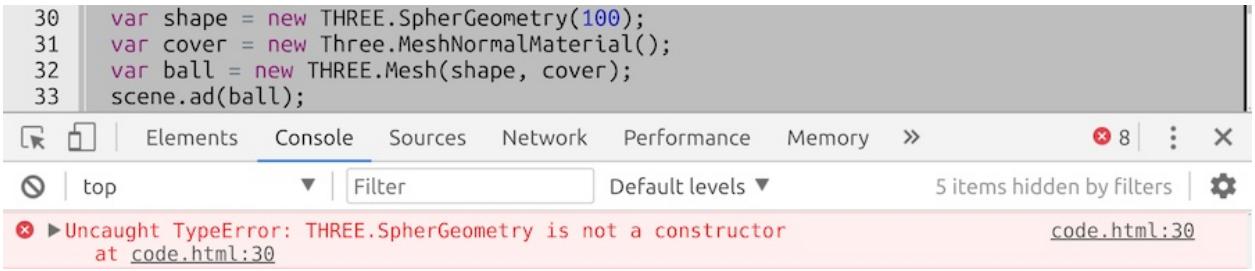
Keep the JavaScript console open for this section. After the closing curly brace of the `eat` function, add a blank line and then type the highlighted code.

```
function eat(food) {
  console.log(food, '!!! Nom. Nom. Nom');
}

» var shape = new THREE.SpherGeometry(100);
» var cover = new Three.MeshNormalMaterial();
» var ball = new THREE.Mesh(shape, cover);
» scene.ad(ball);
```

You'll notice that the editor does not see any errors in this code. The browser reads the JavaScript code and says, "Yup, that looks like perfectly fine JavaScript to me. I'll run it now!" However, problems pop up when the code is actually run, and you'll see errors in the JavaScript console.

Possible Error Message—Not a Constructor

Let's take a look at what went wrong. In the JavaScript console, you should see an error message.



This message is trying to tell us that `SphereGeometry` is spelled incorrectly on line 31. Check the code; it turns out we missed an `e` and typed `SpherGeometry` instead. Don't worry about the rest of the message, we'll learn about *constructors* later.

Console Line Numbers Are Not Always Exact
3DE does its best to get the line numbers in the

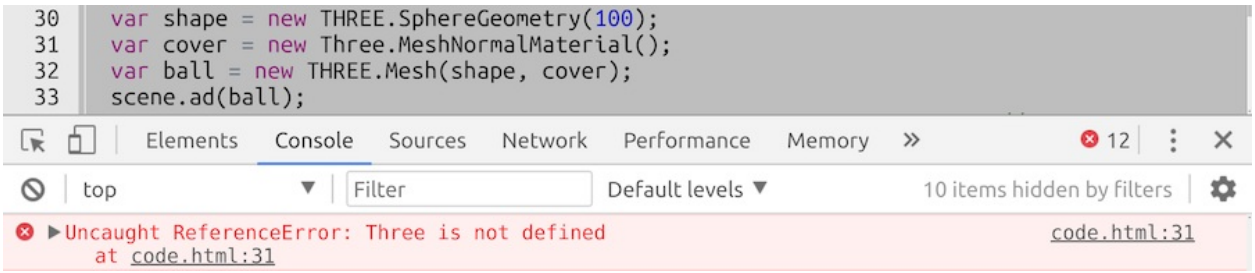
console correct, and sometimes it succeeds—it may even be correct for you now. But other times it can be off by a few lines. Start by looking at the exact line number. If that doesn't seem like it matches the error, then check the next few lines.

So if you find this message in the JavaScript console, double-check the spelling of the "THREE" shapes and covers. Also double-check that the correct letters are capitalized.

Possible Error Message—Three Is Not Defined

Fix the spelling of `SphereGeometry`. Even after doing that, the ball still doesn't appear on the screen. Something else is wrong with our code.

Looking in the JavaScript console, you should see something like the following.



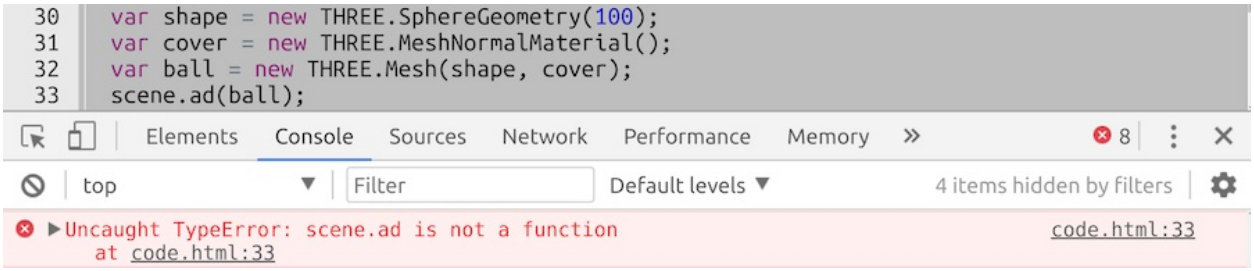
Here, the JavaScript console is telling us we forgot that `THREE` should always be all capital letters. In this book we're using a collection of JavaScript code called *Three.js* Because the authors of Three.js love capital letters so much, there's no such thing as `Three`—only `THREE`.

This is a very common mistake when working with Three.js code. It's also easy to mistype other JavaScript code or names. So try to remember how fussy JavaScript is the next time you see a "not defined" error.

We can fix this problem by replacing the `Three` in the code with `THREE`.

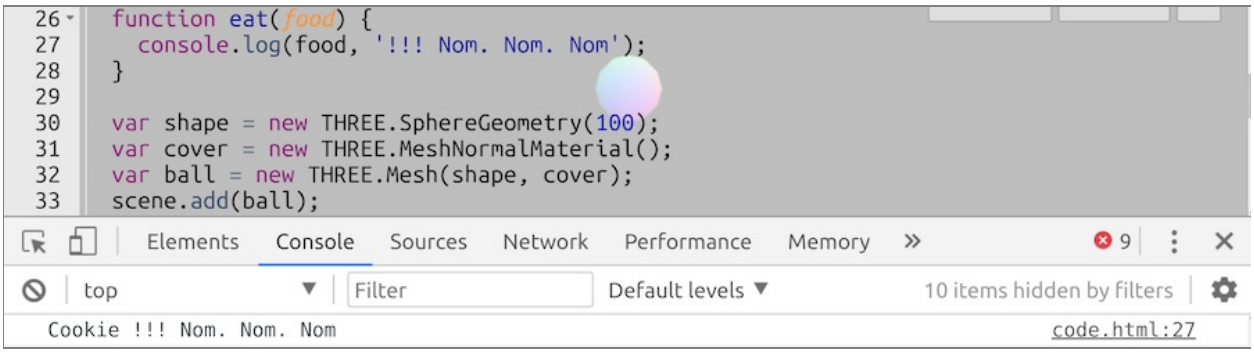
Possible Error Message—Not a Function

Even with those two issues fixed, the sphere is still not visible and we have another error message in the console.



In this case, we told the browser that we could use code named `ad`. The browser tried, but it was unable to find anything that worked because we meant to use `add()`, not `ad()`. In other words, we don't want to *ad* the ball to the screen; we want to *add* it.

After fixing that line, you'll finally see a ball, and the "Nom. Nom. Nom." message again appears in the JavaScript console.



Recovering When 3DE Is Broken

Breaking a web browser is surprisingly easy. It is so easy and happens so often that people have invented many descriptions for a broken browser: frozen, locked, stopped, or just broken. If you create a sphere with a million chunks, the browser will completely freeze up. If you create a code loop with no stopping point, the browser will lock. No typing. No scrolling. Nothing.

If the browser is frozen, then the 3DE Code Editor is broken, right?

Well, yes, but there's an easy way to fix it: add `?e` or `?edit-only` to the URL so you're looking at http://code3Dgames.com/3de/?e. This is edit-only mode for 3DE.

Fix the last thing that you typed to break 3DE, and then remove the edit-only question mark from the URL so that you're back at http://code3Dgames.com/3de/. Now you should see the preview again.

On some computers, you may find that you need to close the browser tab or window before you try this. Then you can open a new window or tab, in which you can enter the 3DE edit-only URL. Google Chromebooks, in particular, run edit-only mode better with this procedure.

What’s Next

You picked up an amazing amount of useful information in this chapter. The best programmers are all creative problem solvers. Debugging code is a big part of solving problems. Look back at this chapter from time to time to remind yourself how to do some of this.

You have now seen how to use the code editor’s warning and error indicators to troubleshoot code. You’ve also learned how to use the JavaScript console to see and fix errors that the editor can’t find. And don’t underestimate the power of `console.log()`—it can be a huge help. You now know some of the most common 3D programming mistakes that you might make. And if things go really, REALLY wrong, you can recover from a frozen browser.

Now that we know how to make shapes and where to check when things go wrong, let’s get started on our first game by building our very own avatar.

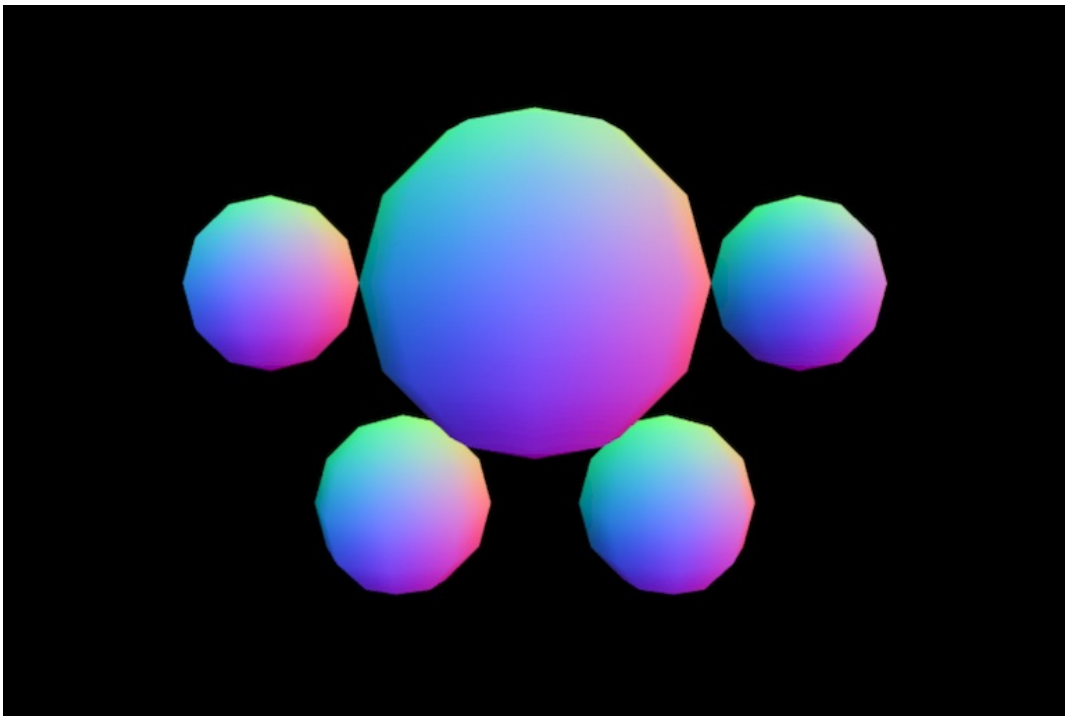
When you're done with this chapter, you will

- *Be able to group simple shapes*
- *Build your own game avatar*
- *Add simple animations to the avatar*

Chapter 3

Project: Making an Avatar

Developing games means lots of parts—the game area, the players in the game, things that get in the way of players, and much, much more. In this project chapter, we'll create a player that we might use in a game—an *avatar*. It will end up looking something like this:



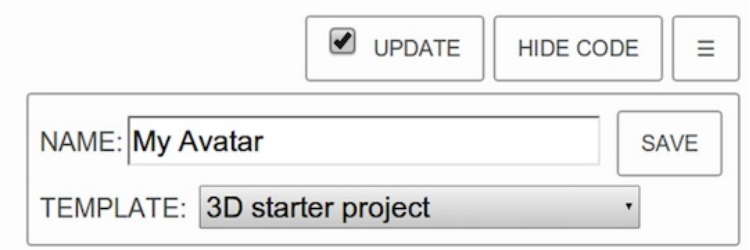
An avatar is who you are within the game world. It shows where you are in the game and what you're doing. Since it's supposed to represent you and me, it should have a good feel to it. We want something more than a plain old box.

The Difference Between a Player and an Avatar

In this book, we'll use the word *player* to mean the person playing the game. The word *avatar* will be used to describe the thing that represents the player inside the game.

Getting Started

Let's open the 3DE Code Editor[6] again and create a new project named **My Avatar** (check *Start a New Project*, if you don't remember how).



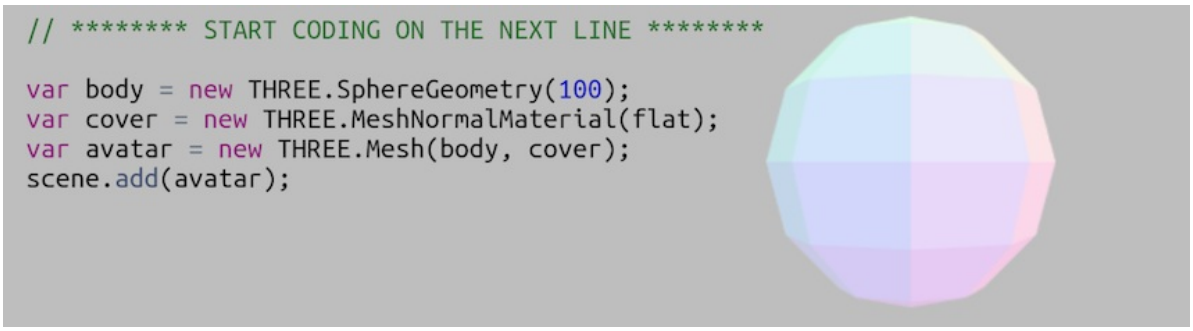
Be sure to leave the template set to **3D starter project**. With that, we're ready to start programming on the line after **START CODING ON THE NEXT LINE**.

Smooth Chunkiness

Let's start our avatar with a large sphere for the body. Start with the same code that we used in Chapter 1, *Project: Creating Simple Shapes*.

```
var body = new THREE.SphereGeometry(100);
var cover = new THREE.MeshNormalMaterial(flat);
var avatar = new THREE.Mesh(body, cover);
scene.add(avatar);
```

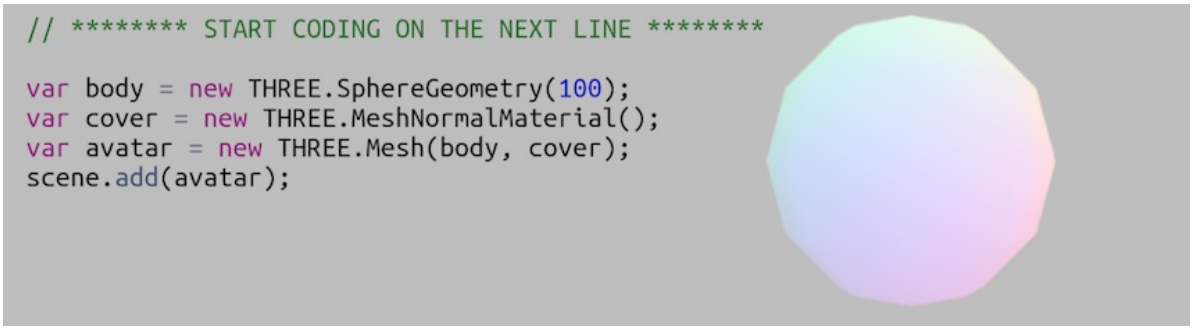
We already know what happens when we type that in—we get a ball in the center of the scene.



Before we get any further, let's remove `flat` from the second line.

```
var body = new THREE.SphereGeometry(100);
» var cover = new THREE.MeshNormalMaterial();
var avatar = new THREE.Mesh(body, cover);
scene.add(avatar);
```

What does that do? When `flat` was there, we were telling our cover to use flat chunks. Removing `flat` lets our cover smooth those chunks out as shown in the figure.



That’s a pretty smooth ball! Even better, we didn’t have to fiddle with the number of chunks to get that smoothness. Our 3D code just does it for us automatically. And if we want it even smoother, we can still play with the number of chunks as we did in *Not Chunky: SphereGeometry(100, 20, 15)*.

The rest of the avatar examples will show smooth chunks. But the choice between smooth and flat chunks is entirely up to you, the game programmer. If you want your avatar to have a retro look, a flat cover is a great choice. And you can always switch later if you change your mind.

Making a Whole from Parts

Let's add a hand next to the body. Add the following lines below the code that you already entered to create the body.

```
var hand = new THREE.SphereGeometry(50);

var rightHand = new THREE.Mesh(hand, cover);
rightHand.position.set(-150, 0, 0);
scene.add(rightHand);
```

Notice that we didn't create a new cover for the hand. Instead we reused the same cover, which we named `cover` when we used it for the avatar's body. That saves us a bit of typing.

Less typing is a good thing since we're all programmers and programmers are lazy at heart. That reminds me of some programming wisdom worth sharing:

	Good Programmers Are Lazy
	I don't mean that programmers hate doing work. We actually love our jobs and often spend too much time working because we love it so much. No, what I mean by *lazy* is that we hate doing work that computers are better at. So instead of creating hands and feet individually, we would rather write a single hand/foot and then copy it as many times as necessary.

Being lazy benefits us in two very important ways:

- We type less. Believe it or not, this is a big win. Not only do we have to type less the first time around, but we have to read less when we want to update later.

- If we want to change the way a limb is created, we only have to change one thing. That is, if we want to change the cover or even the shape of a hand in the future, then we only have to make a change in one place.

So let's see if we can be even lazier when we create the left hand for our avatar:

```
var leftHand = new THREE.Mesh(hand, cover);
leftHand.position.set(150, 0, 0);
scene.add(leftHand);
```

Not only did we not make a new cover for the left hand, but we also didn't create a new mesh! Instead we just used the same mesh for the left hand that we did for the right hand. Now that's lazy!

With that, our avatar should look something like this:

```
var hand = new THREE.SphereGeometry(50);

var rightHand = new THREE.Mesh(hand, cover);
rightHand.position.set(-150, 0, 0);
scene.add(rightHand);

var leftHand = new THREE.Mesh(hand, cover);
leftHand.position.set(150, 0, 0);
scene.add(leftHand);
```

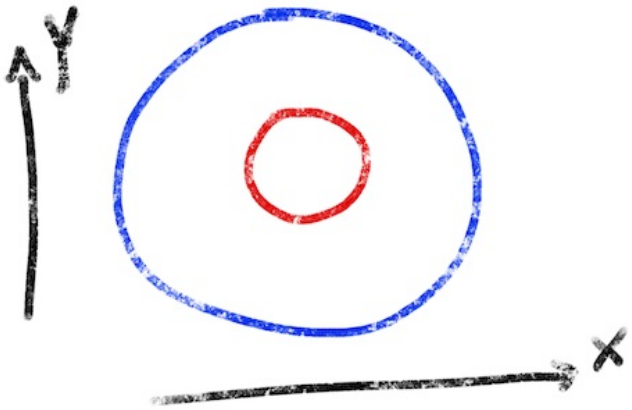
OK, I admit that doesn't look much like a body with hands. It will; just bear with me for a bit longer.

Breaking It Down

Let's take a quick look at why we used those numbers for the hands.

	Read This Section! (at Some Point)
	If you're impatient, go ahead and skip to *Adding Feet for Walking*, where you can continue building our game avatar. You'll still learn plenty just by programming the rest of the avatar. And it's OK to be impatient to keep going. Just be sure to come back here at some point so you can *really* understand the numbers that we're using in this chapter.

When anything is added to a scene, it starts off in the very center. So when we add the body and a hand, it starts off something like this:



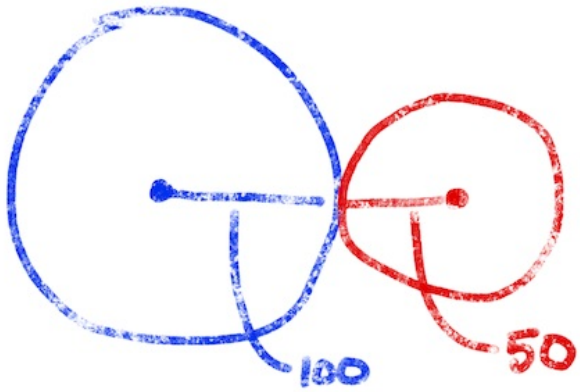
In 3D programming and mathematics, left and right are called the X direction. Up and down are called the Y direction.

This is why we change the X position of the hands:

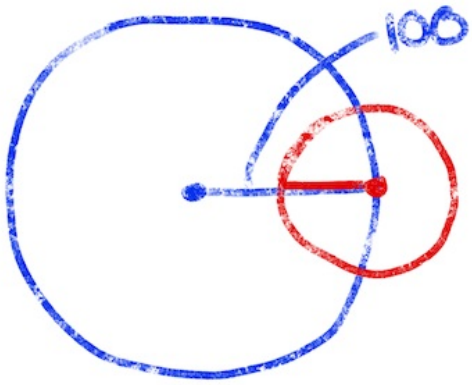
```
var leftHand = new THREE.Mesh(hand, cover);
leftHand.position.set(150, 0, 0);
scene.add(leftHand);
```

The numbers inside `leftHand.position.set(150, 0, 0)` are the X, Y, and Z positions of the left hand (Z would be forward and backward). We set X to `150`, while Y and Z are both `0`. This is really the same thing as `leftHand.position.x = 150`. As we'll see shortly, it can be very convenient to set multiple values on a single line.

But why `150`? The answer is that the radius of the body is 100 and the radius of the hand is 50. We need to move the hand 100 + 50, or 150 in the X (left/right) direction:



If we only moved the center of the hand 100, then we would end up with the hand partly inside the body:



	Let's Play!
	If you're not convinced, try it yourself. Change the number for the X position by fiddling with the first

number in `rightHand.position.set(-150, 0, 0)`. Try it for both the left and right hands. Don’t make them too big, though, or they won’t even be on the screen anymore!

Adding Feet for Walking

For the feet, we'll again use spheres of size `50`. I'll leave it up to you to figure out how to add the relevant lines of code.

Some hints:

- Don't move the feet left/right as far as we did the hands. The feet should be underneath the body.
- You'll have to move them down. The up/down positioning is done with the Y direction instead of the X direction. With `leftHand.position.set(150, 0, 0)`, we set the X position to 150. We want to change the second number too. You may have to use negative numbers to go down—for example, –25.
- Recall that the hand was added before we *rendered* the scene—before the line with `renderer.render(scene, camera)`. The feet should be added before rendering the scene as well.

Here is how we did the right hand; this might help you figure out the feet:

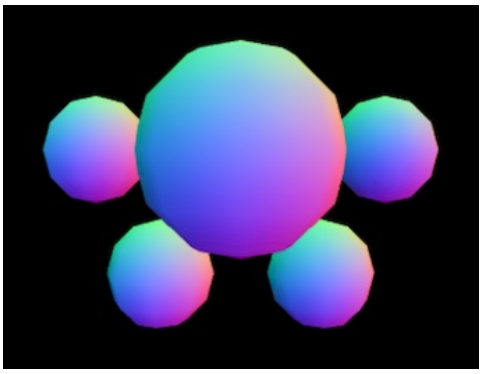
```
var hand = new THREE.SphereGeometry(50);

var rightHand = new THREE.Mesh(hand, cover);
rightHand.position.set(-150, 0, 0);
scene.add(rightHand);
```

Good luck!

	Let's Play!
	Try to place the feet yourself. To move the feet left and right, you change the first number in `rightFoot.position.set(0, 0, 0)`. To move it up and down, you change the second number (the third number is forward and backward).

It may take a while to get it right, but believe me—it's good practice. Try for a bit and then continue with the text.

Did you get it?

This is what it might look like: 

Don't worry if yours is not exactly the same. Yours may even be better!

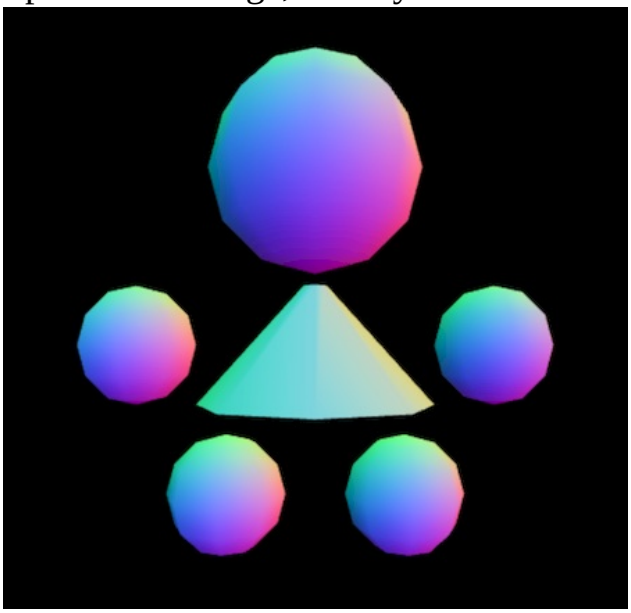
If you're having difficulty, refer to the code that we used to make the avatar:

```
var body = new THREE.SphereGeometry(100);
var cover = new THREE.MeshNormalMaterial();
var avatar = new THREE.Mesh(body, cover);
scene.add(avatar);

var hand = new THREE.SphereGeometry(50);

var rightHand = new THREE.Mesh(hand, cover);
rightHand.position.set(-150, 0, 0);
scene.add(rightHand);

var leftHand = new THREE.Mesh(hand, cover);
leftHand.position.set(150, 0, 0);
scene.add(leftHand);

var foot = new THREE.SphereGeometry(50);
```

```
var rightFoot = new THREE.Mesh(foot, cover);
rightFoot.position.set(-75, -125, 0);
scene.add(rightFoot);

var leftFoot = new THREE.Mesh(foot, cover);
leftFoot.position.set(75, -125, 0);
scene.add(leftFoot);
```

This is everything after **START CODING ON THE NEXT LINE**.

Challenge: Make the Avatar Your Own

If you're up for a challenge, see if you can create an avatar that looks something like this: 

To make this, you need to replace the body with one of the shapes from Chapter 1, *Project: Creating Simple Shapes*, and add a head. Don't worry about arms and legs to connect the hands and feet to the body—that would make it harder in later chapters.

And, of course, you can make whatever kind of avatar you like. *Just remember to make one with hands and feet*—we'll need them in later chapters.

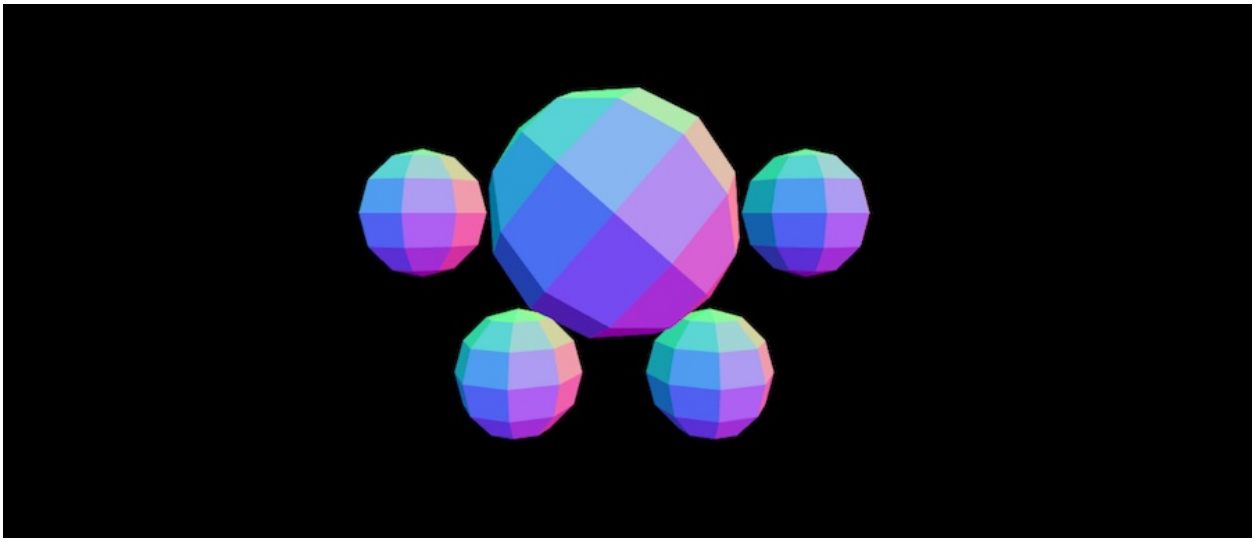
Doing Cartwheels

We'll add controls to our avatar later. But before moving on to the next lesson, let's make the avatar do some flips and cartwheels.

Just like we did at the end of Chapter 1, *Project: Creating Simple Shapes*, we start by changing the very last line of the code (which is just above the `</script>` tag) at the end of the editor. Instead of telling the browser to show the scene one time, we animate the scene as follows.

```
// Now, animate what the camera sees on the screen:
function animate() {
  requestAnimationFrame(animate);
  avatar.rotation.z = avatar.rotation.z + 0.05;
  renderer.render(scene, camera);
}
animate();
```

If you typed everything correctly, you might notice something odd. Just the head is spinning, not the whole avatar.



That might be a cool effect, but it's not what we wanted. So how do we go about spinning the whole avatar?

If you guessed that we add `rotation.z` changes to the hands and feet, you made a

good guess. But that won't work. The hands and feet would spin in place just like the head.

The answer to this problem is a very powerful 3D-programming technique. We group all of the body parts together and spin the group. It's a simple idea, but, as you'll find later, it's surprisingly powerful.

To group the body parts together, we add the parts to the avatar instead of to the scene.

If you look back up to the code for the right hand, you'll see that we added it to the scene. We'll change that line.

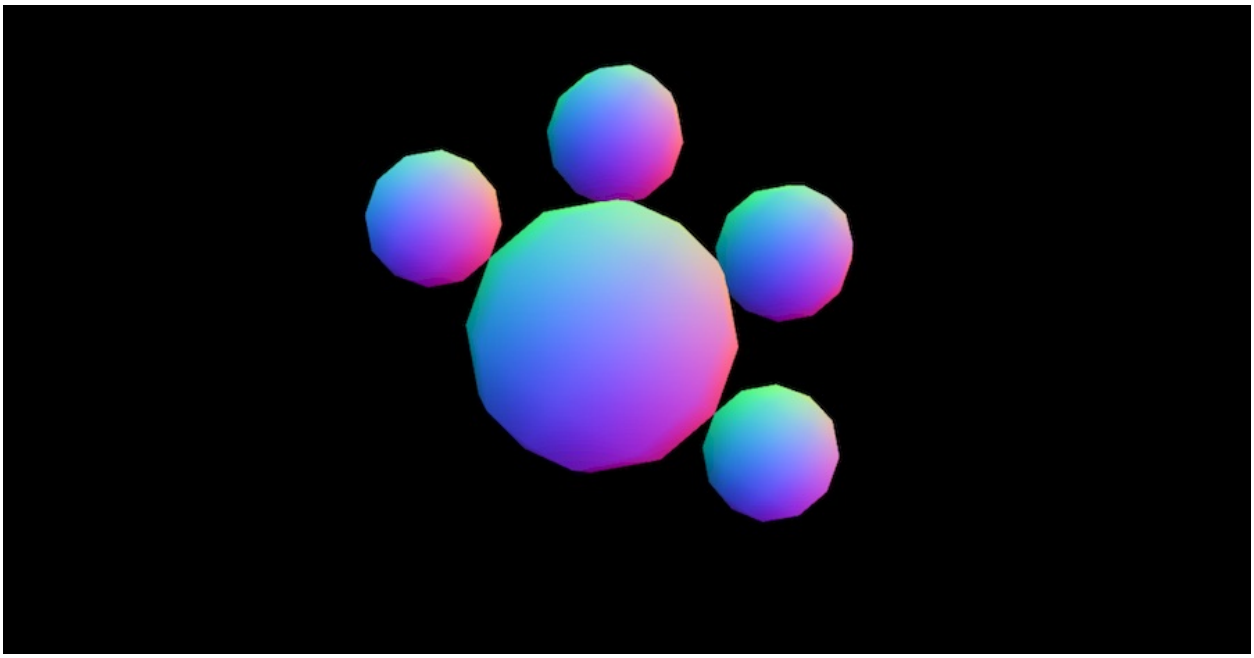
```
  var rightHand = new THREE.Mesh(hand, cover);
  rightHand.position.set(-150, 0, 0);
» scene.add(rightHand);
```

Instead of adding the hand to the scene, we add it to the avatar:

```
  var rightHand = new THREE.Mesh(hand, cover);
  rightHand.position.set(-150, 0, 0);
» avatar.add(rightHand);
```

This line now adds the right hand to the avatar instead of to the scene. Now, the hand will rotate along with the avatar's body.

After doing the same for the `leftHand`, the `rightFoot`, and the `leftFoot`, your avatar should be doing cartwheels—*without* losing any parts!



Sometimes we might not want our avatar to do cartwheels. Let's add a bit of code to control that.

```
① var isCartwheeling = false;
  function animate() {
    requestAnimationFrame(animate);
②   if (isCartwheeling) {
      avatar.rotation.z = avatar.rotation.z + 0.05;
    }
    renderer.render(scene, camera);
  }
  animate();
```

① This is where we say whether our avatar is doing cartwheels or not. If we set this to `true`, then our avatar is doing cartwheels. If we set it to `false` (like we've done here), then our avatar won't do cartwheels.

② Wrap the `avatar.rotation` in an `if`, as shown. Don't forget the curly braces on this line and after the `avatar.rotation` line.

Now change the value of `isCartwheeling` from `false` to `true`. Does the avatar start cartwheeling again?

Let's Play!

Now that you have the avatar cartwheeling, try to make the avatar flip, as well. You should use a value like `isFlipping` to control the flipping. *Hint:* instead of `avatar.rotation.z`, try `avatar.rotation.x` or `avatar.rotation.y`.

Did you get it? If not, it's OK. Really! We'll talk more about this in the next chapters—especially in Chapter 8, *Project: Turning Our Avatar*.

The Code So Far

The entirety of the code will look something like the code in *Code: Making an Avatar*.

Don’t worry if yours isn’t exactly like that code. Your code may be better or just different.

What’s Next

We have a pretty cool-looking avatar. It might be nice for it to have a face or clothes. But you know what would be even better? If we could move our avatar with the keyboard. And that’s just what we’ll do in Chapter 4, *Project: Moving Avatars*.

For now, take some time to play with the size, positioning, and rotation of the parts that make up your avatar.

Footnotes

[6] http://code3Dgames.com/3de

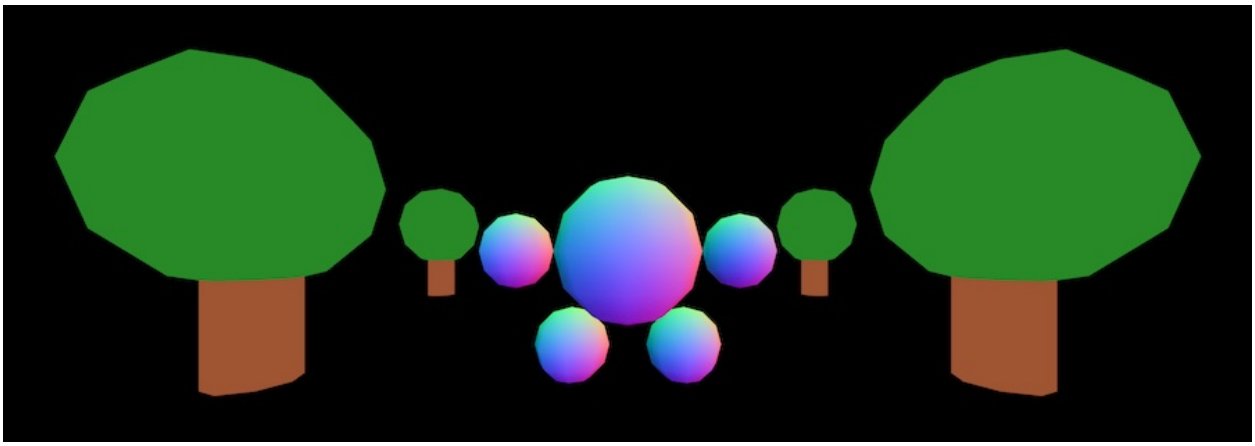
When you're done with this chapter, you will

- *Be able to control things with your keyboard*
- *Understand how to attach cameras so they move along with the avatar*
- *Have your first experience with JavaScript events, which are crazy important and quite weird!*

Chapter 4

Project: Moving Avatars

In Chapter 3, *Project: Making an Avatar*, we covered how to build a game avatar. But an avatar that can't move is pretty dull. So in this chapter we're going to learn how to make the avatar move around the scene. We'll also give it a little forest to play in. It will end up looking something like this:



Seriously, you're gonna feel like you have super powers after you're done with this chapter. And you won't be wrong—programming *is* a super power!

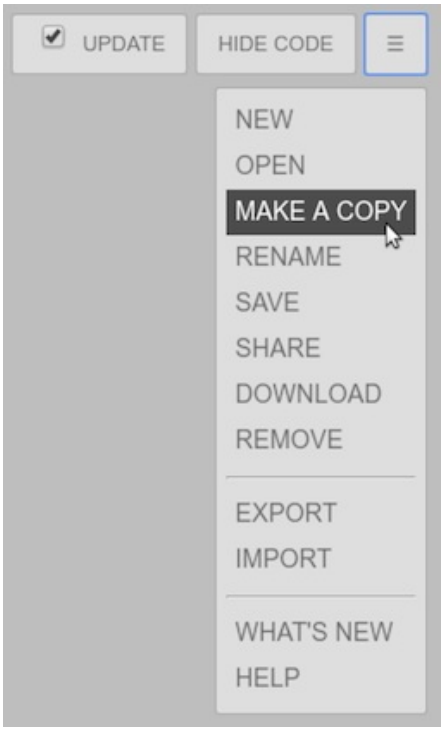
Getting Started

This chapter builds on the work that we did in Chapter 3, *Project: Making an Avatar*. If you haven't already done the exercises in that chapter, go back and do them before proceeding. In particular, you need to go over the animate exercise at the end of that chapter.

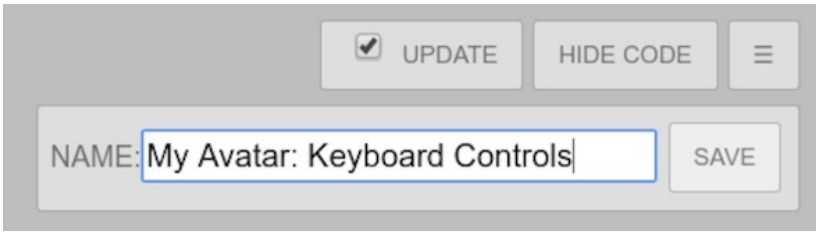
We want to build on the code from the last chapter, but don't want to lose it. Let's copy the avatar project from the last chapter into a new project. That way, our old code will still be there if we need it.

	Save Your Work
	Working code is a rare and wonderful thing. I've been programming for 20 years. I *still* write more broken code than working code. Whenever I finally get something working, the first thing I do is save it. Well, the first thing I do is a little dance—just a small one, I'm pretty terrible—but it's worth a celebration. And anything worth celebrating is worth saving.

To copy a project, click the menu button and choose MAKE A COPY from the menu:



Let’s call this project **My Avatar: Keyboard Controls**. Enter that for the project name.



Then click **SAVE**. With that, we’re ready to add keyboard controls.

Building Interactive Systems with Keyboard Events

When things happen in web browsers, JavaScript thinks about those things as *events*. To have our code do something when an event happens, we write code that *listens* for different kinds of events. Events and event listeners are strange at first, but they make sense once you see some code. So let's code!

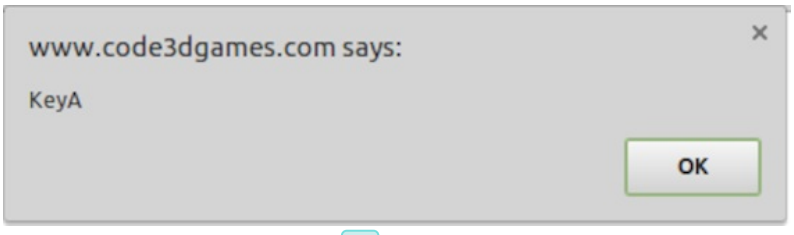
Add the following at the very bottom of our code, below the `animate` line that we added in Chapter 3, *Project: Making an Avatar*.

```
document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  alert(event.code);
}
```

This code listens for a `keydown` event. You probably guessed that these events happen when any key is… pressed down. When our code *hears* a `keydown`, it will tell the `sendKeyDown()` function to use that key to send a command to the avatar.

Before we can tell the avatar to move, we need to figure out which key is pressed. So in our `sendKeyDown()` function, we examine the `event.code` value.

What is that `code`? To answer that, let's try it out! Click the `HIDE CODE` button in the toolbar at the top of the 3DE window, then press the `A` key on your keyboard. You should see something like this alert dialog.



Cool! We pressed the `A` key, which JavaScript calls `KeyA`. What about the arrow keys?

Click the `OK` button on the alert if you haven't already. Then repeat for the left, up, right, and down arrow keys on your keyboard. For the left arrow, you should

discover that the computer thinks it's an `ArrowLeft`. For the up arrow, the computer thinks it's an `ArrowUp`. For the right arrow, the computer detects the key as `ArrowRight`. For the down arrow, the computer thinks it's an `ArrowDown`.

Let's use those key codes to move our avatar!

Converting Keyboard Events into Avatar Movement

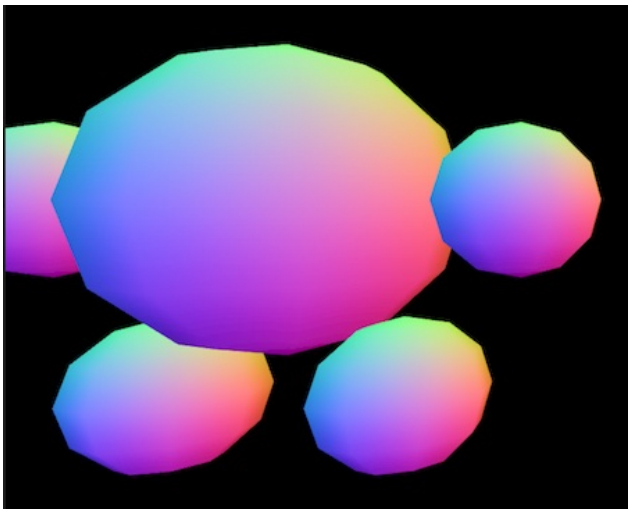
Show your code again, then remove the `alert(event.code)` line inside the `document.addEventListener`. Replace it with the following:

```
  document.addEventListener('keydown', sendKeyDown);
  function sendKeyDown(event) {
»   var code = event.code;
»   if (code == 'ArrowLeft') avatar.position.x = avatar.position.x - 5;
»   if (code == 'ArrowRight') avatar.position.x = avatar.position.x + 5;
»   if (code == 'ArrowUp') avatar.position.z = avatar.position.z - 5;
»   if (code == 'ArrowDown') avatar.position.z = avatar.position.z + 5;
  }
```

We'll talk about `if`, `==` (is it equal?) and `=` (make it equal) in Chapter 7, *A Closer Look at JavaScript Fundamentals*. But this code should make sense even without details. We're checking *if* the key code comes from an arrow key. If the key code is `ArrowLeft`, for example, then we change the avatar's X position by subtracting `5`.

	Let's Play!
	Click the `HIDE CODE` button and give it a try. Use the arrow keys to move the avatar around. Does it work like you expect? *Remember*: If something goes wrong, check the JavaScript console!

If everything is working correctly, then you should be able to move your avatar far away, up close, all the way to the left or right, and even off the screen.

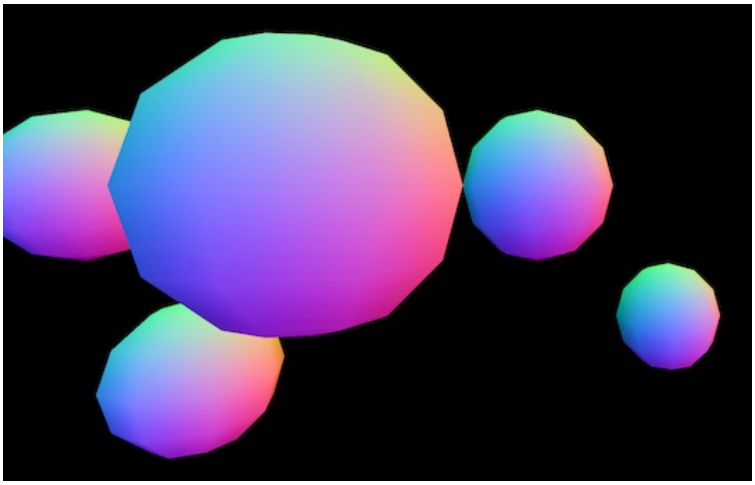


You learned how to make sure the avatar's hands and feet move with the body when we added the ability to do cartwheels back in *Doing Cartwheels*. Since the hands and feet were added to the avatar object instead of the scene, moving the avatar means the hands and feet go along with it.

Let's see what happens if one of the legs is not attached to the avatar. In this case, we'll change the `leftFoot` so that it's added to the scene instead of the avatar.

```
    var leftFoot = new THREE.Mesh(foot, cover);
    leftFoot.position.set(75, -125, 0);
»   scene.add(leftFoot);
```

Run this and the left foot goes missing as shown in the figure.



Don't underestimate the power of this kind of thing. We'll do some crazy stuff with it later.

For now, don’t forget to reattach the left foot to the avatar!

Challenge: Start/Stop Animation

Remember the `isCartwheeling` and `isFlipping` values from when we built the avatar in Chapter 3, *Project: Making an Avatar*? Let's add two more `if` statements to the keyboard event listener. If `KeyC` is pressed, then the avatar should either start or stop cartwheeling. If `KeyF` is pressed, then the flip routine should start or stop.

Hint: You can switch a `true` or `false` by putting an exclamation point in front of it. So you can make `isCartwheeling` switch values with something like `isCartwheeling = !isCartwheeling`. It's weird, but we'll talk more about that in *Booleans*. For now, try to get `isCartwheeling` and `isFlipping` to change when the correct key is pressed.

Were you able to get it working yourself? Don't worry if you didn't—this kind of JavaScript can be a little weird the first time you try it. Here is the `animate` function that handles the cartwheeling and flipping:

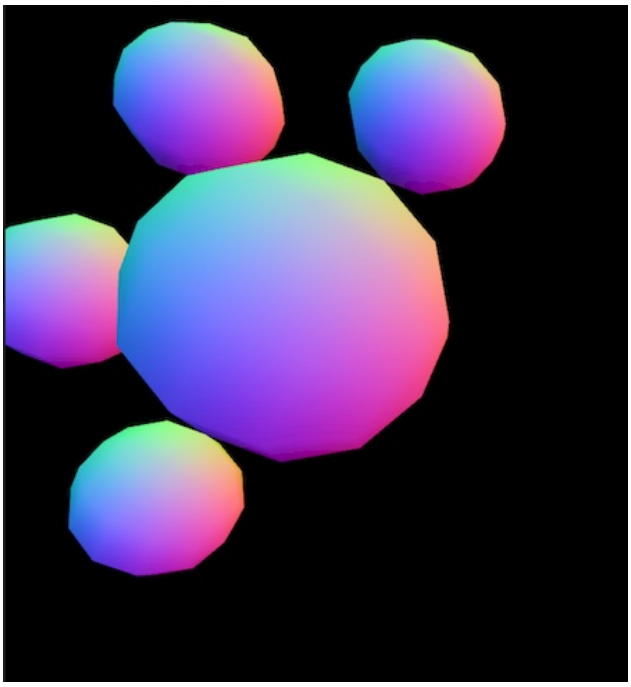
```
var isCartwheeling = false;
var isFlipping = false;
function animate() {
  requestAnimationFrame(animate);
  if (isCartwheeling) {
    avatar.rotation.z = avatar.rotation.z + 0.05;
  }
  if (isFlipping) {
    avatar.rotation.x = avatar.rotation.x + 0.05;
  }
  renderer.render(scene, camera);
}
animate();
```

Here's the complete keyboard code for moving, flipping, and cartwheeling our avatar:

```
document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;
  if (code == 'ArrowLeft') avatar.position.x = avatar.position.x - 5;
  if (code == 'ArrowRight') avatar.position.x = avatar.position.x + 5;
```

```
  if (code == 'ArrowUp') avatar.position.z = avatar.position.z - 5;
  if (code == 'ArrowDown') avatar.position.z = avatar.position.z + 5;

  if (code == 'KeyC') isCartwheeling = !isCartwheeling;
  if (code == 'KeyF') isFlipping = !isFlipping;
}
```

If you've got it right, you should be able to hide your code and make the avatar do flips and cartwheels as it moves off the screen.



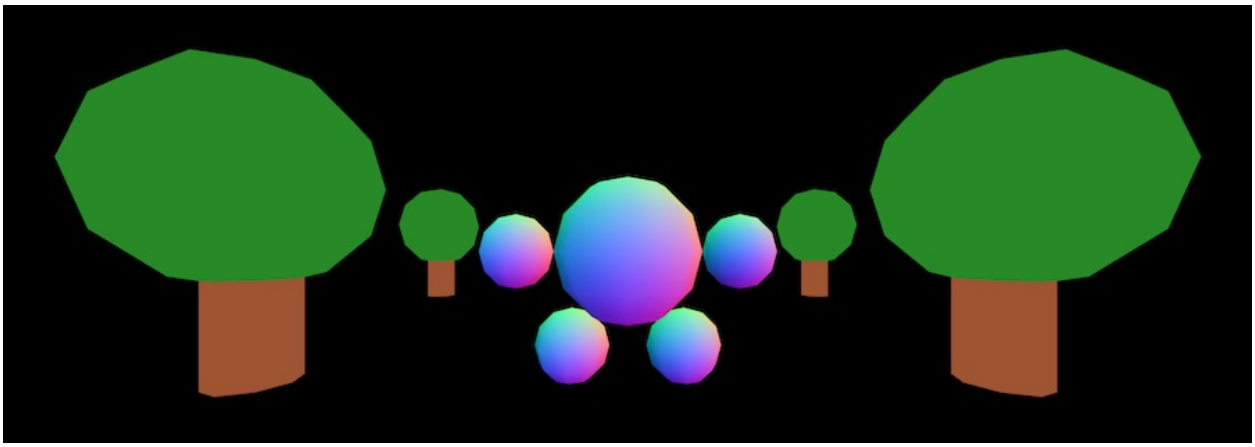
Actually, it's pretty crazy that the avatar can leave the screen. We'll fix that in a bit, but first let's add some scenery for our avatar to explore. Let's give our avatar some trees to walk among.

Building a Forest with Functions

We'll need a lot of trees for our forest. We could build them one at a time, but we're not going to do that. Instead, let's add the following JavaScript after all of the avatar body parts:

```
function makeTreeAt(x, z) {
  var trunk = new THREE.Mesh(
    new THREE.CylinderGeometry(50, 50, 200),
    new THREE.MeshBasicMaterial({color: 'sienna'})
  );

  var top = new THREE.Mesh(
    new THREE.SphereGeometry(150),
    new THREE.MeshBasicMaterial({color: 'forestgreen'})
  );
  top.position.y = 175;
  trunk.add(top);

  trunk.position.set(x, -75, z);
  scene.add(trunk);
}

makeTreeAt( 500,  0);
makeTreeAt(-500,  0);
makeTreeAt( 750, -1000);
makeTreeAt(-750, -1000);
```

If you entered all that code correctly, you'll see the avatar standing in front of a forest of four trees.



That’s pretty cool, but how did we do that?

Breaking It Down

Most of the work is in the `makeTreeAt()` function. As we’ll see in Chapter 5, *Functions: Use and Use Again*, a JavaScript function is a way to run the same code over and over. In this case, the function does all of the repetitive work of building a trunk and treetop. We could have named it anything, but we give it a name that tells us what it does—in this case, it makes a tree at `x` (left/right) and `z` (in/out) coordinates.

The inside of the `makeTreeAt()` function should start to look familiar.

```
   function makeTreeAt(x, z) {
①    var trunk = new THREE.Mesh(
       new THREE.CylinderGeometry(50, 50, 200),
       new THREE.MeshBasicMaterial({color: 'sienna'})
     );

②    var top = new THREE.Mesh(
       new THREE.SphereGeometry(150),
       new THREE.MeshBasicMaterial({color: 'forestgreen'})
     );
③    top.position.y = 175;
④    trunk.add(top);

⑤    trunk.position.set(x, -75, z);
⑥    scene.add(trunk);
   }
```

① Make a trunk out of a cylinder.

② Make a treetop out of a sphere.

③ Move the treetop up (remember Y is up and down) to the top of the trunk.

④ Add the treetop to the trunk.

⑤ Set the position of the trunk to the `x` and `z` values that the function was called with—`makeTreeAt(500,0)`, for example. The Y value of `-75` moves the trunk down enough to look like a tree trunk.

⑥ Add the trunk with its treetop to the scene.

It's important to remember that we have to add the treetop to the trunk and *not* the scene. If the treetop and trunk are both added to the scene, then we would have to remember to move both. With the treetop added to the trunk, moving the trunk also moves the treetop.

A function is not 100% needed here. You're good enough at building shapes by now that you could probably create four cylinder trunks around the scene with four sphere tops on them. But lazy programmers (the best kind!) don't like typing more than we need to. So we teach a function how to build a tree once, then use that function again and again. And if we want to add another 20 trees, we use that function 20 more times.

Also new here is color. We picked those colors from the Wikipedia list of color names.[7] The tree trunk is the color sienna. You can try your own colors if you like. Most color names work, just keep them all lowercase and no spaces (`'forestgreen'` instead of `'forest green'`).

Once we have that function, using it is easy. We make a tree at `x` of `500` and `z` of `0` by *calling* the function with the numbers in parentheses like `makeTreeAt( 500, 0)`.

```
makeTreeAt( 500,  0);
makeTreeAt(-500,  0);
makeTreeAt( 750, -1000);
```

```
    makeTreeAt(-750, -1000);
```

Now that we have a forest, let’s see what we can do about the avatar leaving the screen.

Moving the Camera with the Avatar

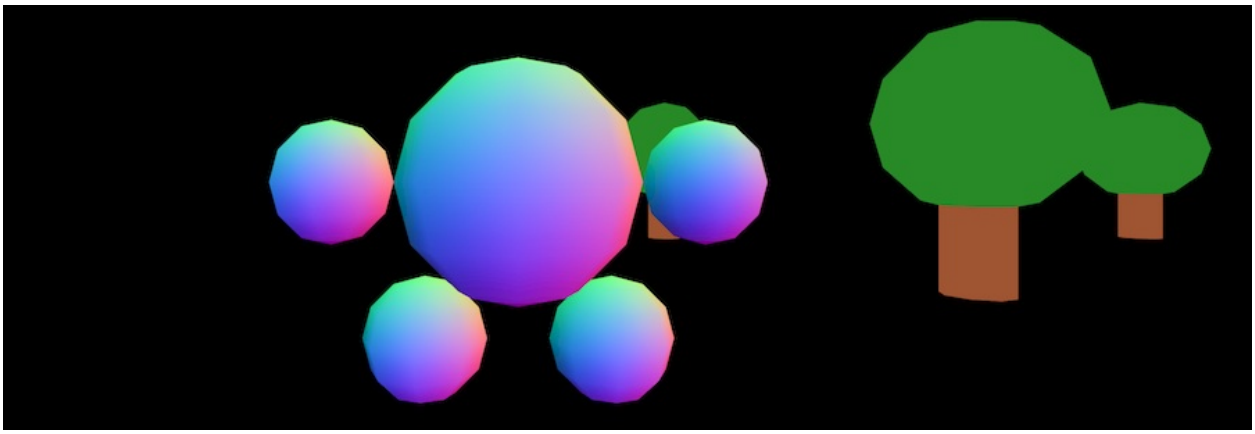
The easiest way to keep the avatar on-screen is to move the camera wherever the avatar moves. If the camera is always pointed at the avatar, then the avatar can't leave the screen!

Now, to get the hands and feet to move along with our avatar, we added them to the avatar's body instead of adding them to the scene. We need to do the same thing with the camera.

First let's find the line that says `scene.add(camera)` and delete it. Then, below the line where the avatar is added to the scene—and above the `makeTreeAt` function, let's add the camera to the avatar:

```
var leftFoot = new THREE.Mesh(foot, cover);
leftFoot.position.set(75, -125, 0);
avatar.add(leftFoot);

» avatar.add(camera);
```

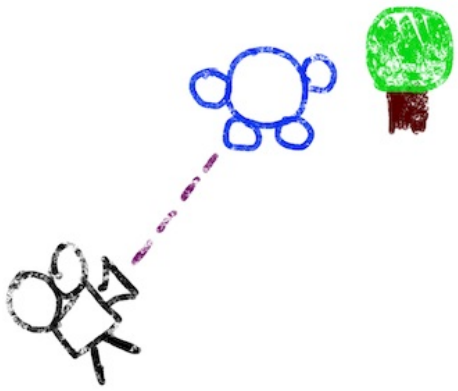
After hiding the code, you'll see that when the avatar is moved, the camera stays right in front of the avatar.



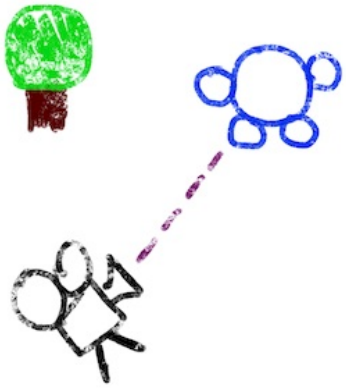
The camera starts 500 units in front of the avatar. The code at the very top sets its position as `camera.position.z = 500` or 500 units in front. Before, it was 500 units in front of the center of the scene. And it stayed there, even if the avatar moved. Now that we've added it to the avatar, it always stays that same distance from

the avatar.

It might help to think of the camera being attached to the avatar with a selfie stick.

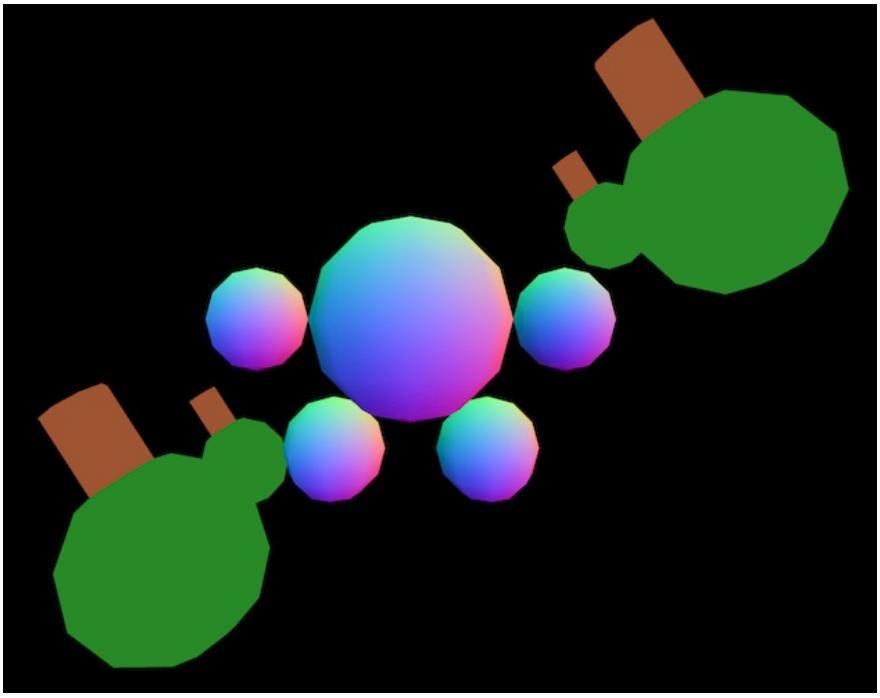


Wherever the avatar goes, the camera goes as well.

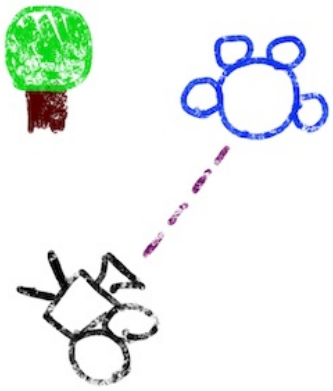


Pretty cool, right? Well, there's a problem with this approach. What happens if the avatar starts cartwheeling or flipping? Try it yourself (remember that we're using the `C` and `F` keys for this)!

The avatar appears to stay still, but everything else starts spinning! See the figure.



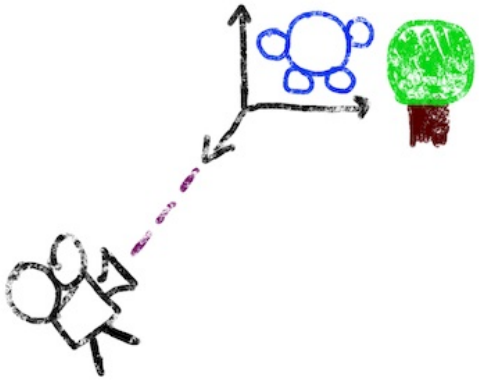
This is because the camera is stuck on the invisible selfie stick that's attached to the avatar. If the avatar spins, the camera goes right along with it.



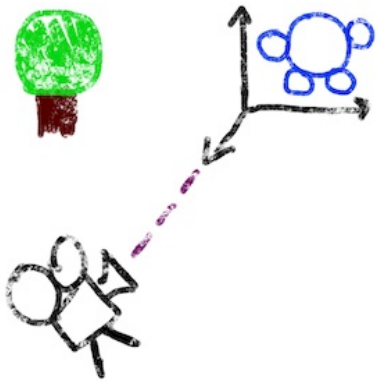
That's not quite what we want. Instead of locking the camera on the avatar, what we really want is to lock the camera on the avatar's *position*.

In 3D programming there is no easy way to reliably lock something to just the position of another thing. But all is not lost.

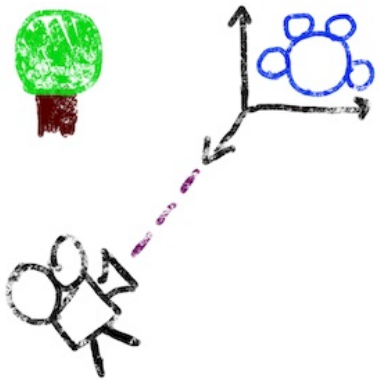
We'll add an avatar position marker to the game as shown in the figure.



If we lock both the camera and the avatar to this marker, then moving the marker moves both the avatar and the camera.



But, more importantly, when the avatar does cartwheels, the camera doesn't move. The avatar is cartwheeling, but the marker doesn't spin. Since the marker is not spinning, the camera doesn't spin either.



In 3D programming, this marker is just a marker. It should be invisible. So we don't want to use meshes or geometries for this. Instead we use `Object3D`. Let's

add the following code before the avatar-generated code, just after **START CODING ON THE NEXT LINE**:

```
var marker = new THREE.Object3D();
scene.add(marker);
```

Now we change the avatar so that it's added to the **marker** instead of the scene:

```
var avatar = new THREE.Mesh(body, cover);
» marker.add(avatar);
```

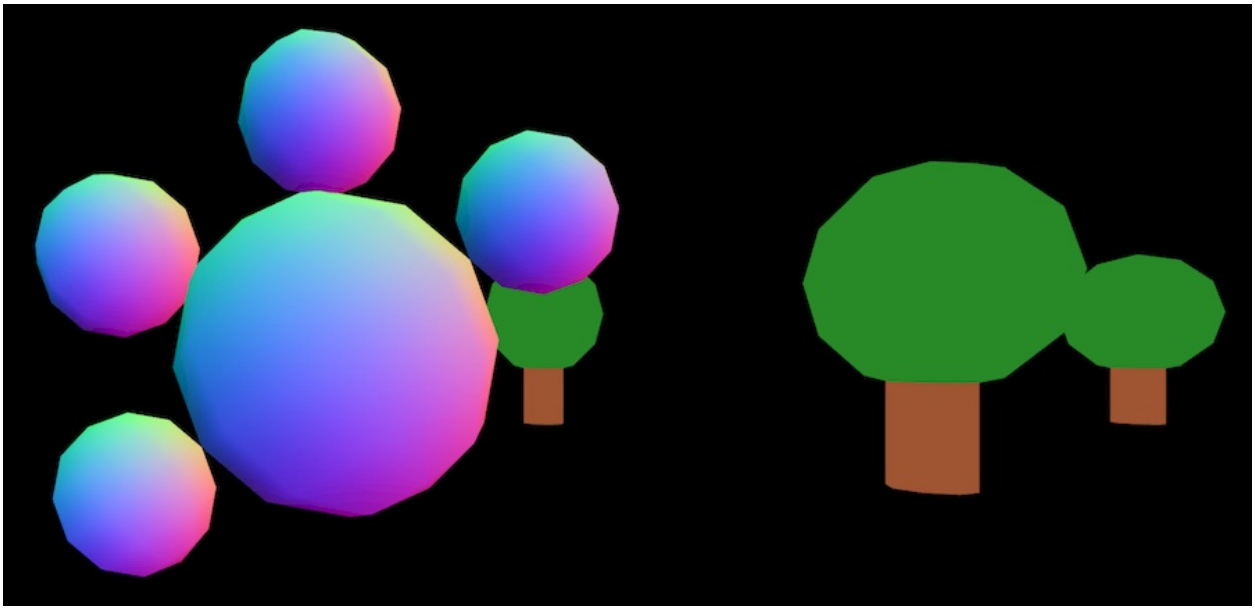
We also need to change how the camera is added. Instead of adding the camera to the avatar, we add it to the marker.

```
marker.add(camera);
```

The last thing we need to change is the keyboard event listener. Instead of changing the position of the avatar, we have to change the position of the marker.

```
document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;
» if (code == 'ArrowLeft') marker.position.x = marker.position.x - 5;
» if (code == 'ArrowRight') marker.position.x = marker.position.x + 5;
» if (code == 'ArrowUp') marker.position.z = marker.position.z - 5;
» if (code == 'ArrowDown') marker.position.z = marker.position.z + 5;

  if (code == 'KeyC') isCartwheeling = !isCartwheeling;
  if (code == 'KeyF') isFlipping = !isFlipping;
}
```

With that, we can move the avatar's position with the keyboard, but when we flip or cartwheel, the camera stays upright.



The Code So Far

If you'd like to double-check your code so far, compare it to the code in *Code: Moving Avatars*.

What’s Next

We covered some very important skills in this chapter. Events like these keyboard events are extremely important in JavaScript. We’re going to see events again. And we’ll group objects like this over and over as our gaming skills improve. Grouping simplifies moving things together, as well as twisting, turning, growing, and shrinking things together.

If you’re eager to continue working on our avatar, skip ahead to Chapter 6, *Project: Moving Hands and Feet*. If you do skip ahead, don’t forget to come back to the next chapter, which explores JavaScript functions more. We’re already using functions to make a forest, to animate, and to listen for events. There’s tons more to learn about functions. And if that doesn’t make you want to read the functions chapter, maybe this will: we create a hundred planets and add flight controls to fly around them!

Really, it’s pretty cool.

Footnotes

[7] https://en.wikipedia.org/wiki/Web_colors

When you're done with this chapter, you will

- *Understand a super-powerful tool (functions) for programmers*
- *Be able to tell stories and calculate things with functions*
- *Build 100 planets and fly among them*

Chapter 5

Functions: Use and Use Again

Computer programs are a little like stories.

> Once upon a time, there were 100 special planets. Each planet was a different color and a different size. They were scattered throughout known space. And there was only one ship fast enough—and one pilot crazy enough—to reach them all.

By now, we have some of the know-how needed to write that story in JavaScript code. But where do we start? How could we organize all that? Are we really going to have to type all the code for 100 planets?

The programming answer for those questions is a nifty tool called a *function*. We briefly talked about functions in Chapter 2, *Debugging: Fixing Code When Things Go Wrong* and Chapter 4, *Project: Moving Avatars*. Now we'll explore how they work.

Programming with functions—especially in JavaScript—can get complicated. Seriously complicated. A lot of programming power is involved with functions, which is partly why they get so complicated. But the basics are pretty easy. In this book, we'll stick with the basics, using functions to do three things:

1. Tell part of the story (build the player, move an avatar when a key is pressed)
2. Do something over and over (make four trees, make 100 planets)
3. Calculate values for use in #1 and #2 (and elsewhere)

As we explore functions in this chapter, we'll talk mostly about #2 and #3. We'll use functions to tell parts of stories—and organize them later. To start exploring, let's try to create 100 planets.

Getting Started

Create a new project in the 3DE Code Editor. Use the **3D starter project** template and call it **Planet Functions**.

The first step in creating 100 planets is to create one planet. After the line that says **START CODING ON THE NEXT LINE**, add the following:

```
var shape = new THREE.SphereGeometry(50);
var cover = new THREE.MeshBasicMaterial({color: 'blue'});
var planet = new THREE.Mesh(shape, cover);
planet.position.set(-300, 0, 0);
scene.add(planet);
```

At this point in the book, we have a good idea what that code will do before we even start typing. It creates a somewhat large ball, wraps it in blue material, and moves it off to the left side of the screen.

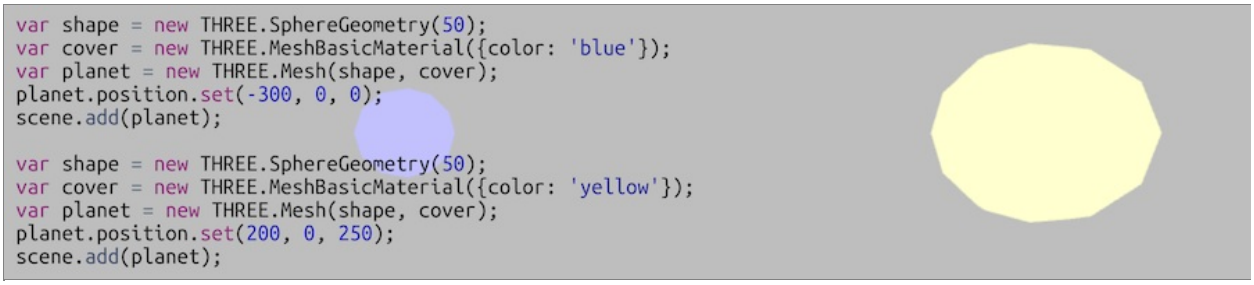
That's one planet.

Skip a line, then add this code:

```
var shape = new THREE.SphereGeometry(50);
var cover = new THREE.MeshBasicMaterial({color: 'yellow'});
var planet = new THREE.Mesh(shape, cover);
planet.position.set(200, 0, 250);
scene.add(planet);
```

This adds a second planet (a yellow one) to the right and a little forward.

That's two planets.



If we have to add ninety-eight other planets to the scene like this, we're going to be typing for a long time. And even after all that typing, we still have to work on other cool stuff like flying spaceships.

Back in Chapter 3, *Project: Making an Avatar*, we used a function to avoid having to repeat the same process for creating a tree four times. So you can probably guess our next step.

Basic Functions

Let's begin by defining a function named `makePlanet()`. After the code that we already have for the two planets, add the following:

```
function makePlanet() {
  var size = 50;
  var x = 0;
  var y = 200;
  var z = 0;
  var surface = 'purple';

  var shape = new THREE.SphereGeometry(size);
  var cover = new THREE.MeshBasicMaterial({color: surface});
  var planet = new THREE.Mesh(shape, cover);
  planet.position.set(x, y, z);
  scene.add(planet);
}
```

This is a nice example of a do-something-over-and-over function. As the name suggests, this will create planets over and over. One hundred planets to be precise.

The body of the function—all of the code in between the opening and closing curly braces—looks similar to the code for the first two planets. The difference is that we assign some values at the top of the function body. Assigning values like this is pretty common in functions—you'll see why shortly.

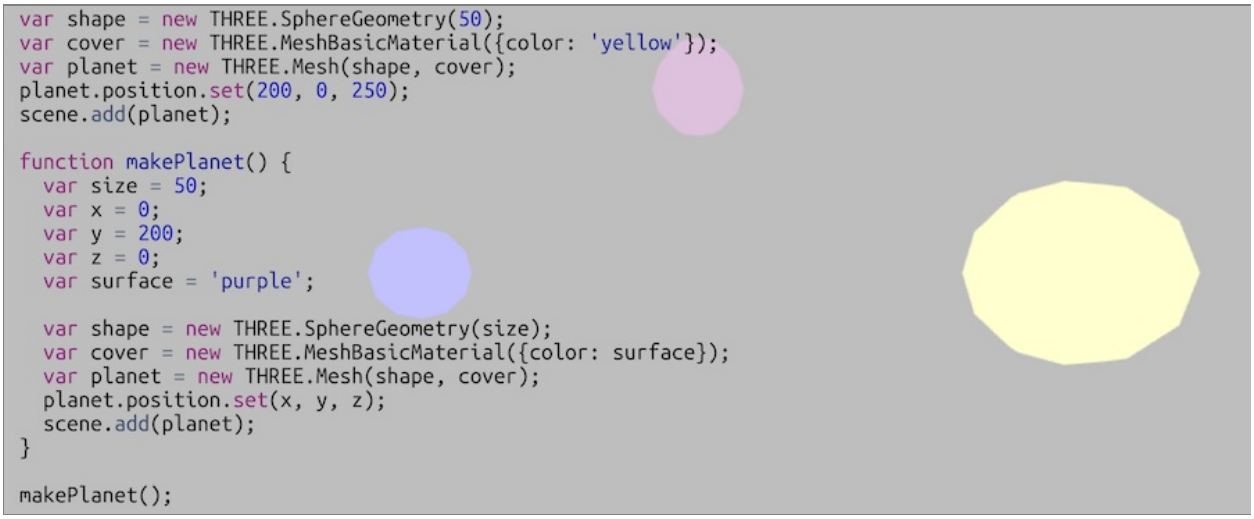
But first, add the following line after the `makePlanet()` function—after the ending curly brace of the function:

```
makePlanet();
```

A line of code like that is *calling* the function—it's asking the function to run. After we define a function, we can call it whenever we like by typing the name and putting parentheses after it.

If everything is typed correctly, you will now see a purple planet above the first

two planets as shown in the figure.



```
var shape = new THREE.SphereGeometry(50);
var cover = new THREE.MeshBasicMaterial({color: 'yellow'});
var planet = new THREE.Mesh(shape, cover);
planet.position.set(200, 0, 250);
scene.add(planet);

function makePlanet() {
  var size = 50;
  var x = 0;
  var y = 200;
  var z = 0;
  var surface = 'purple';

  var shape = new THREE.SphereGeometry(size);
  var cover = new THREE.MeshBasicMaterial({color: surface});
  var planet = new THREE.Mesh(shape, cover);
  planet.position.set(x, y, z);
  scene.add(planet);
}

makePlanet();
```

If you do not see that, you will have to check for errors in the code editor and in the JavaScript console as described in Chapter 2, *Debugging: Fixing Code When Things Go Wrong*.

Once `makePlanet()` is working, we have a way to create as many planets as we like. But we're not quite done. Add a second call to `makePlanet()` as shown:

```
    makePlanet();
» makePlanet();
```

After typing that code, nothing seems to have changed. We still have three planets. What's going on?

It turns out that everything is working; it's just not working as expected. We *do* have four planets. But `makePlanet()` always puts its planets at the same place—at `(0, 200, 0)`.

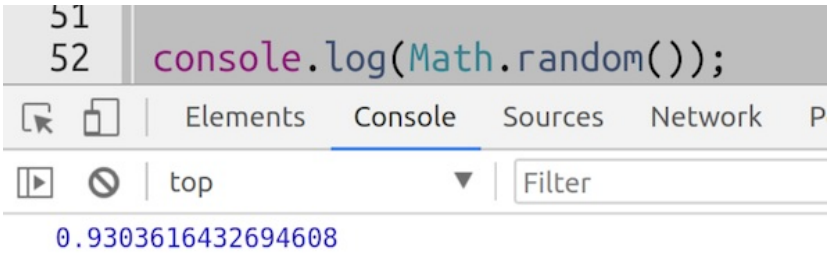
The story we're trying to tell is that planets are *scattered* throughout space. That's another way of saying that we want planets with random positions.

We can get a random number with a function that's built right into JavaScript: `Math.random()`. Let's use `console.log()`—the function we first discovered in Chapter 2, *Debugging: Fixing Code When Things Go Wrong*—to explore how `Math.random()` works. Below the two calls to `makePlanet()`, add the following line

to print a random number to the JavaScript console:

```
console.log(Math.random());
```

Now open the JavaScript console as described in *Opening and Closing the JavaScript Console*. In the console, you should see a random decimal number between 0 and 1.



If you click the **UPDATE** button back in the code editor, you can watch the number change in the JavaScript console. It's never the same number twice. Sometimes it's close to 0. Sometimes it's close to 1. Sometimes it's right in between.

That can be really helpful. Except we don't want to scatter planets between 0 and 1. That would put all of our planets very close to the center of the scene. We want them scattered randomly between 0 and 1000. To do that we could multiply `Math.random()` by 1000 every time we use `Math.random()`. But that's a lot of work. We're lazy programmers, right?

Let's use functions to keep us lazy.

Functions that Return Values

Let's write a function `r()` that returns a number between 0 and some number of our choosing. Add the following function below the `console.log()` statement from the previous section:

```
function r(max) {
  if (max) return max * Math.random();
  return Math.random();
}
```

To see how that works, let's follow that function with four more `console.log()` statements:

```
var randomNum = r();
console.log(randomNum);

randomNum = r(100);
console.log(randomNum);

console.log(r(100));
console.log(r(100));
```

With that, we should have a total of five values logged in the JavaScript console.

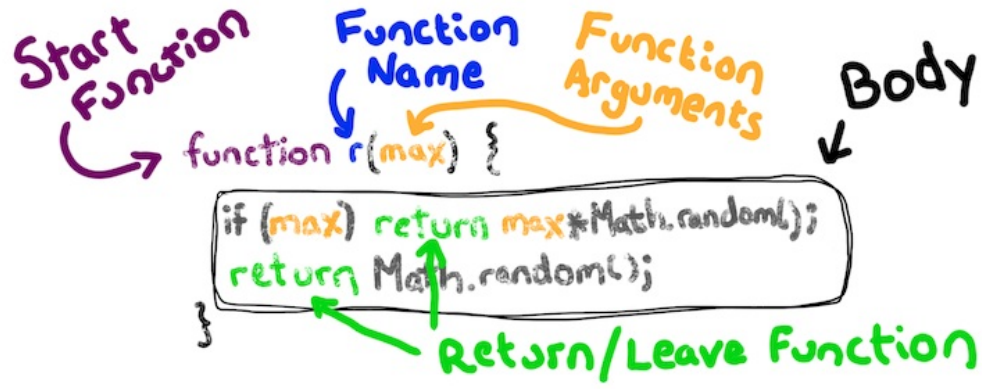
```
0.19286052818077604
0.8719966146656606
9.877757586253711
90.61991645733924
84.37073844604335
```

The first is the value of `Math.random()` from the previous section—it should be a decimal number between 0 and 1. The next number is returned from `r()`—it should also be between 0 and 1. The next three are the result of calling `r(100)`—all three numbers should be between 0 and *100*.

That `r()` function is going to be perfect for our planets scattered throughout space.

But how does it work?

We can diagram a function as:



The various parts of a function are the following:

Function Start
: A function definition always starts with the word `function`—that's how JavaScript knows it's looking at a function instead of other kinds of code.

Name
: We use a function by calling its name—`r()` or `makePlanet()`.

Arguments
: These are values we send to the function. We can pass zero, one, or more values to a function.

- A zero-argument function, like `makePlanet()`, has nothing between the parentheses.
- A one-argument function, like `r(number)`, has the one argument between the parentheses.
- A two-or-more argument function lists the arguments with commas: `playSong(title, howLong)`.

Body

Everything inside the curly braces.

Return / Leave Function

The `return` statement does two things. First, it immediately leaves the function—any code below the `return` statement gets ignored. Second, it sends the value after the `return` statement back to the code that is calling the function.

OK, now back to our `r()` function. The first line of the function body says that *if* we give it a `max` value, then we *return* the product of multiplying `max` and `Math.random()`. We give a function an argument value by calling it with a value like `r(2)`, `r(100)`, or `r(1000)`.

If we don't give `r()` a `max` value—if we call `r()` with nothing between the parentheses—then we skip the `return` on the first line. With no `max` value, the function moves onto the second line where it returns `Math.random()` without any multiplication.

We can take the return value from functions like `r()` and assign them to values—like we did with `randomNum`. We can also log them directly to the console as we did with `console.log(r())`.

Using Functions

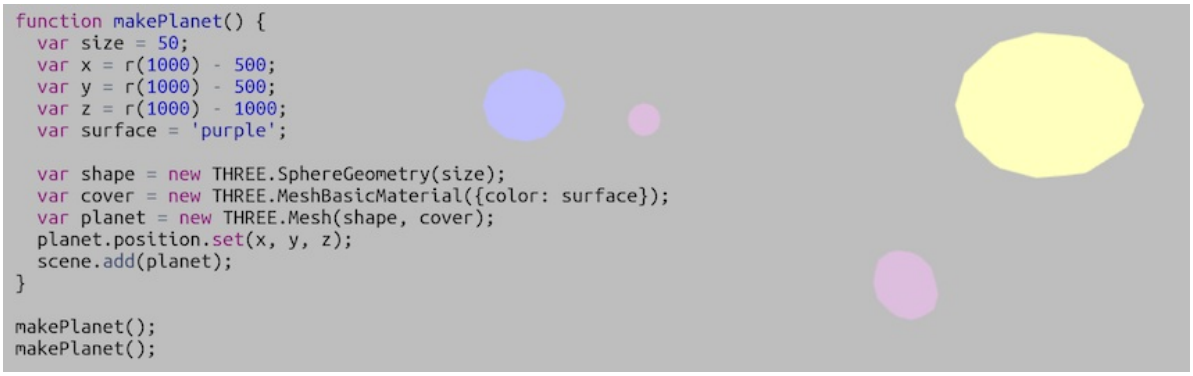
Now that we understand what `r()` does, let's put it to use. Back in the `makePlanet()` function, change the values of `x`, `y`, and `z` as shown.

```
  function makePlanet() {
    var size = 50;
»   var x = r(1000) - 500;
»   var y = r(1000) - 500;
»   var z = r(1000) - 1000;
    var surface = 'purple';

    var shape = new THREE.SphereGeometry(size);
    var cover = new THREE.MeshBasicMaterial({color: surface});
    var planet = new THREE.Mesh(shape, cover);
    planet.position.set(x, y, z);
    scene.add(planet);
  }
```

Calling `r(1000)` will give us a random number between 0 and 1000. For `x`, we subtract `500` from `r(1000)` to give us a random number between –500 (all the way left) and 500 (all the way right). If you're unsure about this math, send `r(1000) - 500` to `console.log()` and click the `UPDATE` button a couple of times to see what goes in the JavaScript console.

We do the same thing with `y`, to randomly place the planets between -500 (down) and 500 (up). We subtract `1000` from `r(1000)` for `z` so that the planets are well away from our starting position.

The result should be some randomly scattered planets!



Well, two randomly scattered planets plus the blue and yellow planets we placed the hard way. Thanks to functions, it's easy to call `makePlanet()` as many times as we like. We are definitely not going to type `makePlanet()` 100 times—we're lazy, remember? Instead, we're going to create a programming "loop" to do it. The following code will loop over `makePlanet()` 100 times for us. Add it below the two other calls to `makePlanet()`.

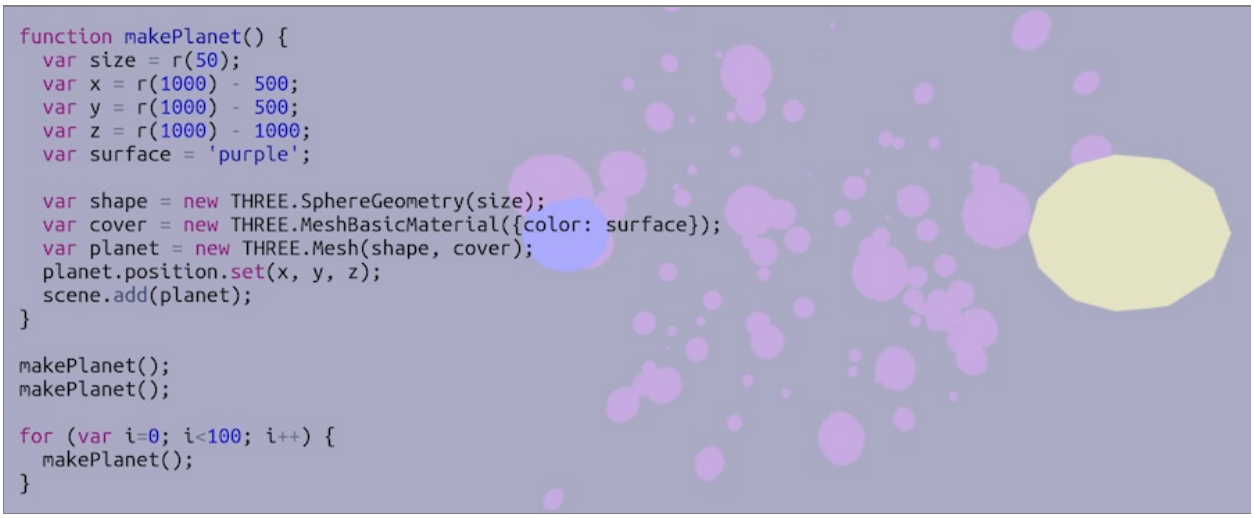
```
for (var i=0; i<100; i++) {
  makePlanet();
}
```

Don't worry too much about how loops work—we'll see them again in Chapter 7, *A Closer Look at JavaScript Fundamentals*.

While we're making changes, we should also change the size of the planet to be a random number between 0 and 50. That's easy thanks to our `r()` function. Change the `size` value at the beginning of `makePlanet()` to be `r(50)`.

```
function makePlanet() {
» var size = r(50);
  var x = r(1000) - 500;
  var y = r(1000) - 500;
  var z = r(1000) - 1000;
  var surface = 'purple';

  var shape = new THREE.SphereGeometry(size);
  var cover = new THREE.MeshBasicMaterial({color: surface});
  var planet = new THREE.Mesh(shape, cover);
  planet.position.set(x, y, z);
  scene.add(planet);
}
```

With that, the size of each planet is a random number between 0 and 50. We have some big planets and some small planets—much more realistic!



```
function makePlanet() {
  var size = r(50);
  var x = r(1000) - 500;
  var y = r(1000) - 500;
  var z = r(1000) - 1000;
  var surface = 'purple';

  var shape = new THREE.SphereGeometry(size);
  var cover = new THREE.MeshBasicMaterial({color: surface});
  var planet = new THREE.Mesh(shape, cover);
  planet.position.set(x, y, z);
  scene.add(planet);
}

makePlanet();
makePlanet();

for (var i=0; i<100; i++) {
  makePlanet();
}
```

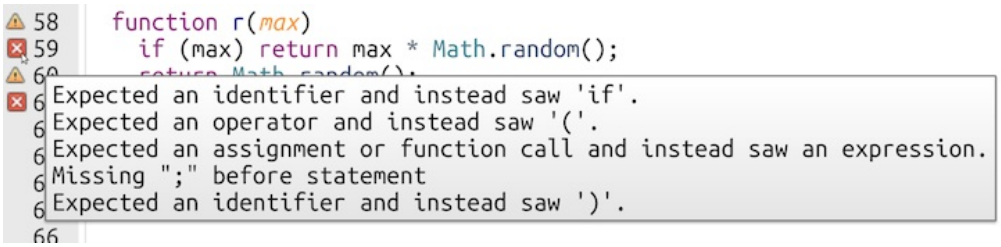
More importantly, we can begin to see the power of functions. Think back to the first two planets we created in this chapter. We wrote ten lines of code to create two planets. If we kept doing that for 100 planets, we would have written *500* lines of code. Thanks to functions, we wrote less than 20 lines of code. Yay!

Breaking Functions

You have some idea of how powerful functions are. But beware for they're easy to break in lots of different ways. We saw some of this back in Chapter 2, *Debugging: Fixing Code When Things Go Wrong*. Now that we understand functions better, it's worth taking another look at breaking them.

The most common thing to do is forget a curly brace:

```
function r(max)
  if (max) return max * Math.random();
  return Math.random();
}
```

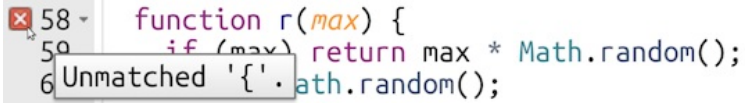


It's hard to miss that error! The error messages don't say that we forgot the curly brace, which would be more helpful. But it's pretty clear that *something* was expected before the `if` statement. Once we know that, it should be easy to track down the error.

What happens if we put the curly brace back, but remove the curly brace after the `return` statement?

```
function r(max) {
  if (max) return max * Math.random();
  return Math.random();
```

The code editor is definitely able to figure that problem out!



So pay attention to the code editor warnings when you are programming with functions.

Challenge

Try to figure out where to find the error messages for the following examples. *Hint*: As in *Debugging in the Console*, some of these may only show up in the JavaScript console.

Forgotten parentheses around the argument:

```
function r max {
  if (max) return max * Math.random();
  return Math.random();
}
```

Forgotten function argument:

```
function r() {
  if (max) return max * Math.random();
  return Math.random();
}
```

Wrong variable name inside the function:

```
function r(max) {
  if (number) return number * Math.random();
  return Math.random();
}
```

Function called with the wrong name:

```
function r(max) {
  if (max) return max * Math.random();
  return Math.random();
}

var randomNum = randomNumber();
console.log(randomNum);
```

Wow! There sure are a lot of ways to break functions. Believe me when I tell you that you'll break functions in these and many other ways. But that's OK. Each time you fix broken code, you'll learn something new. And learning new stuff is the most important thing you can do to become a great programmer.

	Great Programmers Break Things All the Time
	Because they break things so much, they are *really* good at fixing things. This is another skill that makes great programmers great.

Be sure to fix the `r()` function if you want to try some of the bonus code that follows.

Bonus #1: Random Colors

Purple is a great color. But if you prefer more variety, it would be nice to *randomly* choose color. The problem is that the colors we've seen so far have been named colors, like blue.

```
  var shape = new THREE.SphereGeometry(50);
» var cover = new THREE.MeshBasicMaterial({color: 'blue'});
  var planet = new THREE.Mesh(shape, cover);
  planet.position.set(-500,0,0);
  scene.add(planet);
```

Named colors like blue, yellow, and purple are hard to randomize. Happily, you can make colors other ways in programming. One of the most common ways uses a number between 0 and 1 to specify how much red, green, and blue go into a color.

Here are some common colors in red, green, blue (also known as RGB) format. You don't need to type these—they're just interesting to look through.

```
var red   = new THREE.Color(1, 0, 0);
var blue  = new THREE.Color(0, 1, 0);
var green = new THREE.Color(0, 0, 1);
var cyan  = new THREE.Color(1, 1, 0);
var white = new THREE.Color(1, 1, 1);
var black = new THREE.Color(0, 0, 0);
var grey  = new THREE.Color(0.5, 0.5, 0.5);
```

Numbers between 0 and 1? Our `r()` function can do that! If we don't pass any arguments to `r()`, it will return `Math.random()`, which generates a number between 0 and 1.

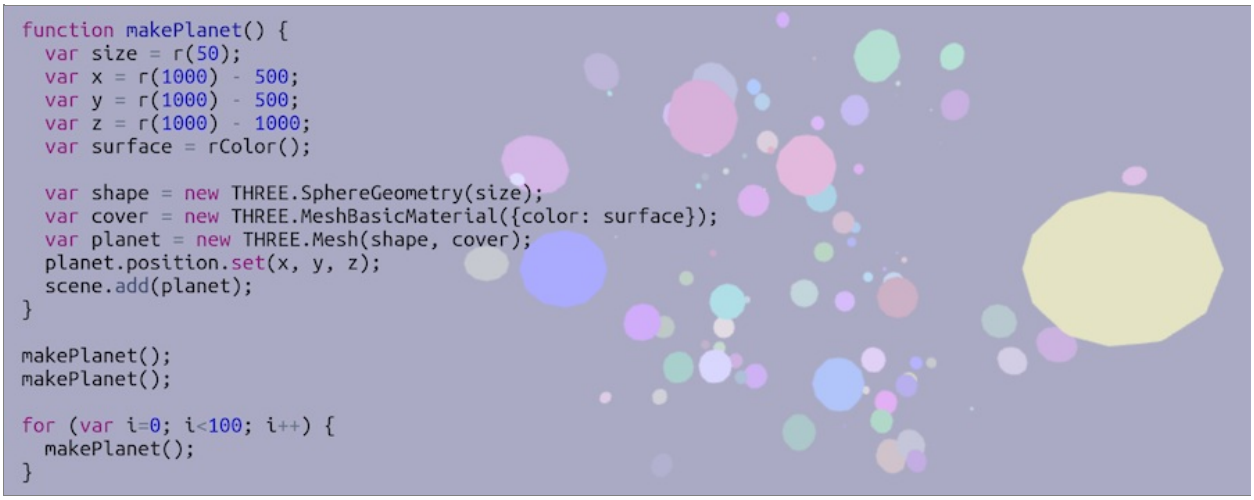
So below the `console.log()` statements that we used to test the `r()` function, add a new `rColor()` function.

```
function rColor() {
  return new THREE.Color(r(), r(), r());
}
```

Then, back in the body of `makePlanet()`, change the `surface` from `'purple'` to `rColor()` instead.

```
  function makePlanet() {
    var size = r(50);
    var x = r(1000) - 500;
    var y = r(1000) - 500;
    var z = r(1000) - 1000;
»   var surface = rColor();

    var shape = new THREE.SphereGeometry(size);
    var cover = new THREE.MeshBasicMaterial({color: surface});
    var planet = new THREE.Mesh(shape, cover);
    planet.position.set(x, y, z);
    scene.add(planet);
  }
```

Thanks to another simple function, we now have 100 planets, of various shapes and colors, scattered throughout space!



Bonus #2: Flight Controls

Where's the fun in scattering a bunch of planets around if you can't fly through them, right?

Fly controls are much more complicated than they might seem—very much more complex than the controls we're building for our avatar. So we'll load in new controls instead of writing our own. All the way at the top of our code, add a new `<script>` tag to load in `FlyControls.js`.

```
  <body></body>
  <script src="/three.js"></script>
» <script src="/controls/FlyControls.js"></script>
```

We'll talk more about loading code like that in Chapter 9, *What's All That Other Code?*.

Loading the code is only half of the job. The other half is using it. To do that, add the following below the `rColor()` function, which should be at the bottom of our code.

```
var controls = new THREE.FlyControls(camera);
controls.movementSpeed = 100;
controls.rollSpeed = 0.5;
controls.dragToLook = true;
controls.autoForward = false;

var clock = new THREE.Clock();
function animate() {
  var delta = clock.getDelta();
  controls.update(delta);

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

To use the controls, hide your code. The following keyboard keys let you fly or spin in different directions.

Motion	Direction	Keys
Move	Forward / Backward	W / S
Move	Left / Right	A / D
Move	Up / Down	R / F
Spin	Clockwise / Counterclockwise	Q / E
Spin	Left / Right	Left Arrow / Right Arrow
Spin	Up / Down	Up Arrow / Down Arrow

Hint: It's pretty fun to press `Q` and `W` at the same time. But you might get a little dizzy!

Experiment with the `movementSpeed` and `rollSpeed` values as you like. The values used should work fairly well with our planets. The `autoForward` value moves forward unless you press a key. If `dragToLook` is set to `false`, then the scene moves whenever you move the mouse.

Ever since we spun our shapes in Chapter 1, *Project: Creating Simple Shapes*, we've been using the `animate()` function. Here, we tell the fly controls to update their position in the body of the animate function. A clock *delta* is the amount of time that's passed since the last time the controls updated the scene. The fly controls use this delta to make the updates nice and smooth. Without that, flying might look jittery or stutter at times.

Speaking of the `animate()` function, it's a function like the ones that we just wrote. Take a look.

```
① function animate() {
    var delta = clock.getDelta();
    controls.update(delta);

②   renderer.render(scene, camera);
③   requestAnimationFrame(animate);
  }
④ animate();
```

① Defines `animate()` like any other function. It's called like any other function at ④.

② Tells JavaScript to update the 3D scene.

③ This is something we haven't seen yet—it's sending the `animate()` function itself as an argument to the `requestAnimationFrame()` function that's built into browsers!

The call to the `requestAnimationFrame()` function is pretty wild and it hints at how complex functions can get. It's built into browsers so that browsers can call a function whenever the browser isn't busy doing something else (such as updating something on the page, loading new information from a web server, or running another function). We're telling the browser to call our `animate()` function again as soon as it's not busy. And when that happens, the `animate()` function will get a clock delta, update the fly controls, re-render the scene, and… ask the browser to do it all again as soon as the browser is ready!

Without something like `requestAnimationFrame()`—and the ability to pass a function as an argument, animations and 3D programming on the web would be much harder. Almost everything would stutter or completely stop at times. So yay for `requestAnimationFrame()` and functions!

The Code So Far

In case you'd like to double-check the code in this chapter, it's included in *Code: Functions: Use and Use Again*.

What’s Next

Functions are very powerful tools for JavaScript programmers. We saw two good uses for functions. Over-and-over functions like `makePlanet()` save a lot of coding. Calculating values like the random `r()` function also helps to make our coding lives easier. We even saw that the `animate()` function we’ve been using everywhere is a kind of tells-a-story function. It tells the story of animation in our games.

Most importantly though, we put all of these together to build a cool little universe. And we’re still just scratching the surface!

As you’ll see shortly, we’ll use functions a lot in the upcoming chapters. Let’s get started in the next chapter as we teach our avatar how to move its hands and feet!

When you're done with this chapter, you will

- *Understand some important math for 3D games*
- *Know how to swing objects back and forth*
- *Have an avatar that looks like it's walking*

Chapter 6

Project: Moving Hands and Feet

When we last saw our avatar in Chapter 4, *Project: Moving Avatars*, it was moving around pretty well. But it was a little… stiff. Even when the body moved, the hands and the feet stayed still. In this chapter we'll give our avatar a little more life.

Getting Started

We're again building on work from previous chapters. Since we did so much work to get the avatar moving in Chapter 4, *Project: Moving Avatars*, let's make a copy of that project to work on in this chapter.

If it's not already open in the 3DE Code Editor, open the project that we named My Avatar: Keyboard Controls. To make a copy of it, click the menu button and choose MAKE A COPY from the menu.

Name the project My Avatar: Moving Hands and Feet and click the SAVE button.

With that, we're ready to start adding life to our avatar!

Moving a Hand

Let's start with a hand. Recall from previous chapters that hands and feet are just balls that stick out from the head. We built the right hand in JavaScript with this:

```
var rightHand = new THREE.Mesh(hand, cover);
rightHand.position.set(-150, 0, 0);
avatar.add(rightHand);
```

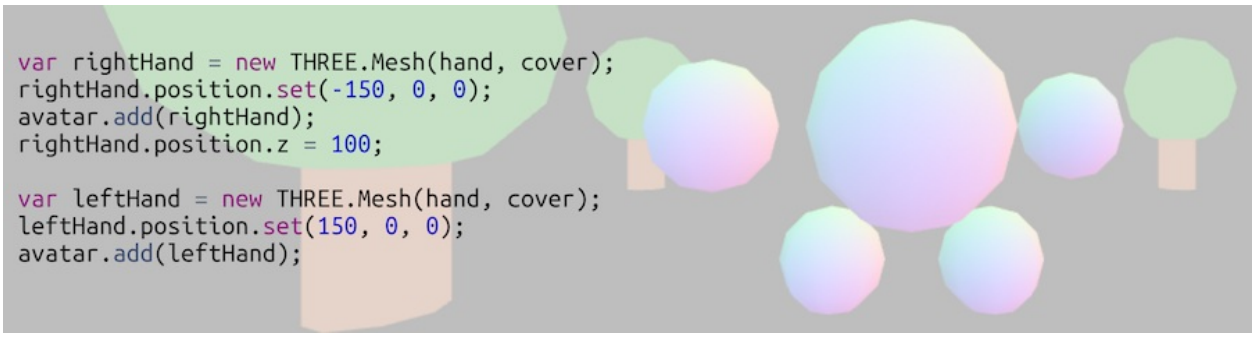
As you know, the three numbers we use to set the position of the hand are the X position (left/right), the Y position (up/down), and the Z position (in/out). In the case of the right hand, we placed it –150 from the center of the avatar.

In addition to setting all three numbers for the position, we can change just one of the positions by updating `position.x`, `position.y`, or `position.z`. To move the right hand forward (toward the viewer), add the `position.z` line shown:

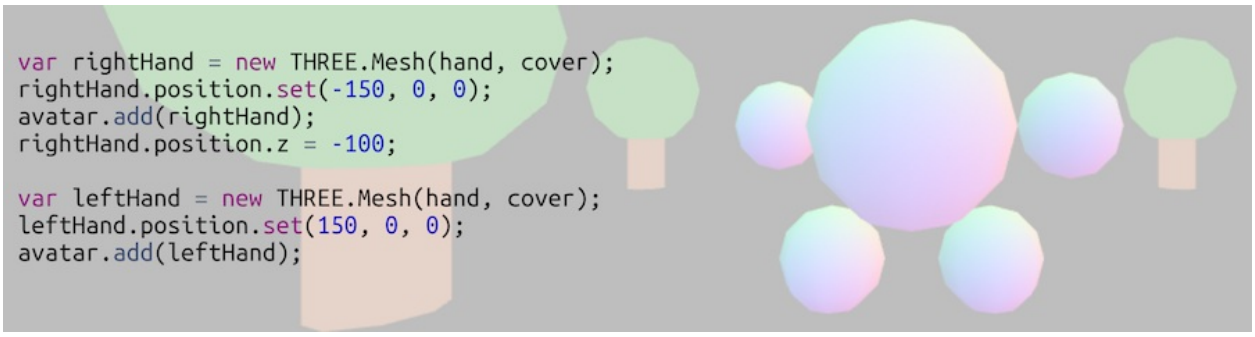
```
var rightHand = new THREE.Mesh(hand, cover);
rightHand.position.set(-150, 0, 0);
avatar.add(rightHand);
rightHand.position.z = 100;
```

Change the value of `position.z` from `100` to `-100`. What happens? What happens if you keep changing between `100` and `-100`?

When `z` is 100, the hand is moved forward.



When `z` is `-100`, the hand has moved backward so that we almost can't see the hand behind the body.



And when you change `position.z` back and forth between -100 and 100, it's almost like the hand is swinging back and forth. Congrats! You just learned a famous animation technique!

In some games, it's enough to move a thing from one place to another place to make it seem like it's moving. But we can do better in our game.

Start by deleting the line that sets the `position.z`. We don't want to set it once. We want to animate it. That means that we'll be working in the `animate()` function again. After Chapter 4, *Project: Moving Avatars*, we're already animating cartwheels and flips in the function.

```
var isCartwheeling = false;
var isFlipping = false;
function animate() {
  requestAnimationFrame(animate);
  if (isCartwheeling) {
    avatar.rotation.z = avatar.rotation.z + 0.05;
  }
  if (isFlipping) {
    avatar.rotation.x = avatar.rotation.x + 0.05;
  }
  renderer.render(scene, camera);
}
animate();
```

That works well, but before we start animating more stuff in the `animate()` function, let's tidy things up a bit.

Quick Code Cleanup

A lot is happening in our `animate` function—so much that it's a little messy. This

mess can make it hard to read our code. We'll be adding even more stuff to `animate`. Unless we do something, that `animate` function is going to get ugly.

So let's perform a little code cleanup. Just below the `animate()` function, create an `acrobatics` function. That function will do the flipping and cartwheeling, so you can cut and paste the `if (isCartwheeling)` and `if (isFlipping)` from `animate()` into this new `acrobatics()` function. Then we can call `acrobatics` from within `animate`, making it easier to read.

```
var isCartwheeling = false;
var isFlipping = false;

function animate() {
  requestAnimationFrame(animate);
»   acrobatics();
  renderer.render(scene, camera);
}
animate();

function acrobatics() {
  if (isCartwheeling) {
    avatar.rotation.z = avatar.rotation.z + 0.05;
  }
  if (isFlipping) {
    avatar.rotation.x = avatar.rotation.x + 0.05;
  }
}
```

Let's take a minute to make sure everything still works. Hide the code and press `C` and `F` to make sure cartwheeling and flipping still work. If something has gone wrong, check the JavaScript console!

If everything is working, show the code again. Now, we can add three more things in and around the `animate` function.

```
① var clock = new THREE.Clock();
var isCartwheeling = false;
var isFlipping = false;

function animate() {
```

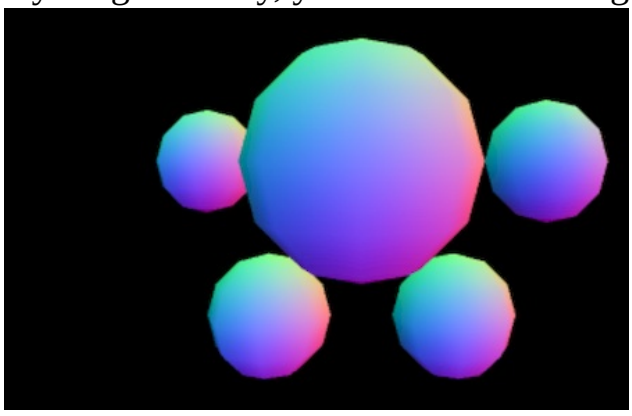
```
    requestAnimationFrame(animate);
②   walk();
    acrobatics();
    renderer.render(scene, camera);
  }
  animate();

③ function walk() {
    var speed = 10;
    var size = 100;
    var time = clock.getElapsedTime();
    var position = Math.sin(speed * time) * size;
    rightHand.position.z = position;
  }
```

① We'll use this 3D clock as a timer for our animation.

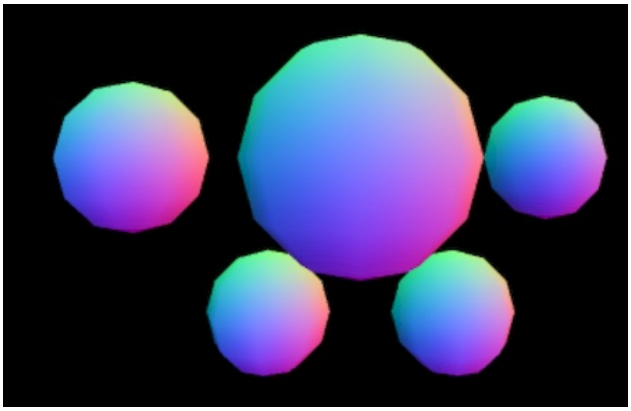
② In addition to performing acrobatics, now we'll also walk. After typing this, your code will break temporarily. That's OK! We'll fix it in the very next step.

③ This is the function that moves the hands and feet. It goes after the `animate()` function and above the `acrobatics()` function. There's something special in this function!

If you've typed in everything correctly, you should see the right hand of the avatar jiggling back:



and forth as shown in figure.



And it should be moving pretty quickly.

Cool! But how does that work? Well, it's all thanks to the power of a beautiful thing called… a *sine*. It's crazy useful in 3D programming.

As you might guess from the name, `Math.sin` has something to do with math. A sine makes a number between -1 and 1. More importantly, a sine function makes a number that smoothly changes back and forth between -1 and 1 as another number gets bigger. That might not seem like a big deal. It is.

In our code, the sine function is using time. Since time is always increasing, the sine function keeps moving back and forth between -1 and 1. That's perfect for moving hands and feet back and forth!

On that same line, we also do some multiplication. In JavaScript, the asterisk character (`*`) is used to multiply numbers. We multiply time by `speed` amount. We multiply the sine result by `size`.

	Let's Play!
	Experiment with the numbers inside `walk`. If you change `speed` from 10 to 100, what happens? If you change `size` from 100 to 1000, what happens? Try using `position.x` or `position.y` instead of `position.z`. Try changing `position.y` and `position.z` at the same time. Once you have a feel for those numbers, try to get

the other hand and the feet all moving at the same time. Play!

Swinging Hands and Feet Together

How did it work? Were you able to get all of the hands and feet swinging back and forth? If not, don't worry—it's a little tricky.

If you tried moving the hands and feet in the same way, you might have noticed that our avatar is moving awfully strangely. Both feet and both hands move forward at the same time. And then both feet and both hands swing back at the same time. No one walks like that in real life.

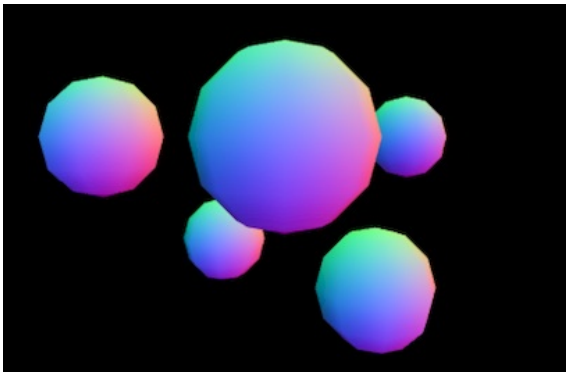
When you walk, one foot is in front and the other is behind. In avatar terms, one foot is in the positive Z direction, while the other is in the negative Z direction:

```
  var speed = 10;
  var size = 100;
  var time = clock.getElapsedTime();
  var position = Math.sin(speed * time) * size;

» rightFoot.position.z = -position;
» leftFoot.position.z  =  position;
```

People also usually move their right hand forward when their *left* foot is forward. And if the right hand is forward, then the left hand should be back. We can make our avatar do this with the following code:

```
  function walk() {
    var speed = 10;
    var size = 100;
    var time = clock.getElapsedTime();
    var position = Math.sin(speed * time) * size;

»   rightHand.position.z = position;
»   leftHand.position.z = -position;
    rightFoot.position.z = -position;
    leftFoot.position.z = position;
  }
```

With that, our avatar's hands and feet should be swinging back and forth in a nice walking motion.



Walking When Moving

Right now, our avatar is constantly walking—even when we're not controlling it with our controls from Chapter 4, *Project: Moving Avatars*. Let's fix this problem.

First, let's add a way to track the direction in which the avatar is moving. Add four more lines above the `animate()` function, starting with the value of `isMovingRight`:

```
var clock = new THREE.Clock();
var isCartwheeling = false;
var isFlipping = false;
var isMovingRight = false;
var isMovingLeft = false;
var isMovingForward = false
var isMovingBack = false;
```

That code says that our avatar is not moving at first. Until something happens, our avatar is not moving right, left, forward, or back—it is false that the avatar is moving in any of those directions.

Next, add a new function named `isWalking()`. It's going to tell us whether the avatar is walking. We'll add this code after the `walk()` function (and just above the `acrobatics()` function).

```
function isWalking() {
  if (isMovingRight) return true;
  if (isMovingLeft) return true;
  if (isMovingForward) return true;
  if (isMovingBack) return true;
  return false;
}
```

For `isWalking()` to tell us whether the avatar is moving, it needs to return a value. If the avatar is moving right, it returns `true`. If it's moving left, forward, or back, it also returns `true`. If it's not doing any of those, then our avatar is not moving

and it returns `false`.

Now, back in `walk()`, add a line to the top of the function:

```
  function walk() {
»   if (!isWalking()) return;

    var speed = 10;
    var size = 100;
    var time = clock.getElapsedTime();
    var position = Math.sin(speed * time) * size;
    rightHand.position.z =  position;
    leftHand.position.z  = -position;
    rightFoot.position.z = -position;
    leftFoot.position.z  =  position;
  }
```

This line of code means *if the avatar is not walking, then return immediately from the function.* Unlike in `isWalking()`, we're not returning a value like `true` or `false`. We're just leaving `walk()` without doing anything else. That is, if the avatar is not walking, then leave the `walk` function without running any of the code that makes the avatar look like it's walking.

Once that's done, our avatar's hands and feet should stop moving. The values for `isMovingRight` and the others are `false`, and there's nothing in our code yet to change them. So `isWalking()` always returns `false`. And because `isWalking()` is always `false`, any time `walk()` gets called, it returns right away without doing anything.

We want the avatar's hands and feet to move when we press the keyboard controls. So, we need to update the `sendKeyDown` function. Add the lines shown. Be sure to include the curly braces for each of the four `if` statements!

```
  document.addEventListener('keydown', sendKeyDown);
  function sendKeyDown(event) {
    var code = event.code;
    if (code == 'ArrowLeft') {
      marker.position.x = marker.position.x-5;
①     isMovingLeft = true;
```

```
    }
    if (code == 'ArrowRight') {
      marker.position.x = marker.position.x+5;
②     isMovingRight = true;
    }
    if (code == 'ArrowUp') {
      marker.position.z = marker.position.z-5;
③     isMovingForward = true;
    }
    if (code == 'ArrowDown') {
      marker.position.z = marker.position.z+5;
④     isMovingBack = true;
    }
    if (code == 'KeyC') isCartwheeling = !isCartwheeling;
    if (code == 'KeyF') isFlipping = !isFlipping;
  });
```

① A left arrow means the avatar is moving left.

② A right arrow means the avatar is moving right.

③ An up arrow means the avatar is moving forward.

④ A down arrow means the avatar is moving backward.

As long as the player keeps one of these keys pressed down, the avatar will keep moving in the proper direction. JavaScript keyboard handlers just work like that. We've turned the movement controls on, but we still need to be able to turn them off. Since we used `keydown` to decide when a key is being pressed, you can probably guess how we'll decide when a key is let go.

After the last line of the `keydown` event-listener code—after the `});` line, add the following `keyup` event-listener code.

```
document.addEventListener('keyup', sendKeyUp);
function sendKeyUp(event) {
  var code = event.code;
  if (code == 'ArrowLeft') isMovingLeft = false;
  if (code == 'ArrowRight') isMovingRight = false;
  if (code == 'ArrowUp') isMovingForward = false;
  if (code == 'ArrowDown') isMovingBack = false;
```

```
}
```

With that, we can hide our code and try out our controls. We should be able to move our avatar with the arrow keys and see the avatar's hands and feet swing back and forth. When we let go of those keys, the avatar should stop walking.

Cool!

	Let's Play!
	If you're up for a challenge, let's aim for better acrobatics controls. Since we have code to listen for `keydown` and `keyup` events, try to make the cartwheels and flips start when the `C` or `F` key is pressed and stop when the `C` or `F` key is released. Do you think the controls are better this way? If so, leave them in there—it's your game!

The Code So Far

If you would like to double-check the code in this chapter, turn to *Code: Moving Hands and Feet*.

What's Next

We now have a new way to bring our avatars to life. Back in Chapter 4, *Project: Moving Avatars*, we were able to move the avatar around the scene and perform flips and cartwheels. In this chapter we made parts of the avatar move—making the avatar seem much more alive.

The new concept in this chapter was not a JavaScript thing or even a 3D thing. It was a math thing: `sine`. Even if you've learned about it in math class, I bet you didn't learn to use it like we did here!

One thing that our avatar still lacks is the ability to turn. Even when the avatar moves to the left or right, it continues to face forward. That's a bit odd, right? In Chapter 8, *Project: Turning Our Avatar*, we'll cover how to rotate the entire avatar.

But first it's time for a quick break to look a little more closely at JavaScript.

When you're done with this chapter, you will

- *Know what many of those JavaScript things are (like `var`)*
- *Be able to keep large lists of things (two different ways!)*
- *Understand 80% of JavaScript worth knowing*

Chapter 7

A Closer Look at JavaScript Fundamentals

Before we add more to our avatar, this is a good time to look closer at JavaScript. There's a lot in this chapter. Don't worry if it gets to be overwhelming. It's even OK if you only look through this chapter quickly your first time through the book. Just remember to come back at some point—this chapter has a lot that is important to know if you want to program well.

Like any other programming language, JavaScript was built so that both computers and people could understand it. Like any other programming language, it's just as likely that computers and people will be confused by it.

That's a solid programming joke. Sadly, most programmers have awful senses of humor. I'm considered hilarious. If you have better jokes than that, you'll be a welcome addition to the programming world! Anyway…

JavaScript programming describes things and what those things do, just like English and other languages. When we built our avatar, we used JavaScript to describe its head, hands, and feet. We also described how the 3D renderer should draw the scene in our browser. To put it all together, JavaScript has keywords and structures that both computers and humans can understand.

We've seen a lot of that so far, but haven't looked at it too closely. Let's do that now.

Getting Started

Instead of drawing and moving shapes in this chapter, we're going to explore the JavaScript programming language. We can do this in the JavaScript console, so let's start by opening that. Refer to *Opening and Closing the JavaScript Console*, if you don't remember how.

The JavaScript console is a web programmer's best friend. Modern browsers pack the JavaScript console with all kinds of features that help programmers improve network performance, security, memory usage, and lots more. In this book, we use it to debug our code and to experiment with JavaScript. We talked about debugging in Chapter 2, *Debugging: Fixing Code When Things Go Wrong*. In this chapter, we experiment. And when programmers experiment, we play!

First, a quick tip…

You Can Change Code in the Console (sort of)

If you make a mistake in your code and then press `Enter`, you can't just click on the mistake to change it.

```
> var name = "Alice';
Uncaught SyntaxError: Invalid or unexpected token
>
```

But, you can press the `Up` arrow key on your keyboard to bring up the last thing that you typed.

```
> var name = "Alice';
Uncaught SyntaxError: Invalid or unexpected token
> var name = "Alice";
```

Then you can fix the problems and press `Enter` to try again.

	Don't Type the Same Thing Over and Over in the JavaScript Console.

We programmers have to type code. But we programmers like tools that make typing easier. Use the up and down arrow keys to access and navigate the *history* of the code you type in the JavaScript console. Making small changes to code you already typed saves lots of time.

All right, let’s have some fun with JavaScript in the console!

Describing Things in JavaScript

Have you noticed how we introduce new things in JavaScript?

```
var speed = 10;
var title = '3D Game Programming for Kids';
var isCool = true;
```

The `var` keyword declares new things in JavaScript. It tells both the computer and the humans reading the code, “Get ready—something new is coming!”

The `var` Keyword

The `var` keyword is short for *variable*. A variable is a thing that can change.

```
> var i
< undefined
> i = 0;
< 0
> i = 1;
< 1
> i = 2;
< 2
```

First, the variable `i` is set to the value of 0. Then, it is set to 1. Last, the variable is set to the value of two. The value that the variable points to changes—it *varies*—as the code does stuff (or as we type changes in the console).

We always use `var` when we first introduce a variable in our code. Without it, JavaScript can get confused about which value gets set to a variable and when. In other words, not using `var` can make buggy code that is hard to fix. So *always* use `var` when you introduce a variable…

…Except in the JavaScript console. Code in the JavaScript console is for quick experiments, so there’s little chance to create bugs. When writing these quick experiments, programmers save time by not typing `var` *or* ending lines with semicolons.

The other reason to skip `var` in the JavaScript console is that the value being set is not reported. When you set a variable in the JavaScript console without `var`, the value is reported after pressing `Enter`.

```
> title = "3D Game Programming for Kids"
<· "3D Game Programming for Kids"
```

Weirdly, if you set a variable with `var` in the JavaScript console, the console reports the result as `undefined`.

```
> var title = "3D Game Programming for Kids"
<· undefined
```

It's *not* undefined though. Typing the variable name alone and pressing `Enter` will report its value.

```
> title
<· "3D Game Programming for Kids"
```

JavaScript is a great language. It can still behave oddly at times.

	Always Introduce Variables with `var` (but Not in the JavaScript Console).
	Code is better if you always use `var` when first introducing a variable in a project or inside a function. It's easier to read and less likely to lead to weird bugs. But don't use `var` in the JavaScript console, because JavaScript is weird.

Different Kinds of Things in JavaScript

JavaScript can set variables to lots of different things. It can set numbers, words, dates, and true-and-false values. JavaScript has two different ways to set "nothing" (JavaScript is weird). There are a couple of ways to list things. There are even ways to set a variable to 3D shapes!

We'll take a close look each of these—and ways to code with them.

Code Is for Computers and Humans, Comments Are Only for Humans

We write code for both computers *and* people. Code is written for computers so computers can do stuff. Code is written for people so we can understand code and add features to it. Sometimes code can be a little hard for people to understand. When that happens, programmers add *comments* to their code.

Computers know that they're supposed to ignore comments. Programmers know that we should read them to better understand the following code.

In JavaScript, double slashes indicate comments. Type the following in the JavaScript console.

```
// Set today's date
date = new Date()

// Set January 1, 2050
date = new Date(2050, 0, 1)
```

Unless you've seen JavaScript dates before, you probably wouldn't know what that code does (especially that 0 is the first month in JavaScript). But, thanks to comments, it's pretty easy to figure it out.

	You Don't Need to Type the Comments (but You Should).
	The comments you see in this book are meant to give you helpful hints. You don't *have* to type them in. But you should. When you open your code later to make a change, comments will help you remember why you did things. Really, instead of copying comments, you should add your own. If you have a hard time understanding something when you code it, chances are you will

have a hard time understanding it when you come back to add new stuff. Comments are little hints for your future self!

Next, let’s take a close look at the different kinds of JavaScript things, starting with numbers.

Numbers, Words, and Other Things in JavaScript

We know that we can change things, but how can each kind of variable change in JavaScript? Let's take them one at a time.

Numbers

You can use standard math symbols to add and subtract numbers in JavaScript. Try the following in the JavaScript console, pressing `Enter` after each math problem.

```
5 + 2
10 - 9.5
23 - 46
84 + -42
```

You should get back the following answers (the answer is shown in the comments below the math problem).

```
5 + 2
// 7

10 - 9.5
// 0.5

23 - 46
// -23

84 + -42
// 42
```

So it even works with negative numbers. Remember this, because negative numbers will be handy as we play with 3D graphics.

OK, so adding and subtracting are pretty easy in JavaScript. What about multiplication and division? Plus and minus sign keys are on most keyboards, but not `×` and `÷` keys.

For multiplication, use the asterisk (`*`) character:

```
3 * 7
// 21

2 * 2.5
// 5

-2 * 4
// -8

7 * 6
// 42
```

Division is done with the slash (**/**) character:

```
45 / 9
// 5

100 / 8
// 12.5

84 / 2
// 42
```

One other thing to know about numbers is that when doing a lot of arithmetic at the same time, you can use parentheses to group things. The math inside parentheses is always calculated first:

```
// Same as 5 * 6
5 * (2 + 4)
// 30

// Same as 10 + 4
(5 * 2) + 4
// 14
```

What happens without the parentheses? Can you guess why?

Without parentheses, multiplication is done first, then addition. Remember the “order of operations” from your math class!

One last number operator worth a mention is the increment operator **++**. It increases a variable by one, but it returns the previous value.

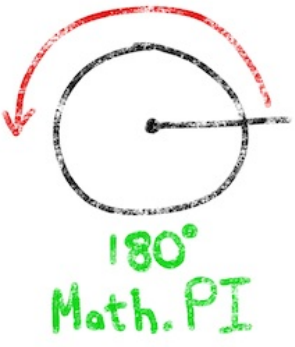
```
i=0
// 0

j=i++
// 0

i
// 1

j
// 0
```

When we call `i++` on the second line, it increases the value of `i` from 0 to `1`. But it returns the value before the increase, which is set to `j`. So `j` is set to `0` at the same time that `i` is increased by one. Be sure to try this out in the JavaScript console. The behavior might seem a little strange, but it's quite helpful as we'll see later in this chapter and throughout the book.

Geometry

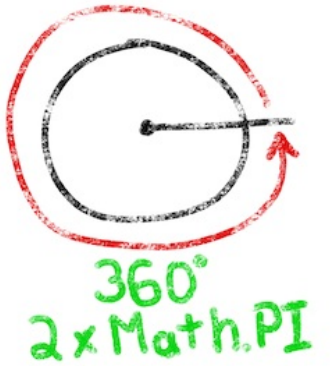
We're working on 3D game concepts in this book, which means geometry. We'll discuss geometry in more detail as part of the various project chapters that need it. For now, let's consider two geometric functions: sine and cosine. If you don't know them, don't worry—you'll get to know them in the games.

Just remember that in JavaScript, we don't use degrees. Instead we use *radians*. What are radians? Instead of saying that we turned 180° when we spun halfway around a circle, we would say that we "turned *pi* radians around."



Pi is a special number in math. Its value is about 3.14159. You will often see the symbol π used for pi. We'll call it pi since JavaScript calls it `Math.PI`.

If you remember the donuts from *Rendering Donuts (Not the Kind You Eat) with Torus*, this is how we made the half-eaten donut. We told it to go 3.14 radians around—or about pi radians around. Going around a full circle is twice as much as a 180° turn, more commonly called a 360° turn, which is two pi radians, or 2 × pi.



360° is `2*Math.PI` in JavaScript. A handy conversion table follows:

Degrees	Radians	JavaScript
0°	0	`0`
45°	pi ÷ 4	`Math.PI/4`
90°	pi ÷ 2	`Math.PI/2`
180°	pi	`Math.PI`
360°	2 × pi	`2*Math.PI`
720°	4 × pi	`4*Math.PI`

Geometric functions are also available in JavaScript's `Math` code collection. An example is the sine that we saw in Chapter 6, *Project: Moving Hands and Feet*. JavaScript shortens both sine and its companion, cosine, to `sin` and `cos`:

```
Math.sin(0)
// 0

Math.sin(2*Math.PI)
// 0

Math.cos(0)
// 1
```

Really, Really Close to Zero

Depending on your computer, when you tried `Math.sin(2*Math.PI)` in the JavaScript console, you may not have gotten the right answer. The sine of 2 × pi is 0, but you may have seen something like `-2.4492127076447545e-16` instead. This shows that computers are not perfect. Sometimes their math can be off by a tiny amount.

When JavaScript has `e-16` at the end of a number, it means that it's a decimal number with 16 places to the left. In other words, `-2.45e-16` is the same thing as writing `-0.000000000000000245`. That's a really, really small number—you'd have to add it together more than two million times to get to `1`.

These `Math.` functions and the simple arithmetic operators will pop up quite often as we progress. Math is lots of fun in 3D game programming!

Strings

Words and letters are *strings* in JavaScript. Strings start and end with a quotation

mark.

```
title = "3D Game Programming for Kids"
```

Strings can also start and end with an apostrophe.

```
title = '3D Game Programming for Kids'
```

Many programmers prefer apostrophes because we don't have to hold down the Shift key. Yes, programmers really are that lazy.

That said, some people prefer the look of quotes and they are useful when the string needs to hold an apostrophe.

```
motto = "Don't be lazy"
```

No matter what we use, JavaScript always uses quotation marks. So don't be surprised if the JavaScript console reports your apostrophe string with quotes.

Strings are combined together using the plus operator.

```
str1 = 'Howdy'
str2 = 'Bob'

str1 + ' ' + str2;
// "Howdy Bob"
```

Pretty weird, right? We already mentioned that most keyboards don't have multiplication or division keys. So it's not too surprising that there are no stick-two-strings-together keys. It is a little surprising that JavaScript reuses the plus sign.

What do you suppose happens if you try to join a string and a number? Well, give it a try:

```
str = 'The answer to 7 + 4 is: '
answer = 7 + 4

str + answer
```

Try This Yourself

You're following along with all of this in the JavaScript console, right? Definitely try this one out!

The result is that, when combining a string and a number, JavaScript treats the number as a string:

```
str = 'The answer to 7 + 4 is: '
answer = 7 + 4

str + answer
// "The answer to 7 + 4 is: 11"
```

But beware! It's easy to confuse JavaScript with the plus operator. Try each of the following in the JavaScript console.

```
'The answer to 7 + 4 is ' + 7 + 4
'The answer to 7 + 4 is ' + (7 + 4)
```

Did you try it? The moral of this lesson is that you should use parentheses if you try to do math when combining numbers with strings. But the real lesson is to do math before you combine the result with a string—like we did when we set the `answer` variable.

Booleans

A *Boolean* value is either true or false.

```
no = false
// false

yes = true
// true
```

It’s possible to flip Booleans with the `not` operator. In JavaScript, the exclamation point is the `not` operator.

```
theOpposite = !yes
// false

theOppositeOfTheOpposite = !!yes
// true
```

We won’t use Booleans directly like this very often. We’ll usually see comparison operators that make Booleans.

```
isTenGreaterThanSix = 10 > 6
// true

isTwelveTheSameAsEleven = 12 == 11
// false
```

The list of comparison operators is as follows:

Operator	Name	Description
==	Equal	Are the two values equal?
<	Less than	Is the first value less than the second?
>	Greater than	Is the first value greater than the second?
<=	Less than or equal	Is the first value less than or equal to the second?
>=	Greater than or equal	Is the first value greater than or equal to the second?

	Double Equal Sign vs. Single Equal Sign
	A double equal sign (`==`) is a JavaScript expression that checks whether something is equal to something else. It makes no changes to anything—it only checks values and produces a Boolean value.

As we have seen throughout the book, a single equal sign (`=`) *makes* a variable equal to something else. It is called the *assignment operator* because it assigns a value to a variable—it changes that variable.

You might be wondering whether it's wise to have two very different operators that look so similar. It isn't. This is a very common source of mistakes—even for people who've been programming for years. But since it's been around for so long, it probably won't be changing any time soon. So be on the lookout for these kinds of mistakes.

Two other operators work with Boolean values: the *and* operator (`&&`) and the *or* operator (`||`). These let us ask whether two things are true, whether one of two things is true, and other similar questions.

Below we use these operators to find out whether April 1, 2025 is a weekday or weekend. As mentioned earlier, wacky JavaScript thinks January is the `0` month, making April the `3` month.

```
date = new Date(2025, 3, 1)
// Tue Apr 01 2025 14:00:00 GMT-0400 (EDT)

isWeekDay = date.getDay() != 0 && date.getDay() != 6
// true

isWeekend = date.getDay() == 0 || date.getDay() == 6
// false
```

The value of `getDay()` is the day of the week, starting with `0` for Sunday and ending with `6` for Saturday. So, a date is a weekday if `getDay()` is not `0` *and* it's not `6`. Similarly, a date is a weekend if `getDay()` is `0` *or* `6`. But don't use `||` or `&&` too much—it can make code hard to read.

Nothing

JavaScript even has a way to describe nothing. In fact, JavaScript loves nothing so much that it has two things that mean nothing!

```
meansNothing = null
alsoMeansNothing = undefined
```

When programmers say a value is empty, we use `null`. The `undefined` value is not normally set by programmers. Instead, JavaScript uses that to identify variables that have never been set.

Listing Things

At times it's quite handy to be able to describe a list of things. In JavaScript, lists are made with square brackets. A list of amazing movies might look something like this:

```
amazingMovies = [
  'Star Wars',
  'The Empire Strikes Back',
  'Indiana Jones and the Raiders of the Lost Ark'
]
```

The number of things in a list is the "length" of the list.

```
amazingMovies.length
// 3
```

To get one entry in the list, use square brackets with a number, starting at 0.

```
amazingMovies[0]
// "Star Wars"

amazingMovies[1]
// "The Empire Strikes Back'

amazingMovies[2]
// "Indiana Jones and the Raiders of the Lost Ark"

amazingMovies[3]
// ???
```

Be sure to try this in the JavaScript console. The `0` entry is "Star Wars," the first

entry in the list. The `2` entry is “Indiana Jones and the Raiders of the Lost Ark,” the last entry in the list. The length of the list is `3`—the `0`, `1`, and `2` entries. So what happens when you access `amazingMovies[3]`? Try it out!

To change a value in a list, we can use the assignment operator.

```
amazingMovies[2] = 'Indiana Jones and the Last Crusade'
// "Indiana Jones and the Last Crusade"
```

To add a value to a list, use `push()`, which will return the new length of the list.

```
amazingMovies.push('Wonder Woman')
// 4

amazingMovies[3]
// "Wonder Woman"
```

Lists are a great way to store a bunch of information. Let’s look at another way, called “maps”.

Maps

Maps and lists both collect information in JavaScript. The main difference between lists and maps is the way that you get values from them.

In lists, we get values based on where they are in the list—the value at `0`, the value at `1`, the value at `42`. Lists are useful, but sometimes not as easy to use as we might like—especially if we want to store different kinds of information.

To create maps in JavaScript, you use curly braces.

```
greatMovie = {
  name: 'Toy Story',
  year: 1995,
  stars: ['Tom Hanks', 'Tim Allen']
}
```

Maps get their name because they *map* a key to a value. In `greatMovie`, the key `name` *maps* to the value `"Toy Story"`. The key `year` maps to 1995. The key `stars`

maps to a list of two actors.

To get information out of a map, we use square brackets, just like we did with lists. But instead of using a number inside the square brackets, we use a string version of the key.

```
greatMovie['name']
// "Toy Story"

greatMovie['year']
// 1995
```

Another way to get information from a map is by placing the key after a period.

```
greatMovie.name
// "Toy Story"

greatMovie.stars
// ["Tom Hanks", "Tim Allen"]

greatMovie.stars[0]
// "Tom Hanks"
```

To change a value or to add a value, we can use assignment.

```
greatMovie['name'] = 'Toy Story 2'
// "Toy Story 2"

greatMovie['description'] = 'Woody is stolen by Al'
// "Woody is stolen by Al"
```

We will see maps again when we talk about flat shading in Chapter 9, *What's All That Other Code?*. Maps will also play a big part later.

Control Structures

We've talked about the individual parts that make up JavaScript: the numbers, strings, and Booleans that serve as nouns in JavaScript sentences. In this section, we talk about JavaScript *sentences* and *paragraphs*—about combining the parts into larger code structures.

Running Code Only If Something Is True

Sometimes we want to skip over code. For example, we probably don't want to animate the scene if the game is over. In these cases, we use the `if` keyword.

```
gameOver = true
if (gameOver) console.log('Game Over!!!')
// "Game Over!!!"
```

Simple `if` statements like this can go on a single line. When more than one thing is being done after an `if`, we need to wrap these things inside curly braces.

```
gameOver = true
if (gameOver) {
  now = new Date()
  console.log('Game ended on ' + now.toDateString())
}
// "Game ended on Sun Apr 01 2018"
```

In this case, we set the `now` variable to today's date on the first line of the `if` statement. On the second line, we use that `now` variable to report when the game ended.

The expression that follows `if` doesn't have to be a variable. It just needs to result in a Boolean value.

```
gameOver = true
score = 400

if (gameOver && score > 100) console.log('Great Game!!!')
// "Great Game!!!"
```

We can also extend an `if` statement with `else if` and `else`.

```
score = 10

if (score > 100) console.log('Great Game!!!')
else if (score > 20) console.log('Game Over.')
else console.log('Game Over :(')
// "Game Over :("
```

Use the up-arrow console editing trick to try different values for `score`. If `score` is more than `100`, the first message should be logged to the console. Otherwise, if `score` is more than `20`, then the simple "Game Over" message should be logged. And if the score is not greater than `100` or `20`, then the sad game over message should be logged.

Don't Overuse `if`s

`if`, `else if`, and `else` are very powerful, but can be used too much. If you have to use them, try to stick with the one-line version—they are easier to read and less likely to add bugs in code.

Loops

We first saw loops in Chapter 5, *Functions: Use and Use Again*, where we had our code loop 100 times to create 100 planets. So we know that loops are a great way to do something again and again. Let's look closer now.

A *for* loop has the following structure:

```
for (var i=0; i<5; i++) {
  console.log('Loop #' + i)
```

```
}
// "Loop #0"
// "Loop #1"
// "Loop #2"
// "Loop #3"
// "Loop #4"
```

Three expressions are inside the parentheses of a for loop. The first declares a looping variable. We use the `var` keyword to declare that `i` is the looping variable—and that it starts with a value of `0`. The next expression is a Boolean that describes when the loop continues to run. In the above example, we say that, as long as `i` is less than `5`, the loop continues. The last expression is run after each loop, which we use to increase the value of `i` by one with the `++` operator.

Each time the loop runs, the code inside the curly braces is run. In this case, we have a single line that logs the loop number to the console.

The loop variable can start with any value. The condition that keeps the loop running can be any Boolean expression. The last expression can do anything, including decreasing the loop variable by 1.

```
for (var i=5; i>=0; i--) {
  console.log('Loop #' + i)
}
// "Loop #5"
// "Loop #4"
// "Loop #3"
// "Loop #2"
// "Loop #1"
// "Loop #0"
```

Loops are especially great for working with lists.

```
amazingMovies = [
  'Star Wars',
  'The Empire Strikes Back',
  'Indiana Jones and the Raiders of the Lost Ark'
]

for (var i=0; i<amazingMovies.length; i++) {
  console.log('GREAT: ' + amazingMovies[i])
```

```
}
// "GREAT: Star Wars"
// "GREAT: The Empire Strikes Back"
// "GREAT: Indiana Jones and the Raiders of the Lost Ark"
```

We're using the same `i` looping variable and the same `++` expression to increase that looping variable. The condition that keeps the loop going is that `i` is less than the length of `amazingMovies`—as long as `i` is less than 3, the loop continues.

Recalling how we look things up in lists, the first time through the loop, `i` is `0`, so we log `amazingMovies[0]`—Star Wars—to the console. The last time through the loop when `i` is less than 3, we log `amazingMovies[2]`—The Raiders of the Lost Ark—to the console. Just be sure not to use `<=` for deciding whether a list loop can continue. Can you guess why? Try it out!

Another form of for loops works only with maps. It looks a little like the regular format, but uses the `in` keyword to get all of the keys in a map.

```
greatMovie = {
  name: 'Toy Story',
  year: 1995,
  stars: ['Tom Hanks', 'Tim Allen']
}

for (var key in greatMovie) {
 console.log(key + ': ' + greatMovie[key])
}
// "name: Toy Story"
// "year: 1995"
// "stars: Tom Hanks,Tim Allen"
```

Inside the loop, we log each key along with the value the key maps to. The two different for loops are great for getting information out of lists and maps.

What's Next

This chapter introduced a lot of information. Don't worry if not all of it makes sense yet. When you work through the later chapters, come back here if you have questions. More and more of this will begin to make sense as you progress.

The material in this chapter and in Chapter 5, *Functions: Use and Use Again*, covers a large portion of the JavaScript language. You really know a lot of JavaScript now! But don't mistake knowing a lot for being able to use it well. We know how to make "words" and "sentences" in JavaScript, but we are still a long way from writing great stories, which is why so many chapters are still left to go in this book!

With that in mind, let's get back to adding cool stuff to our avatar!

When you're done with this chapter, you will

- *Know even more fun math for 3D programming*
- *Know how to rotate something to face a specific direction*
- *Be able to make smooth animations*

Chapter 8

Project: Turning Our Avatar

We're nearly finished animating our avatar. In Chapter 4, *Project: Moving Avatars*, we learned how to make our avatar move. In Chapter 6, *Project: Moving Hands and Feet*, we made the avatar look like it was walking. Now we need to make it look as though it can turn when we switch directions. Turning, or rotating, is not new to us—we already can make the avatar turn when flipping and cartwheeling. But this time, we want to make our avatar face a particular direction.

Getting Started

If it's not already open in the 3DE Code Editor, open the project that we named **My Avatar: Moving Hands and Feet** (from Chapter 6, *Project: Moving Hands and Feet*). To make a copy of it, click the menu button and choose **MAKE A COPY** from the menu.

Name the project **My Avatar: Turning** and click the **SAVE** button.

Facing the Proper Direction

Getting the avatar to face the proper direction is fairly easy—especially with all that we already know. Just as we did when we added the walking motion of the hands and feet, we'll add another function to animate turning our avatar.

Let's start by going to the list of things that control our avatar.

```
var clock = new THREE.Clock();
var isCartwheeling = false;
var isFlipping = false;
var isMovingRight = false;
var isMovingLeft = false;
var isMovingForward = false;
var isMovingBack = false;
```

At the bottom of that list, add `direction` and `lastDirection`.

```
var direction;
var lastDirection;
```

We need `direction` so our code knows which way to turn the avatar. We need `lastDirection` so the code can do nothing when the avatar is already facing—or turning—the right way.

Now we'll write a new function to turn our avatar. We'll call this function `turn`. Add it below the `animate()` function.

```
function turn() {
  if (isMovingRight) direction = Math.PI/2;
  if (isMovingLeft) direction = -Math.PI/2;
  if (isMovingForward) direction = Math.PI;
  if (isMovingBack) direction = 0;
  if (!isWalking()) direction = 0;

  avatar.rotation.y = direction;
}
```

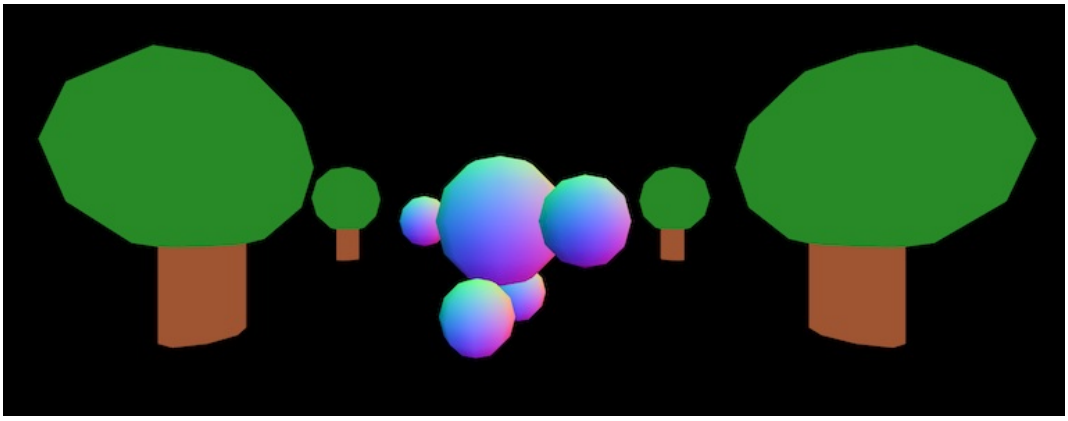
Finally, we call this function inside of `animate()`. Add the call to `turn()` just after

the first line—after the `requestAnimationFrame(animate)` line.

```
  function animate() {
    requestAnimationFrame(animate);
»   turn();
    walk();
    acrobatics();
    renderer.render(scene, camera);
  }
  animate();
```

Hide the code and try it out!

If everything is working, when we walk left or right, the avatar faces the direction in which it's moving:



That is pretty amazing. At this point, you've made a pretty complicated game avatar. Take a moment to think about all that you've accomplished:

- Given the avatar a body, hands, and feet
- Made the avatar move so that all the pieces move with it
- Made the avatar do cartwheels and flips
- Stuck the camera to the avatar
- Stuck the camera to the avatar's position so that flips and cartwheels don't make us dizzy

- Made the hands and feet swing back and forth when the avatar walks
- Made the hands stop moving when the avatar is not moving
- Made the avatar face the direction that it’s walking

That is an incredible amount of JavaScript 3D programming. You have done very well to make it this far. But we can still do so much more!

First let’s take a closer look at that `turn` function so we’re sure we understand what’s going on there.

Breaking It Down

In the `turn` function, why do we set the direction to values like `Math.PI` and `-Math.PI/2`?

Recall from *Geometry*, that angles, or the amount of rotation, use radians instead of degrees. At the start, the avatar is facing toward the camera. So 0° of rotation is 0 radians of rotation, which means facing backward. And 180° is `pi` radians, which means facing forward into the screen. The following table is the complete list we're using in the `turn` function:

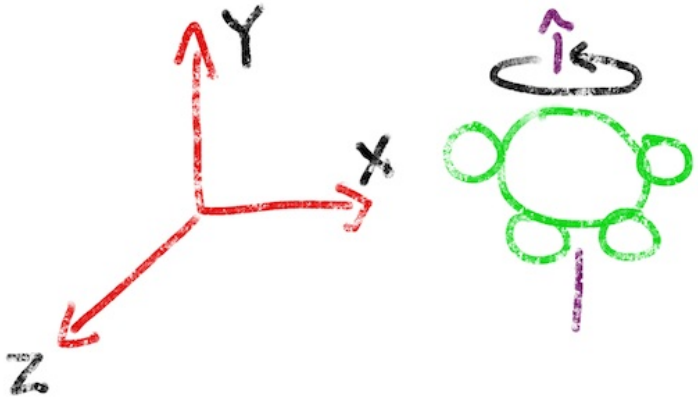
Direction	Degrees	Radians	JavaScript
Forward	180°	pi	`Math.PI`
Right	90°	pi ÷ 2	`Math.PI/2`
Left	-90°	-pi ÷ 2	`-Math.PI/2`
Backward	0°	0	`0`

Why `rotation.y`?

So that explains the number that we use for the `direction` variable in the function `turn`, but why do we set `rotation.y`? Why not `rotation.z` or `rotation.x`?

Well, for one thing, we already change `rotation.x` when we do cartwheels and `rotation.z` when we flip. So it makes sense that we'd use something else to spin.

We set the `rotation.y` because we want to spin the avatar around the y-axis. Recall that, in 3D, the y-axis is pointing up and down. If you imagine a pole sticking right up the middle of the avatar, that is the avatar's y-axis:



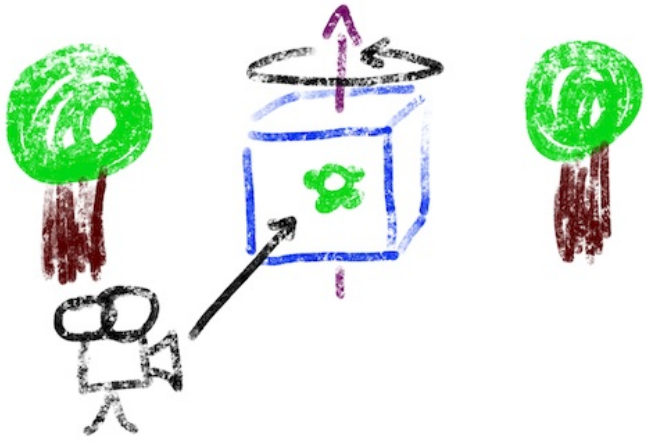
Spinning the avatar around this pole is what it means to rotate the avatar around the y-axis.

Don’t Forget About avatar.rotation!

If you tried turning `marker.rotation` instead of `avatar.rotation`, you may have noticed that not only did the avatar spin, but everything else seemed to spin as well. This is because we attached the camera to the avatar’s `marker`:

```
var marker = new THREE.Object3D();
scene.add(marker);

marker.add(camera);
```

Think of the marker as an invisible box that holds the avatar’s parts. By adding the camera to the marker, we’re sticking it to one side of the marker. If we spin the box, then the camera has to go along with it:



This is also why we added the hands and feet to the avatar's head instead of to the avatar's marker. When we turn the avatar inside the marker, its hands and feet need to move with it—not stay still with the marker.

Which Direction When Not Walking

One last thing to note inside `turn()` is that when the avatar is not walking, we face forward.

```
if (!isWalking()) direction = 0;
```

This line tells `turn()`, if the avatar is not moving, then spin it back around to face forward. As a game programmer, you may prefer to keep the avatar facing in the direction it was facing when it stopped walking. That is, if the avatar is walking right and the player lets go of the right arrow key, then you might want the avatar to still face right. If that's your preference, then remove that line. You're the game programmer—it's up to you!

Animating the Spin

When we turn our avatar, it immediately faces a new direction. Let's make it a little more realistic by animating a turn to the new direction. For that, we'll need a new JavaScript *code collection*. We'll talk more about code collections in the next chapter, Chapter 9, *What's All That Other Code?*. For now, just think of them as a way to use code from somewhere else.

The code collection that we're going to use will help us animate between different positions and rotations. The code collection is called *Tween*. It gets its name from its ability to change *between* values over time.

For this, go to the top of your code (the very top, not just to the `START CODING ON THE NEXT LINE` line). Add the `<script>` tag for `tween.js`, as shown:

```
    <script src="/three.js"></script>
»   <script src="/tween.js"></script>
```

The first step in using Tween is to add its `update` function to our `animate` function, just above the call to `turn()`.

```
    function animate() {
      requestAnimationFrame(animate);
»     TWEEN.update();
      turn();
      walk();
      acrobatics();
      renderer.render(scene, camera);
    }
    animate();
```

Next we need to change the function `turn` that we just wrote. Instead of setting the direction right away, we'll start a tween that will spin the avatar in the new direction. Replace the the last line of the `turn()` function—the one that sets `avatar.rotation.y`—as shown.

```
    function turn() {
      if (isMovingRight) direction = Math.PI/2;
```

```
    if (isMovingLeft) direction = -Math.PI/2;
    if (isMovingForward) direction = Math.PI;
    if (isMovingBack) direction = 0;
    if (!isWalking()) direction = 0;

    if (direction == lastDirection) return;
    lastDirection = direction;

»   var tween = new TWEEN.Tween(avatar.rotation);
»   tween.to({y: direction}, 500);
»   tween.start();
  }
```

Be sure to include the lines that check and set `lastDirection`. These two lines first check to see whether the direction the avatar is now facing is the same as the direction it was facing the last time `turn()` was called. We don't want to animate a turn to face left if we're already turning to face left!

The three tween lines at the bottom do the actual spinning. The first line says that our tween should change the avatar's rotation. The second says that we want to tween the Y rotation to stop at `direction`—and that we want it to take 500 milliseconds (½ a second). Last, we start the tween!

	Let's Play!
	We told the Tween code collection that it will run from start to finish in 500 milliseconds. This number was at the end of the line that started with `to`. It takes 1000 milliseconds to make one second, so 500 milliseconds is half a second. The spin of the avatar takes less than a second. Experiment with that number to get it the way you like. Is 1000 too long? Is 10 too short? You decide!

The Code So Far

If you would like to double-check the code in this chapter, turn to *Code: Turning Our Avatar*.

What’s Next

Wow! Our simple avatar simulation is getting quite sophisticated, isn’t it? We’ve already put quite a bit of work into our avatar, but you may have noticed that it can pass right through our trees. In the next project chapter, we’ll talk about collision detection and use it to make our avatar stop when it collides with a tree.

But first it’s time to take a closer look at all of that JavaScript code that was added for us when we started this project.

When you're done with this chapter, you will

- *Know a little about making web pages*
- *Understand the starter code*
- *Be comfortable changing the starter code*

Chapter 9

What's All That Other Code?

We skipped over a lot so we could start programming right away. It was totally worth it, too—we've created and animated shapes, we've built and controlled an avatar, and we've learned some impressive debugging skills.

Even though we've been able to do all that, the stuff above the "START CODING ON THE NEXT LINE" is important to understand. And thanks to all of the skills we've already picked up, it will be that much easier to understand what all that other code does.

Getting Started

Create a new project from the `3D starter` template in the 3DE Code Editor. Name the project `All that other code`.

A Quick Introduction to HTML

At the very top of our code is the following HTML:

```
<body></body>
```

HTML is the Hypertext Markup Language. HTML is not a programming language. So what's it doing messing up our beautiful JavaScript code?

It's used to build web pages, but it doesn't make web pages do interesting things. Doing interesting things on web pages is why JavaScript got invented.

Even though it's not a programming language, we still need HTML. Since JavaScript is a web programming language, we need a web page where we can program—even if it is just a simple page.

The first line contains `<body>` *tags*. Instead of price tags or names tags, HTML tags describe things that are useful on the web. Most HTML tags, like `<body>`, come in pairs. The first part of the pair is the name of the tag surrounded by a less-than symbol (`<`) and a greater-than symbol (`>`)—like `<body>`. The second part is the same thing, but with a slash (`/`) before the tag name—like `</body>`.

HTML authors normally put writing, images, links to other pages, and more in between the `<body>` and `</body>` tags. Since we've been programming, we haven't put anything between the `<body>` tags. Our web pages start with no "body". Instead, we've been filling that empty body with our 3D scenes.

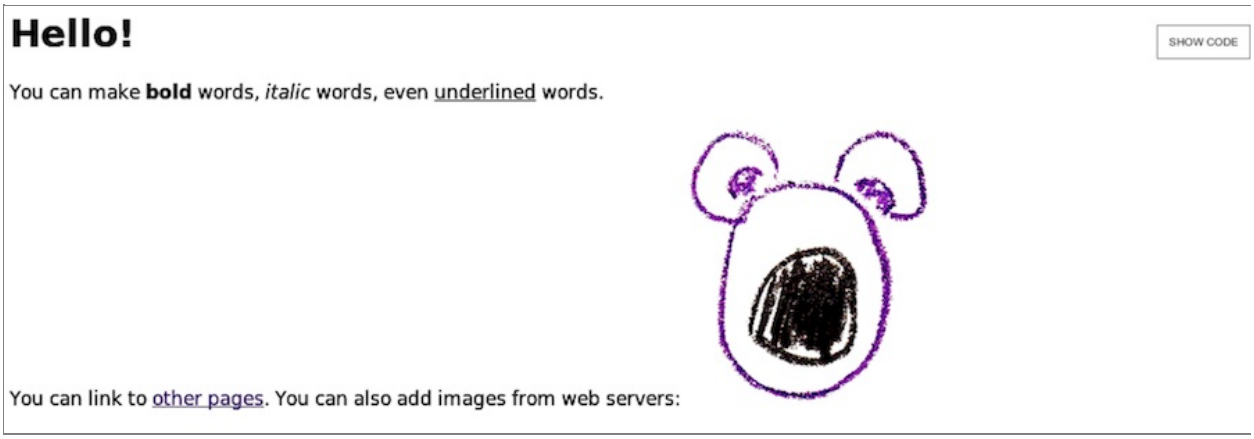
To get a quick sense of what HTML does, add the following HTML in between the two `<body>` tags, as shown:

```
<body>
  <h1>Hello!</h1>
  <p>
    You can make <b>bold</b> words,
    <i>italic</i> words,
    even <u>underlined</u> words.
  </p>
```

```
    <p>
      You can link to
      <a href="http://code3Dgames.com">other pages</a>.
      You can also add images from web servers:
      <img src="/images/purple_fruit_monster.png">
    </p>
  </body>
```

Ignore 3DE Warnings for HTML
 Your HTML code may get red X warnings. These can be safely ignored. 3DE is meant to edit JavaScript, not HTML, so it can get confused.

If you hide the code in the 3DE Code Editor, you'll see something like this:



This is a JavaScript book, not an HTML book, but you already see some of what's possible with HTML.

Following the `<body>` tags is another line of HTML with opening and closing `<script>` tags.

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

Just like the `<body>` tags, these `<script>` tags are HTML. These tags load JavaScript from elsewhere on the web so that we can use it. In this case, we're loading the very excellent 3D JavaScript code named Three.js.

JavaScript doesn't have to come from other locations. For most of this book, we're coding *inside* an HTML web page. But when JavaScript code gets very large—like a bunch of 3D commands and functions—it's very helpful to load it using `<script>` tags. So far in this book, we've written approximately 120 lines of code and it's probably already getting hard to find some of the code that we've written. Just imagine if we included 1000 lines of someone else's code in the same place as ours!

Setting the Scene

To do anything in 3D programming, we need a scene. Think of the scene as the universe in which all of the action is going to take place.

Scenes are really simple to work with. We've been adding objects to them throughout the book. After things have been added to a scene, it's the scene's job to keep track of everything. In fact, that's pretty much all we need to know about scenes—after creating one, we add lots of stuff to it and the scene takes care of the rest.

The following code in 3DE does just that:

```
// The "scene" is where stuff in our game will happen:
var scene = new THREE.Scene();
var flat = {flatShading: true};
var light = new THREE.AmbientLight('white', 0.8);
scene.add(light);
```

The `flat` value is a map, which we described in *Maps*. Back when we first introduced meshes in chapter 1, we used the `flat` value to make the chunks in our cover material flat and chunky. This is where that `flat` value comes from.

Finally, we add a light to the scene. That light will be handy when we start exploring Chapter 12, *Working with Lights and Materials*.

Using Cameras to Capture the Scene

Scenes do a great job of keeping track of everything, but they don't show us what's happening. To see anything in the scene, we need a camera. Notice the following code in 3DE:

```
// The "camera" is what sees the stuff:
var aspectRatio = window.innerWidth / window.innerHeight;
var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
camera.position.z = 500;
scene.add(camera);
```

This takes what's in the scene and shows it on our computer screens.

The purpose of the `aspectRatio` is to determine the shape of the browser window. This is the same thing as aspect ratios for movie screens and TV sets. A large TV with a 4:3 aspect ratio might be four meters wide and three meters tall (OK that's a *really* large TV). An even larger 4:3 screen might be twelve meters wide and nine meters tall (multiply both the 4 and 3 in 4:3 by 3 to get 12:9). Most movies today are made at an aspect ratio of 16:9, which would mean a nine-meter-tall screen would be sixteen meters wide—four extra meters when compared with the same-height 4:3 aspect ratio.

Why does this matter for us? If you try to project a movie made in 16:9 onto a 4:3 screen, a lot of squishing has to be done. Similarly, a 4:3 movie would need to be stretched to be shown on a 16:9 screen. Instead of stretching or squishing, most movies are chopped so that you miss those four meters of action. Our Three.js library doesn't chop—it stretches or squishes. In other words, it's pretty important to get the aspect ratio right.

After we build a new camera, we need to add it to the scene. Like anything in 3D programming, the camera is placed at the center of the scene we add it to. We move it 500 units away from the center in the Z direction ("out" of the screen) so that we have a good view of what's going on back at the center of the scene.

Using a Renderer to Project What the Camera Sees

The scene and the camera are enough to describe how the scene looks and where we're viewing it from, but one more thing is required to show it on the web page. This is the job of the *renderer*. It shows, or *renders*, the scene as the camera sees it:

```
// The "renderer" draws what the camera sees onto the screen:
var renderer = new THREE.WebGLRenderer({antialias: true});
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

When we create the `renderer` value, we use a map that sets "antialias" to `true`. *Antialiasing* is a computer term that means smooth edges. 3D graphics look nicer when their sides are smooth, which is why we enable antialiasing here.

We have to tell the renderer the size of the screen it will be drawing to. We set the size of the view to take up the whole browser window (`window.innerWidth` and `window.innerHeight`).

To include the renderer in the web page, we use its `domElement` property. A `domElement` is another name for an HTML tag like those we added earlier in the chapter. Instead of holding a title or paragraph, this `domElement` holds our amazing 3D worlds.

We add that `domElement` to the `document.body`—which is the same `<body>` tag from earlier that held the HTML. The `appendChild` function takes care of adding the `domElement` to the document body. If you're wondering why we have names like `appendChild` and `domElement`, all I can tell you is be glad you're a 3D-game programmer, not a web programmer. Web programmers have to use silly (and hard-to-remember) names like this all the time.

At this point, the renderer *can* draw to the screen, but we still need to tell it to render before anything will show up. This is where `renderer.render()` comes into play at the end of your current code.

```
// Now, show what the camera sees on the screen:
renderer.render(scene, camera);
```

It might seem as though the renderer is an obnoxious younger brother or sister, doing the right thing only when we're extremely specific in our instructions. In a sense this is true, but in another sense all programming is like this. Until we tell the computer in exactly the right way to do something, it often does something completely unexpected.

In the case of the renderer, we can already see why it's nice to have this kind of control. In some of our experiments, we rendered only a single time. But in many of our projects, we render repeatedly inside an `animate` function. Without this kind of control, it would be much harder to choose the right rendering style.

Exploring Different Cameras

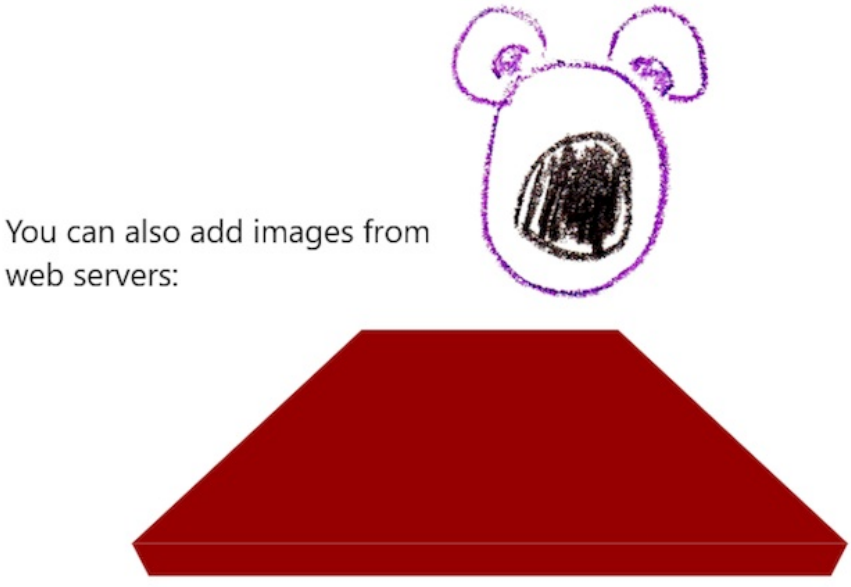
You may have noticed that we call our camera a **PerspectiveCamera**. If that name seems oddly specific, that's because other kinds of cameras exist. For the most part, we want to stick with *perspective* cameras in 3D coding. Let's have a look at what a perspective camera is and compare it to another kind of camera that's sometimes used as well.

A Quick Peek at a Weirdly Named Camera

The other kind of camera is called *orthographic*. To understand what an orthographic camera does, let's add a red road that the purple fruit monster can travel on. Add the following after **START CODING ON THE NEXT LINE**:

```
var shape = new THREE.CubeGeometry(200, 1000, 10);
var cover = new THREE.MeshBasicMaterial({color:'darkred'});
var road = new THREE.Mesh(shape, cover);
scene.add(road);
road.position.set(0, 400, 0);
road.rotation.set(-Math.PI/4, 0, 0);
```

Our perspective camera makes the road look something like this:

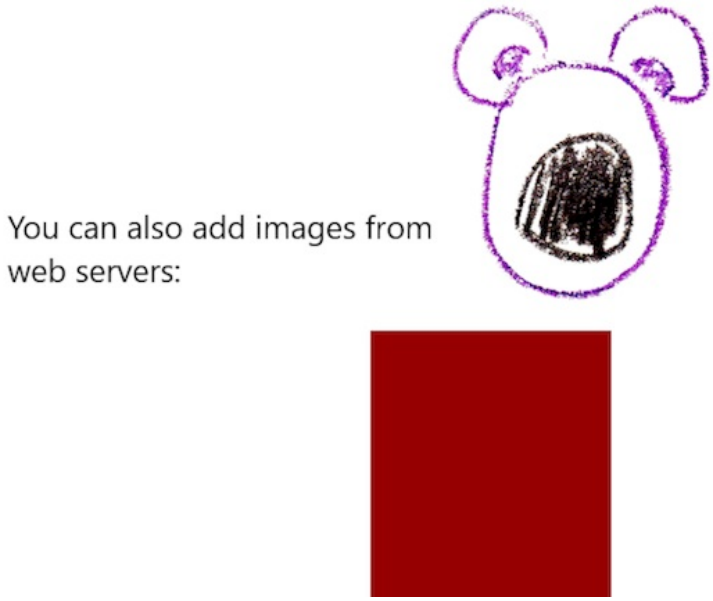


This is a rectangular road, but it doesn't *look* rectangular. It looks as though it's

getting smaller the farther away it gets. The perspective camera does this for us:

```
var aspectRatio = window.innerWidth / window.innerHeight;
var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
```

If we use an *orthographic* camera, on the other hand, everything looks flat:



That is the same road from the previous image. We've just replaced the two lines that create the perspective camera with the following:

```
var width = window.innerWidth;
var height = window.innerHeight;
var camera = new THREE.OrthographicCamera(
  -width/2, width/2, height/2, -height/2, 1, 10000
);
```

As you might imagine, the perspective camera that gives everything a three-dimensional feel is very handy in 3D games. Why would you want to use an orthographic camera?

Orthographic cameras are useful in two cases. The first is when you want to make a flat, 2D game. Using a 3D camera for a flat game just looks weird—especially at the edges of the screen. The other is when we make games with really long distances, such as space games. In fact, we can use orthographic cameras in some of the space simulations we'll do in a little while.

What’s Next

Now that we understand all about cameras, scenes, and loading JavaScript from elsewhere, we’ll change them more and more. But first, let’s teach our game avatar to *not* walk through trees.

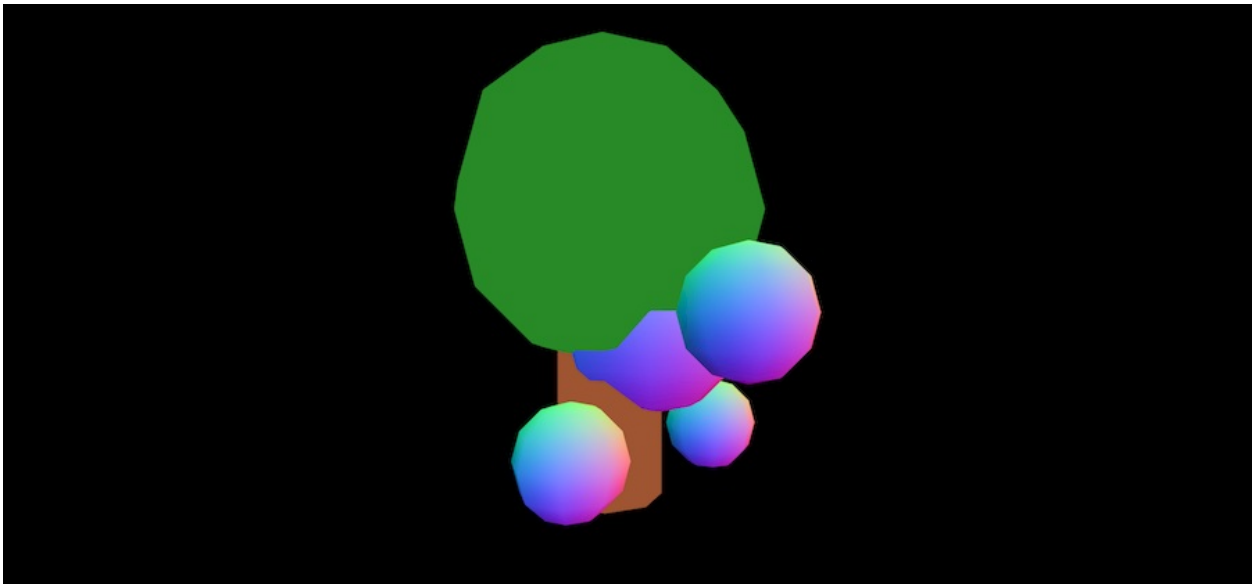
When you're done with this chapter, you will

- *Be able to stop game elements from moving through each other*
- *Understand collisions, which are important in gaming*
- *Have invisible fences around our trees*

Chapter 10

Project: Collisions

We have a slick game avatar. It moves, it walks, it even turns. But you may have noticed something unusual about our avatar. It can walk through trees.



Which is a little weird.

	Code Almost Never Does What We Want
	Code does only what we tell it to do. All too often, it does the opposite of what we *want* it to do. This can be one of the most frustrating things about programming. But finally getting code to do what

you want it to do is one of the most fun things about programming!

In this chapter we'll solve our avatar-in-a-tree problem using collision detection. We'll update our code to detect when our player is colliding with trees and stop the avatar from moving forward.

Collision detection is important in almost any game or simulation. There are different kinds of collision detection. In this chapter, we'll describe and talk about a *ray casting* approach to detecting collisions.

The great thing about ray casting is that it's so simple. Don't confuse simple with basic, though. Even though our collision detection will be simple, it is powerful. It's so powerful that programmers often use it in advanced, sophisticated games —especially when speed is required.

So let's dive in!

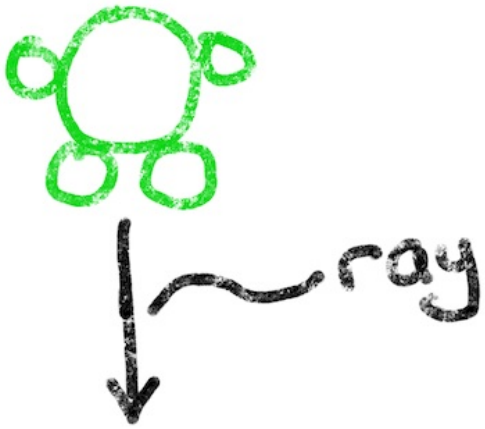
Getting Started

If it’s not already open in the 3DE Code Editor, open the project from Chapter 8, *Project: Turning Our Avatar*, which we named **My Avatar: Turning**.

Make a copy of our avatar project. From the menu in the 3DE Code Editor, select **MAKE A COPY** and enter **My Avatar: Collisions** as the new project name.

Rays and Intersections

To prevent our avatar from walking through trees, we'll start by imagining an arrow pointing down from our avatar.



In geometry, we call an arrow coming from something a *ray*. A ray is what you get when you start in one place and point in a direction. In this case, the place is where our avatar is and the direction is down. Sometimes giving names to such simple ideas seems silly, but it's important for programmers to know these names.

	Programmers Like to Give Fancy Names to Simple Ideas
	Knowing the names for simple concepts makes it easier to talk to other people doing the same work. Programmers call these names *patterns*.

Now that we have our ray pointing down, imagine circles on the ground around our trees.

Here’s the crazy-simple way that we prevent our avatar from running into a tree: we don’t! Instead, we prevent the avatar’s ray from pointing through the tree’s circle.

If at any time, we find that the next movement would place the avatar’s ray where it would point through the circle, we stop the avatar from moving. Think of the ray as a laser beam. It starts from the avatar and points down. If that laser *casts* its light onto the tree’s circle, we have detected a collision. That’s all there is to it!

It’s a little odd, but watching certain science fiction movies can make your life easier as a programmer. Sometimes programmers say weird things that turn out to be quotes from movies. It is not a requirement to watch or even like these movies, but it can help.

One such quote is from the classic *Star Trek II: The Wrath of Khan*. The quote is,

“He is intelligent, but not experienced. His pattern indicates two-dimensional thinking.” The bad guy in the movie was not accustomed to thinking in three dimensions, and the good guys used his two-dimensional thinking against him.

In this case, we want to do the opposite. We are building a three-dimensional game, but we’re only going to think about collisions in two dimensions. We’re not going to worry if the avatar’s head is high enough to touch the bottom of the leaves. We are not going to check whether a hand touches the trunk. That kind of collision detection is possible. But it takes a lot of work—both for programmers and computers. And that difficult collision detection usually adds nothing to our games.

So we cheat. We think about collisions in only two dimensions (X and Z), completely ignoring the up-and-down Y dimension.

At this point, a picture of what to do next should be forming in your mind. We need a list of these tree-circle boundaries that our avatar won’t be allowed to enter. We need to build those circle boundaries when we build the trees. Then we’ll add a ray for our avatar to detect when it’s about to enter a circle boundary. Last, we need to stop the avatar from entering these forbidden areas.

Let’s establish the list that will hold all forbidden boundaries. We’ll do this after the avatar and marker code. Just above the `makeTreeAt()` function, add the following:

```
var notAllowed = [];
```

Recall from *Listing Things*, that square brackets are JavaScript’s way of making lists. Here, our empty brackets create an empty list. The `notAllowed` variable is an empty list of spaces in which the avatar is not allowed.

We’ll make the next change inside the `makeTreeAt` function. When we make our tree, we’ll make the boundaries as well. Add the following code after the line that adds the treetop to the trunk. It should be just above the line that sets the trunk position with `trunk.position.set(x, -75, z)`.

```
var boundary = new THREE.Mesh(
  new THREE.CircleGeometry(300),
  new THREE.MeshNormalMaterial()
);
boundary.position.y = -100;
boundary.rotation.x = -Math.PI/2;
trunk.add(boundary);

notAllowed.push(boundary);
```

There's nothing super fancy there. We create our usual 3D mesh—this time with a simple circle geometry. We rotate it so that it lays flat and position it below the tree. And, of course, we finish by adding it to the tree.

But we're not done with our boundary mesh. At the end, we push it onto the list of disallowed spaces. Now every time we make a tree with the `makeTreeAt` function, we're building up this list. Let's do something with that list.

Let's add an `isColliding()` function. We'll add this after our other functions, `walk()`, `isWalking()`, and `acrobatics()`, and just above the `keydown` and `keyup` event listeners.

```
function isColliding() {
  var vector = new THREE.Vector3(0, -1, 0);
  var raycaster = new THREE.Raycaster(marker.position, vector);

  var intersects = raycaster.intersectObjects(notAllowed);
  if (intersects.length > 0) return true;

  return false;
}
```

This function returns a Boolean—a true-or-false answer—depending on whether the avatar is colliding with a boundary. To get that answer, we create a ray then check whether it points through anything. As described earlier, a ray is the combination of a direction—or vector (down in our case) and a point (in this case, the avatar's `marker.position`). We then ask that ray whether it goes through (intersects) any of the `notAllowed` objects. If the ray does intersect one of those objects, then the `intersects` variable will have a length that is greater than 0. In that case, we have detected a collision and we return `true`. Otherwise, there's no

collision and we return `false`.

Collisions are a tough problem to solve in many situations, so you're doing great by following along with this. But we're not quite finished. We can now detect when an avatar is colliding with a boundary, but we haven't actually stopped the avatar yet. Let's do this in the code that handles keyboard keys being pressed.

When an arrow key is pressed, our `sendKeyDown()` function changes the avatar's position.

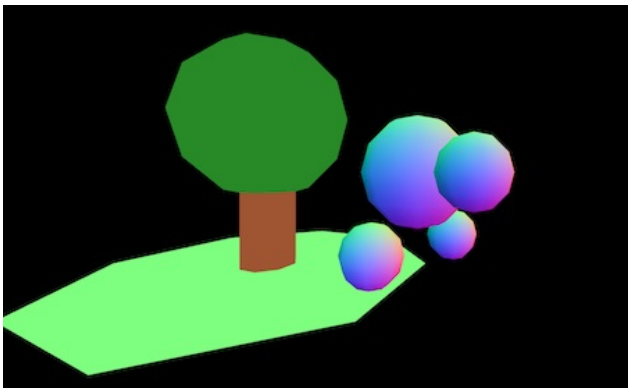
```
if (code == 'ArrowLeft') {
  marker.position.x = marker.position.x - 5;
  isMovingLeft = true;
}
```

A change like that might mean that the avatar has crossed the boundary. If so, we have to undo the move right away. Add the following code at the bottom of `sendKeyDown()`. Put it just after the `if (code == 'KeyF')` line and above the line with the closing curly brace.

```
if (isColliding()) {
  if (isMovingLeft)    marker.position.x = marker.position.x + 5;
  if (isMovingRight)   marker.position.x = marker.position.x - 5;
  if (isMovingForward) marker.position.z = marker.position.z + 5;
  if (isMovingBack)    marker.position.z = marker.position.z - 5;
}
```

Read through these lines to make sure you understand them. That bit of code says *if we detect a collision, then check the direction in which we're moving. If we're moving left, then reverse the movement that the avatar just did—go back in the opposite direction the same amount.*

With that, our avatar can walk up to the tree boundaries, but go no farther.



Yay! That might seem like some pretty easy code, but you just solved a very hard problem in game programming.

The Code So Far

If you'd like to double-check the code in this chapter, it's in *Code: Collisions*.

What’s Next

Collision detection in games is a really tricky problem to solve, so congratulations on getting this far. It gets even tougher once you have to worry about moving up and down in addition to left, right, back, and forward. But the concept is the same.

Usually we rely on code collections written by other people to help us with those cases. We’ll use such a code library when we get to the mini-games.

But first let’s put the finishing touch on our avatar game. In the next chapter, we’ll add sounds and scoring. Let’s get to it!

When you're done with this chapter, you will

- *Be able to add sounds to games*
- *Be able to add simple scoring to a game*
- *Have a silly game to play*

Chapter 11

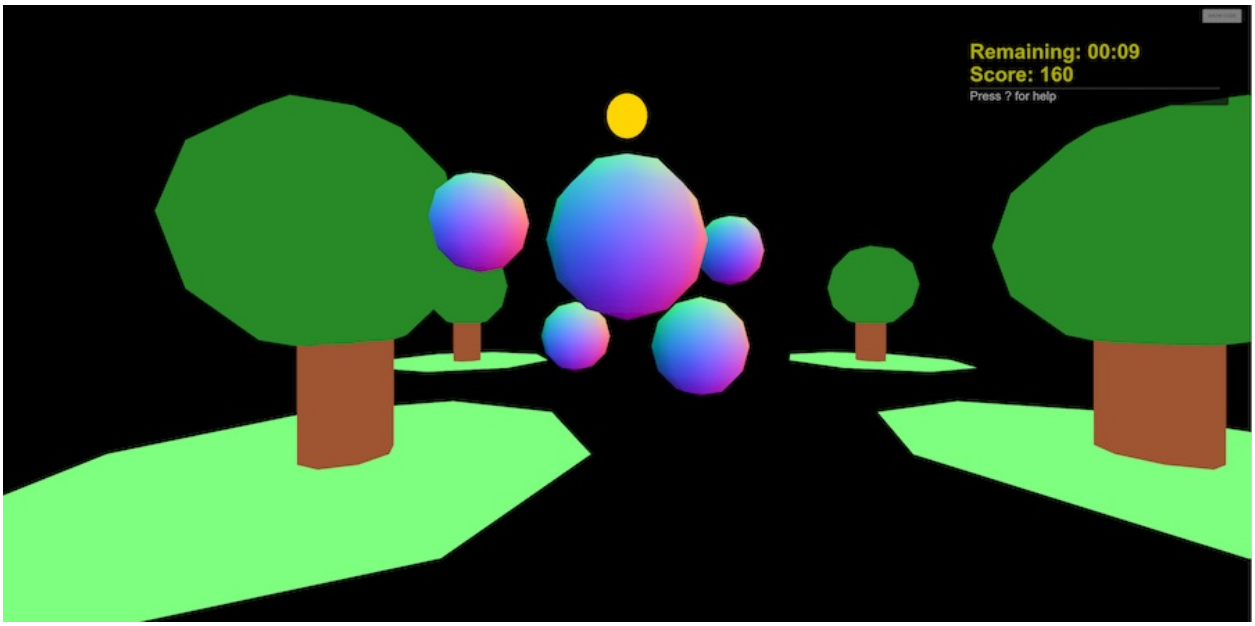
Project: Fruit Hunt

We have an avatar. We have trees. Surely there's fun for the avatar to have in those trees. So let's make a game of it!

We've spent 10 chapters talking about building an avatar and a game area. By now you have a pretty good idea what it takes to code these things. Now, this is not going to be The Greatest Game Ever. Our game will be somewhat simple, but enough to give you ideas for what you might want to add next.

Our game will challenge the avatar to get stuff out of those trees. The trees are hiding yummy fruit that the avatar wants. And if the avatar can get to the fruit in time, it will score points.

It will end up looking something like this:



Much thanks to fellow game programmer Sophie H. for coming up with the game concept used in this chapter!

Getting Started

To make this game, we need the avatar, the trees, and the collision-detection functions that we've worked on throughout this book. After Chapter 10, *Project: Collisions*, we have everything that we need to get started on this project. So let's make a copy of the Collisions project.

From the menu in the 3DE Code Editor, select MAKE A COPY and enter Fruit Hunt! as the new project name.

To keep score in this game, we'll need something new—a scoreboard. To make sounds in our game, we need something that can play audio. Rather than write our own scoreboard and sound code, we will load code collections to do it for us. In Chapter 9, *What's All That Other Code?*, we saw how to load code collections with the <script> tag at the very top of the code. We need to add two more.

We'll make these changes by starting a new line after the two <script> tags at the top of the page with src attributes. Add the following <script> tags to pull in the scoreboard and sound code.

```
<script src="/scoreboard.js"></script>
<script src="/sounds.js"></script>
```

Since this is just the "getting started" section of our program, those lines won't actually change anything in the game. To see the scoreboard, we need to configure it and turn it on. Let's do that next.

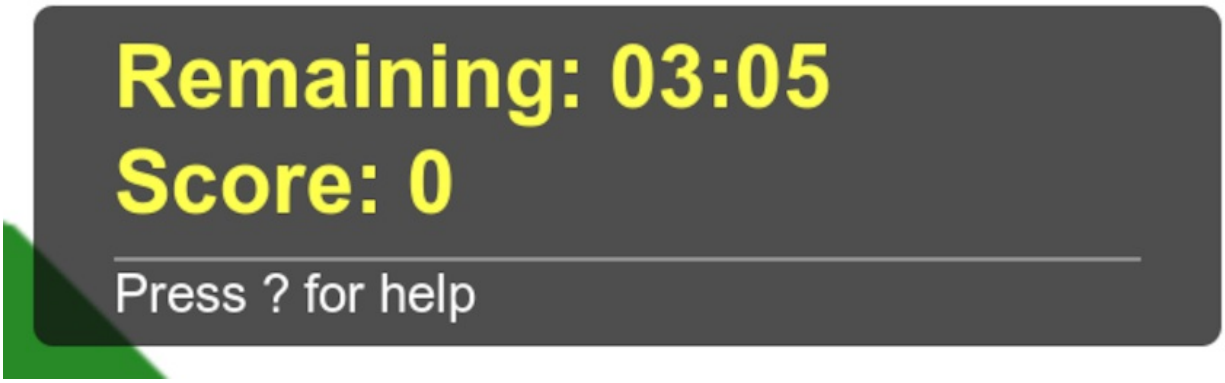
Starting a Scoreboard at Zero

The rest of the code in this chapter will go in the usual place, below the `START CODING ON THE NEXT LINE` line. More specifically, add the following scoreboard code after the avatar and marker code, but above the `makeTreeAt()` function.

```
var scoreboard = new Scoreboard();
scoreboard.countdown(45);
scoreboard.score();
scoreboard.help(
  'Arrow keys to move. ' +
  'Space bar to jump for fruit. ' +
  'Watch for shaking trees with fruit. ' +
  'Get near the tree and jump before the fruit is gone!'
);
```

This creates a new scoreboard. It tells the scoreboard to start a countdown timer, show the score, and add a help message. That should add a nifty-looking scoreboard to our screen, complete with the time remaining in the game (45 seconds), the current score (0), and a hint to players that they can press the question mark (`?`) key to get some help.

The scoreboard should look like the following:



Before making the game behave the way the help text says it should, we need to teach the game what to do when time runs out. To do so, add the following on the line after all of the scoreboard code:

```
scoreboard.onTimeExpired(timeExpired);
```

```
function timeExpired() {
  scoreboard.message("Game Over!");
}
```

Here, we tell the scoreboard that, on the moment that time expires, it should call the `timeExpired()` function. That function sets the scoreboard message to `Game Over!`.

That’s it for the scoreboard. Now, let’s figure out a way for the player to add points to the scoreboard.

Giving Trees a Little Wiggle

The goal of this game will be to find fruit in the trees. The fruit will be our game's treasure. At any given time, only one tree will have treasure. To show which tree it is, we'll give it a little shake. But first we need a list of trees.

In Chapter 10, *Project: Collisions*, we added a list of `notAllowed` boundaries above the `makeTreeAt()` function. Now, we add a list of `treeTops` in the same place.

```
  var notAllowed = [];
» var treeTops = [];
```

Next, inside the body of the `makeTreeAt()` function, let's push treetops onto this `treeTops` list.

```
  notAllowed.push(boundary);
» treeTops.push(top);
```

Now that we have a list of treetops, we can hide treasure in one and shake it. After the four calls to the `makeTreeAt` function, add the following function to update the treasure tree number:

```
var treasureTreeNumber;
function updateTreasureTreeNumber() {
  var rand = Math.random() * treeTops.length;
  treasureTreeNumber = Math.floor(rand);
}
```

`Math.random()` is an old friend by now. It returns a decimal number between 0.0 and 1.0. We multiply that by the total number of tree tops, which is currently 4. That gives us a random number between 0.0 and 4.0, which gets us close to what we need to pick a treetop from the list. But we need a little help from the `Math.floor()` function.

We saw in Chapter 7, *A Closer Look at JavaScript Fundamentals* that the first thing in a JavaScript list is at `0`. If there are four things in our `treeTops` list, the first is at `treeTops[0]` and the others would be at `treeTops[1]`, `treeTops[2]`, and

`treeTops[3]`. That means that we need to set `treasureTreeNumber` to `0`, `1`, `2`, or `3`. Right now we have random numbers like `0.1223`, `1.448`, `2.993`, and `3.8822`. What we want is just the number before the decimal point. That happens to be exactly what `Math.floor()` does!

Now that we have a way to randomly pick a treasure tree, we need to shake that tree. Add the following below the `updateTreasureTreeNumber()` function:

```
  function shakeTreasureTree() {
①   updateTreasureTreeNumber();

②   var tween = new TWEEN.Tween({shake: 0});
③   tween.to({shake: 20 * 2 * Math.PI}, 8*1000);
④   tween.onUpdate(shakeTreeUpdate);
⑤   tween.onComplete(shakeTreeComplete);
⑥   tween.start();
  }
```

That function has a lot of code in it, but it does just two things: update the treasure tree number and shake the tree. Each of the six lines does something important:

① Updates the tree number—by calling the `updateTreasureTreeNumber()` function that we just wrote.

② Sets the starting point, which is 0 shakes.

③ Sets the ending point and the amount of time it will take to get there. The endpoint is 20 times 2 × pi. The shaking is seen for 8 seconds, or 8000 milliseconds.

④ Sets the function that keeps getting called as the value changes. This function gets called hundreds of times as the tween moves between the start and end point.

⑤ Sets the function that gets called once at the end.

⑥ Starts the shaking!

It is no surprise that we use `updateTreasureTreeNumber()` to set the tree number—that's why we wrote that function. The rest of the function is another tween—though a little different than the tween we saw in Chapter 8, *Project: Turning Our Avatar*.

This tween is going to move back and forth. In Chapter 6, *Project: Moving Hands and Feet*, we saw that `Math.sin()` is great for back and forth. So this tween will use `Math.sin()` to shake the tree.

The tween starts the shake at 0 and moves to 20 times 2 × pi (`20 * 2 * Math.PI`). In Chapter 7, *A Closer Look at JavaScript Fundamentals*, we saw that going from 0 to 2 × pi in `Math.sin()` starts at 0, goes back and forth, then stops at 0. So 20 times that is 20 shakes back and forth.

When we used a tween to turn our avatar, we let the tween change the avatar's rotation directly. In this case, we want to shake the tree top back and forth. We need to call `Math.sin()` for that, which is why we have to tell the tween to call `shakeTreeUpdate()` with updates.

We have not actually created `shakeTreeUpdate()` yet, so add the following below `shakeTreasureTree()`:

```
function shakeTreeUpdate(update) {
  var top = treeTops[treasureTreeNumber];
  top.position.x = 50 * Math.sin(update.shake);
}
```

Whenever the tween calls this function with an update—probably around 50 times a second, this function finds the correct treetop and moves the position with `Math.sin()`.

When the tween is done, we've told it to call the `shakeTreeComplete()` function. Add that function as shown:

```
function shakeTreeComplete() {
  var top = treeTops[treasureTreeNumber];
  top.position.x = 0;
```

```
  setTimeout(shakeTreasureTree, 2*1000);
}
```

Again, this function grabs the current treetop. Then it moves the treetop back to the center of the trunk by setting its `x` position to 0.

The last part of `shakeTreeComplete` sets a timeout for 2 seconds. The `setTimeout()` function in JavaScript calls another function after a certain amount of time has passed. In this case, we wait for 2 seconds after the treasure tree stops shaking, then call the `shakeTreasureTree` function to pick a new treasure tree and start it shaking.

All that remains now is to call `shakeTreasureTree()` the first time. Add the following call to `shakeTreasureTree()` below `shakeTreeComplete()`:

```
shakeTreasureTree();
```

After you've typed in all of that, a different tree should be wiggling uncontrollably telling the player that there's treasure to collect. Now that we have treasure in the trees, let's give the avatar a way to grab that treasure.

Jumping for Points

To score points, the avatar needs to jump next to the current treasure-filled tree. We'll do two things when this happens: score some points and make a nice little animation of the treasure.

But first we need a key that will start a jump. To do this, we add the following `if` statement to the `sendKeyDown()` function:

```
if (code == 'Space') jump();
```

You can add that `if` code just above the other `if` statements that turn the avatar. With that, *if* the space key is pressed, the `jump()` function is called.

We add the `jump` function above the `isColliding` function.

```
function jump() {
  if (avatar.position.y > 0) return;
  checkForTreasure();
  animateJump();
}
```

If the avatar is already jumping—if its up-and-down `y` position is more than 0, then the code returns without doing anything. Otherwise, we check for nearby treasure and animate the jump on our screen.

To check whether the avatar is close enough to grab treasure, add the `checkForTreasure()` function below `jump()`.

```
function checkForTreasure() {
  var top = treeTops[treasureTreeNumber];
  var tree = top.parent;
  var p1 = tree.position;
  var p2 = marker.position;
  var xDiff = p1.x - p2.x;
  var zDiff = p1.z - p2.z;

  var distance = Math.sqrt(xDiff*xDiff + zDiff*zDiff);
  if (distance < 500) scorePoints();
```

```
}
```

The `checkForTreasure` function might look a little big, but it really does just two things:

1. Calculate the distance between the current treasure tree and the avatar
2. If that distance is less than 500, add some points to the scoreboard

Most of `checkForTreasure()` calculates the distance between avatar and tree. It finds the current treasure-filled treetop. Then we need to get the "parent" of the treetop—the tree trunk to which the treetop was added. We set two position values: the tree's position and the avatar's. Using those positions, we get the differences in `x` position and `z` position. Once we have the differences, the distance is easy: the square root of the `xDiff` squared plus the `zDiff` squared.

	Pythagorean Theorem Alert
	If you've already learned a bit of trigonometry, you may have recognized the Pythagorean theorem in the `checkForTreasure` function. We used it to find the distance between two points: the avatar and the active tree. If you haven't seen the Pythagorean theorem in school yet, now you have even more reason to pay attention when you do see it!

If the distance is less than 500, we call the `scorePoints()` function. For now, we keep that function very simple—if the time left is 0, we do nothing; otherwise we add 10 points to the scoreboard. Add `scorePoints()` after the `checkForTreasure` function.

```
function scorePoints() {
  if (scoreboard.getTimeRemaining() == 0) return;
  scoreboard.addPoints(10);
```

```
}
```

Be sure to add the first line in that function; otherwise players can get points after time has expired!

The last thing we need to do is animate the jump so we can see it on the screen. The avatar should start at `y` of 0 and end at `y` of 0, smoothly going up, then smoothly back down. If you guessed that it's time for another tween, you are correct!

Add the `animateJump()` function below `scorePoints()`.

```
function animateJump() {
  var tween = new TWEEN.Tween({jump: 0});
  tween.to({jump: Math.PI}, 400);
  tween.onUpdate(animateJumpUpdate);
  tween.onComplete(animateJumpComplete);
  tween.start();
}
```

This is very similar to the tween we used for shaking the trees. It starts the jump at 0. It ends the jump at `Math.PI` after 400 milliseconds. Remember that `Math.sin()` of 0 is 0. `Math.sin()` then increases to 1 and back down to 0 as we get to `Math.PI`, making that the perfect spot to end the jump.

Just like the tree shaking tween, we have functions to call when the tween is updating and when it completes. Add these functions after `animateJump()`:

```
function animateJumpUpdate(update) {
  avatar.position.y = 100 * Math.sin(update.jump);
}

function animateJumpComplete() {
  avatar.position.y = 0;
}
```

That should do it! If you hide the code, you can now move about, find the active tree, and jump to get treasure out of it. If you're very fast, you can even jump multiple times next to the active tree to get multiple points.

This is already a fun game, but we can add a few tweaks to make it even better.

Making Our Games Even Better

We've spent a good deal of time in this book adding animations to our avatar. We do this partly to understand important concepts like grouping objects, but also because this is a lot of what 3D game programmers do.

What makes a game compelling and fun enough to make players keep coming back is a combination of interesting gameplay and the occasional glimpses of realism. Our avatar doesn't *really* need to have hands and feet that move as in real life, but this animation helps make the game seem more real. In this game, the gameplay is pretty simple: press the `space bar` near the treasure to get points.

Adding Animation and Sound

How many tweaks you add is up to you, the game programmer. But for this chapter, let's add two things when the avatar gets the treasure-fruit: a little sound and an animation of the fruit the avatar just got.

Adding sound to the game is the easier of the two, so we'll tackle that first. In the `scorePoints` function, add a call to `Sounds.bubble.play`.

```
  function scorePoints() {
    if (scoreboard.getTimeRemaining() == 0) return;
    scoreboard.addPoints(10);
»   Sounds.bubble.play();
  }
```

You can find more information on the Sounds.js library in *Sounds.js*. The library has a fairly small number of sounds to pick from, but enough to get started writing games.

With that line added, we can score points and hear sound when the avatar jumps up to grab treasure-fruit. But we're not actually seeing any of that golden fruit.

To see and animate the fruit, we need to add the fruit to the avatar's marker, then tween it. The tween will be a little different than those we've done so far, as it

will animate *two* things. It will rise above the avatar and it will spin. The following code, which we can add after the `scorePoints` function, will do all of that:

```
var fruit;
function animateFruit() {
  if (fruit) return;

  fruit = new THREE.Mesh(
    new THREE.CylinderGeometry(25, 25, 5, 25),
    new THREE.MeshBasicMaterial({color: 'gold'})
  );
  marker.add(fruit);

  var tween = new TWEEN.Tween({height: 200, spin: 0});
  tween.to({height: 350, spin: 2 * Math.PI}, 500);
  tween.onUpdate(animateFruitUpdate);
  tween.onComplete(animateFruitComplete);
  tween.start();
}

function animateFruitUpdate(update) {
  fruit.position.y = update.height;
  fruit.rotation.x = update.spin;
}

function animateFruitComplete() {
  marker.remove(fruit);
  fruit = undefined;
}
```

It's another tween with functions! The only change here is that we're setting two different number properties: the `spin` and the `height` of the fruit. The `spin` starts at 0 and rotates around four times over the course of the entire animation. The fruit also rises from the position `200` to `350` on the screen over the course of the animation.

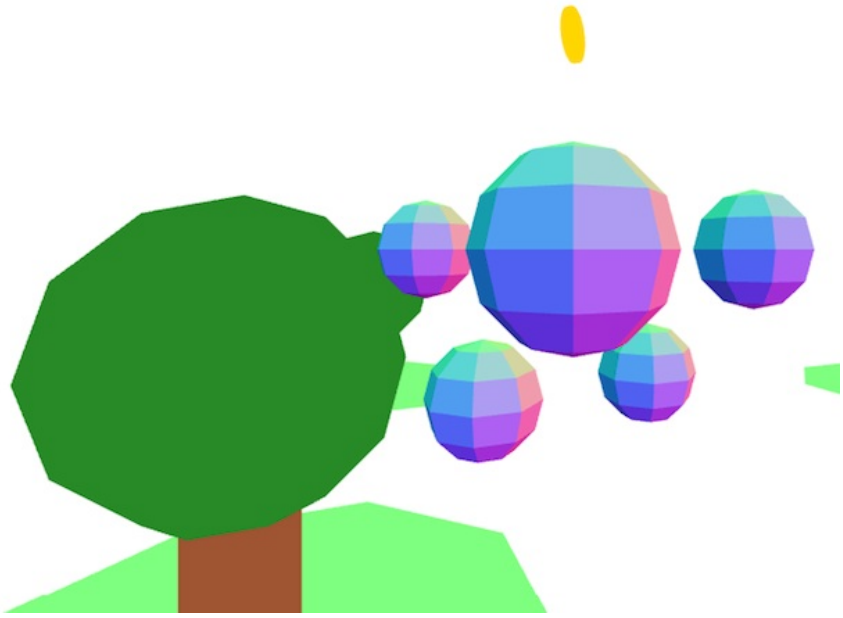
Of course, the `animateFruit` function needs to be called before it will do anything. Add a call to it at the bottom of the `scorePoints` function so it looks like this:

```
function scorePoints() {
```

```
    if (scoreboard.getTimeRemaining() == 0) return;
    scoreboard.addPoints(10);
    Sounds.bubble.play();
»   animateFruit();
  }
```

The result is a nice animation that plays as the avatar collects fruit.

Yay! Score!



What Else Can We Add?

This is it for our avatar that we built from scratch starting all the way back in Chapter 3, *Project: Making an Avatar*. That doesn't mean you can't make this game even better, though!

It's really easy to grab the fruit from a tree in this game. Perhaps you can add a tweak where the avatar is allowed only one piece of fruit from a tree? It might also be nice to penalize a player—think `subtractPoints`—if the avatar jumps when the tree is not active and wiggling. If you think the player is moving too fast or too slow, maybe look in the `sendKeyDown()` function for ways to improve that. You can build the game to have all sorts of nooks and crannies and prizes.

This is the job of the game designer, which happens to be you. Make a copy of the code so far and see what you can add to make the game work the way you want it to. How are you going to make this game great?

The Code So Far

If you'd like to double-check the code in this chapter, turn to *Code: Fruit Hunt*.

What’s Next

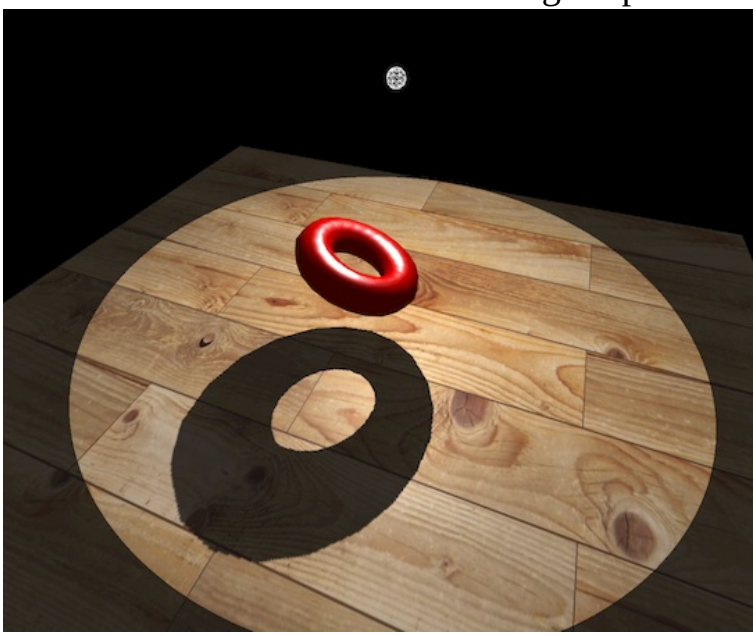
This may be it for our avatar projects, but there’s still plenty to do. Next we’ll explore more of the small touches that go into 3D programming, starting with lights, materials, and shadows.

When you're done with this chapter, you will

- *Be able to make shapes as shiny as you like*
- *Know how to make shadows in 3D games*
- *Be able to add textures to make 3D shapes even more realistic*

Chapter 12

Working with Lights and Materials

In this chapter we'll cover how to build interesting shapes and materials that look like this: 

All the way back in Chapter 1, *Project: Creating Simple Shapes*, we introduced the mesh as the combination of shapes and covers. Since then we've talked a lot about shapes—the different kinds, how we can combine them, how we can move them, and more. But aside from changing colors, we haven't really talked about what's possible with mesh covers.

It's pretty amazing what we can accomplish. So let's get to it!

Getting Started

Start a new project in 3DE. Choose the 3D starter project (with Animation) template from the menu. Then save it with the name Lights and Materials.

We will again work with the best shape in the world, the donut. So add the following code below START CODING ON THE NEXT LINE.

```
var shape = new THREE.TorusGeometry(50, 20, 8, 20);
var cover = new THREE.MeshPhongMaterial({color: 'red'});
var donut = new THREE.Mesh(shape, cover);
donut.position.set(0, 150, 0);
scene.add(donut);
```

If you've done everything correctly, then you should see a very dull red donut. You might be thinking, "That's it?" Well, of course that's not it!

The most important thing to note here is that we're using a new material, the MeshPhongMaterial. This material has a number of features that make it seem more real. Computers have to work harder to render this material, so only use it when you feel that your games will benefit from it.

	Important
	The MeshPhongMaterial gets its name from Bui Tuong Phong, the computer programmer who invented the techniques that make these materials look so pretty. We are able to do amazing things today because of the programmers who came before us. We owe them a lot for their work. We can pay them back by creating amazing things, sharing them, and helping others to build their own beautiful things. Just like Phong did!

Before we start playing with colors, let's spin the donut like we did in Chapter 1,

Project: Creating Simple Shapes. Add rotation code inside the `animate()` function.

```
var clock = new THREE.Clock();
function animate() {
  requestAnimationFrame(animate);
  var t = clock.getElapsedTime();

  // Animation code goes here...
  donut.rotation.set(t, 2*t, 0);

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

One last thing to set up is the location of the camera. It will be easier to see our scene if the camera is up higher, looking down. So above the `START CODING` line, add a `y` position for the camera and point the camera down at the center of the scene.

```
var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
camera.position.z = 500;
camera.position.y = 500;
camera.lookAt(new THREE.Vector3(0,0,0));
scene.add(camera);
```

That won't change too much. A dull red donut should still be spinning in the middle of the scene. Let's brighten things up a bit.

Emitting Light

The dull light in the scene is coming from a light at the very top of the code. Comment out that light.

```
    var light = new THREE.AmbientLight('white', 0.8);
»   //scene.add(light);
```

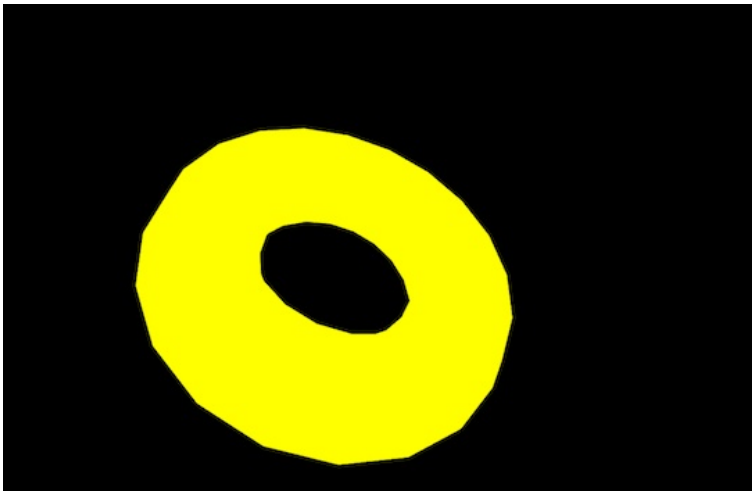
With no light, everything is black. We have a red donut spinning in a dark room and can't see it.

The first kind of light we can play with is "emissive" light. It's the light with which an object itself glows. This is like a white LED bulb that can shine different colors—even though the bulb itself is white, it can glow red, green, yellow…

So let's go back to our donut code to make our red donut… glow yellow.

```
    var shape = new THREE.TorusGeometry(50, 20, 8, 20);
    var cover = new THREE.MeshPhongMaterial({color: 'red'});
»   cover.emissive.set('yellow');
    var donut = new THREE.Mesh(shape, cover);
    donut.position.set(0, 150, 0);
    scene.add(donut);
```

With that, our red donut should be glowing yellow:



Emissive light has its uses, but other lights can be more fun. So comment out (or just delete) the emissive light.

```
// cover.emissive.set('yellow');
```

We’re back to a dark room. Instead of making our donut shine, let’s add some lights to shine on the donut.

Ambient Light

First, let's add some "ambient" light to the scene. A scene is rarely completely black. There's usually a little bit of illumination.

The light that we commented out at the very top of our code is that kind of light. So, let's uncomment it and change its brightness from a fairly high `0.8` (`1.0` is maximum brightness) to a dim `0.1` (`0.0` is no light).

```
» var light = new THREE.AmbientLight('white', 0.1);
» scene.add(light);
```

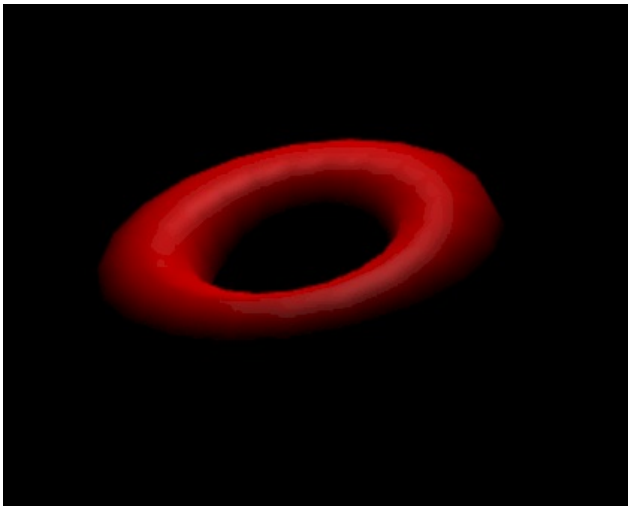
If you hide the code, you can see that this amount of ambient light is just barely enough to see the donut. That probably seems too dark, but it's perfect when working with other lights. That is just what we'll do in the next section!

Point Light

To increase the realism, let's add a "point light," which is kind of like a light bulb. Add the code for our point light below the donut code.

```
var point = new THREE.PointLight('white', 0.8);
point.position.set(0, 300, -100);
scene.add(point);
```

We're positioning the light above and behind the donut. The point light is bright, but not too bright at `0.8`. The result should be a pretty cool-looking donut:



One weird thing about programming with 3D lights is that the light is not coming from an object in the scene. We positioned the source of the light at X-Y-Z coordinates of `(0, 300, -100)`. The light comes from here, but there's no bulb to be seen.

It often helps to see a bulb, so let's add a glowing white bulb to the point light. The light is already there, so this is just a marker—a phony bulb. Add the phony bulb below the real point light code.

```
var shape = new THREE.SphereGeometry(10);
var cover = new THREE.MeshPhongMaterial({emissive: 'white'});
var phonyLight = new THREE.Mesh(shape, cover);
point.add(phonyLight);
```

Just because it is fake, doesn't mean that we have to make it look fake. We give it an emissive color of bright white so that it emits white—kind of like a real light bulb.

Shininess

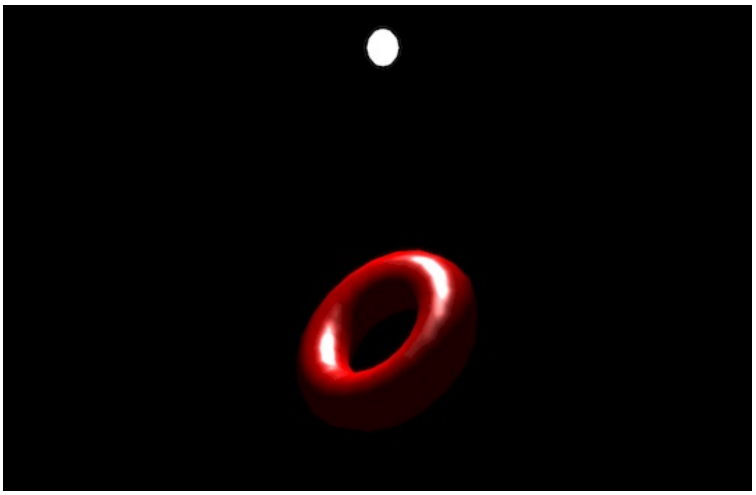
The donut is pretty cool looking already. Next, let's play with the shininess of the material wrapping the donut. In 3D programming, the shininess color is called the *specular color*. The specular color combines with the light shining on it. If a bright light shines on a dark specular color, it produces very little shine. This is how the material looks now. If we make a specular color brighter, then a bright light combines to produce shininess.

Set the specular color of our donut's material just after we create the cover.

```
  var shape = new THREE.TorusGeometry(50, 20, 8, 20);
  var cover = new THREE.MeshPhongMaterial({color: 'red'});
» cover.specular.setRGB(0.9, 0.9, 0.9);
  var donut = new THREE.Mesh(shape, cover);
  donut.position.set(0, 150, 0);
  scene.add(donut);
```

We are using red-green-blue colors like we did to build random colors back in Chapter 5, *Functions: Use and Use Again*. But here, we're not making colors, just variations of gray. When all of the RGB values are the same, you get gray. When they're near 0, an almost black gray is produced. When they are all close to 1, you get a very light, almost white gray.

And when that specular color is light, we see more of the shine as shown in the figure.



We’ve now met all the different kinds of color that work together to add realism to 3D objects. Let’s have a look at something even cooler: shadows.

Shadows

Drawing shadows is a lot of work for computers, so don't go crazy with them. Since they are so much work, they're disabled at first. We have to carefully work through each object that needs a shadow—and turn on shadows for the scene and light as well.

The first thing we need is some ground to see a shadow—we won't see a shadow unless the shadow falls on something. Add the code for the ground below the code for the donut, and above the light code.

```
var shape = new THREE.PlaneGeometry(1000, 1000, 10, 10);
var cover = new THREE.MeshPhongMaterial();
var ground = new THREE.Mesh(shape, cover);
ground.rotation.x = -Math.PI/2;
scene.add(ground);
```

That creates a plane, rotates it flat, and adds it to the scene—without any shadows.

To enable shadows, we follow these four steps:

1. Enable shadows in the renderer.
2. Enable shadows in a light.
3. Enable shadows in the object that casts a shadow.
4. Enable shadows in the object on which the shadow falls.

3D code will not even bother about shadows unless they are enabled in the renderer. To do so, set the `shadowMap.enabled` property in the renderer, which is above the `START CODING` line.

```
  var renderer = new THREE.WebGLRenderer({antialias: true});
» renderer.shadowMap.enabled = true;
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);
```

Unless we say otherwise, lights do not cast shadows. So let's enable shadows from our point light.

```
var point = new THREE.PointLight('white', 0.8);
point.position.set(0, 300, -100);
» point.castShadow = true;
scene.add(point);
```

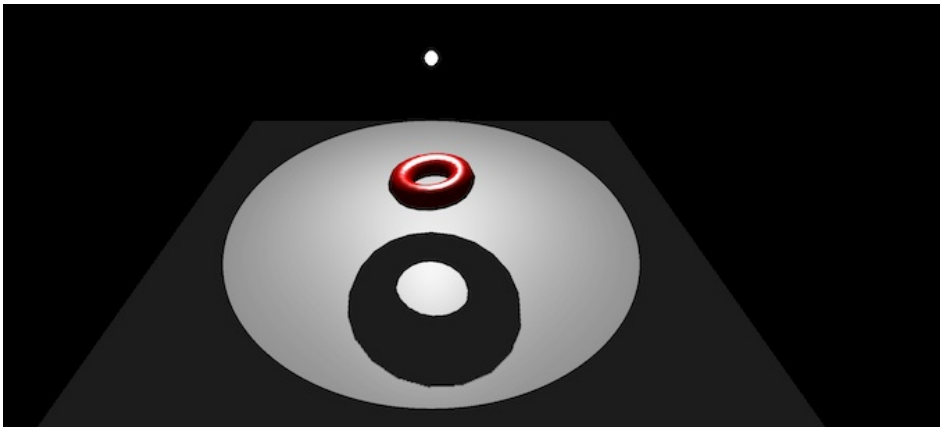
Next, we want our donut to cast a shadow, so enable the `castShadow` property on the donut.

```
var shape = new THREE.TorusGeometry(50, 20, 8, 20);
var cover = new THREE.MeshPhongMaterial({color: 'red'});
cover.specular.setRGB(0.9, 0.9, 0.9);
var donut = new THREE.Mesh(shape, cover);
donut.position.set(0, 150, 0);
» donut.castShadow = true;
scene.add(donut);
```

Last, we tell the ground that it *receives* a shadow.

```
var shape = new THREE.PlaneGeometry(1000, 1000, 10, 10);
var cover = new THREE.MeshPhongMaterial();
var ground = new THREE.Mesh(shape, cover);
ground.rotation.x = -Math.PI/2;
» ground.receiveShadow = true;
scene.add(ground);
```

With that, we should have a shadow for our spinning donut. If you don't see a shadow, check the JavaScript console *and* make sure that you've followed each of the four steps required for shadows.



Four steps might seem like a lot, but shadows require a lot of work by the computer. If every light makes shadows and every object casts a shadow and

every object can have a shadow fall on it…well, then the computer is going to use all of its power drawing shadows and have nothing left for the user to actually play games.

	Let's Play!
	Lights and materials have a lot of properties that interact with each other. The best way to understand them is to play! Change the color of the point light to purple. What happens if the light is blue, but the donut is red? Change the amount of specular RGB in the donut—first keeping all three numbers the same and then making the green and blue values (the second and third values) 0.

Spotlights and Sunlight

So far, we've seen two types of lights:

- Point, which acts like a light bulb
- Ambient, which provides a small amount of light everywhere

Two other lights are worth a quick look:

- Spotlight, which is great for focusing light at a single position
- Directional, which makes shadows like sunlight does

The difference between the lights is easier to see with objects in motion. So, down in the `animate()` function, add the following code to change the donut's position.

```
  donut.rotation.set(t, 2*t, 0);
» donut.position.z = 200 * Math.sin(t);
```

This moves the donut back and forward, even as it continues to spin.

Let's also dim the point light a bit, since we'll have more than one light shining on our donut.

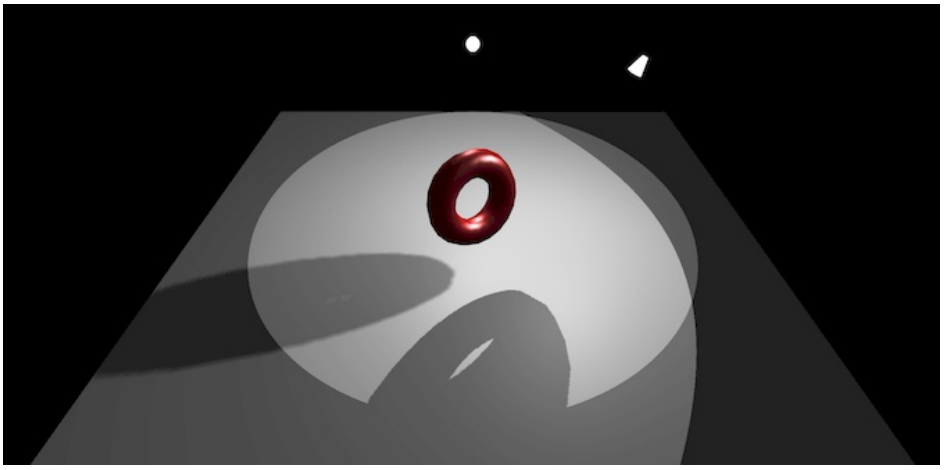
```
» var point = new THREE.PointLight('white', 0.4);
  point.position.set(0, 300, -100);
  point.castShadow = true;
  scene.add(point);
```

Now it's time to add a spotlight. Add the following code after the phony point light:

```
var spot = new THREE.SpotLight('white', 0.4);
spot.position.set(200, 300, 0);
spot.castShadow = true;
spot.shadow.camera.far = 750;
scene.add(spot);
```

This creates a white spotlight with a medium `0.4` brightness. It places the light

`200` to the right and `300` above the center of the scene.



This light also casts shadows. Right before we add this light to the scene, we do something a little different: we change the shadow's camera.

By now you should kind of expect that 3D code tries to do as little work as possible. One way that 3D code does this is by reusing camera code to draw shadows. Instead of seeing objects, shadow cameras see…shadows. Another way that 3D code limits the amount of work is by making shadow cameras not "see" very far.

	Let's Play!
	Try lowering the value of `spot.shadow.camera.far` to something like 500. The shadow from the spotlight should get cut off. The shadow camera simply doesn't bother to see—or draw—a shadow any further than that, which results in a broken shadow. A value of `750` turns out to be just enough to see a proper shadow in this scene.

If you like, you can add a phony spotlight after the spotlight code, just as we did with the point light.

```
var shape = new THREE.CylinderGeometry(4, 10, 20);
```

```
var cover = new THREE.MeshPhongMaterial({emissive: 'white'});
var phonyLight = new THREE.Mesh(shape, cover);
phonyLight.position.y = 10;
phonyLight.rotation.z = -Math.PI/8;
spot.add(phonyLight);
```

Notice with both the point light and spotlight, our donut's shadow gets longer as it gets further away from the light. That's how real-life shadows work with light bulbs and spotlights, but that's not how our shadows act in sunlight. In the sunlight, our shadows stay exactly the same length as we walk.

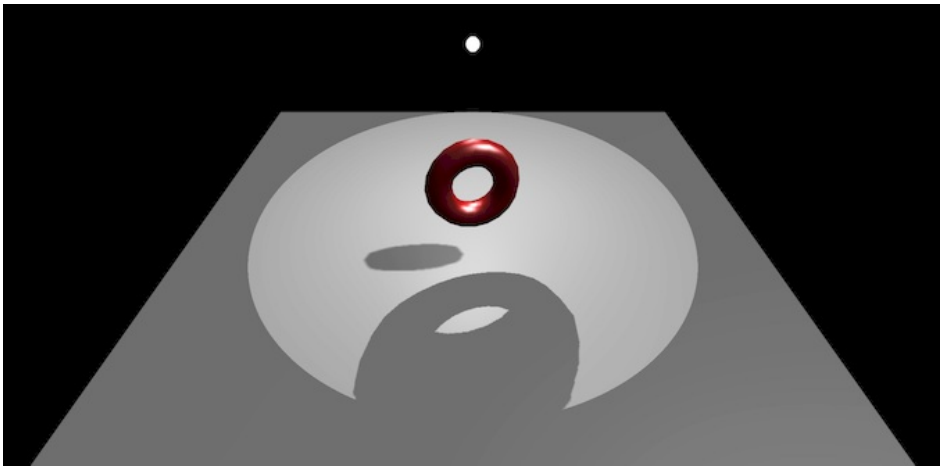
In 3D programming, sunlight is made with a "directional" light. To see that, comment out the line that adds the spotlight to the scene.

```
var spot = new THREE.SpotLight('white', 0.4);
spot.position.set(200, 300, 0);
spot.castShadow = true;
spot.shadow.camera.far = 750;
» //scene.add(spot);
```

Then, below the phony spotlight code, add a directional light, placing it at the same coordinates that we used for the spotlight.

```
var sunlight = new THREE.DirectionalLight('white', 0.4);
sunlight.position.set(200, 300, 0);
sunlight.castShadow = true;
scene.add(sunlight);
var d = 500;
sunlight.shadow.camera.left = -d;
sunlight.shadow.camera.right = d;
sunlight.shadow.camera.top = d;
sunlight.shadow.camera.bottom = -d;
```

As with the spotlight, we have to adjust the directional light's shadow camera. The spotlight's shadow camera was lazy about drawing shadows away from the light. The directional light's shadow camera is lazy everywhere else: up, down, left, and right! But once we've instructed the shadow camera to not be *too* lazy, we should have a shadow that stays exactly the same size as our donut moves back and forth.



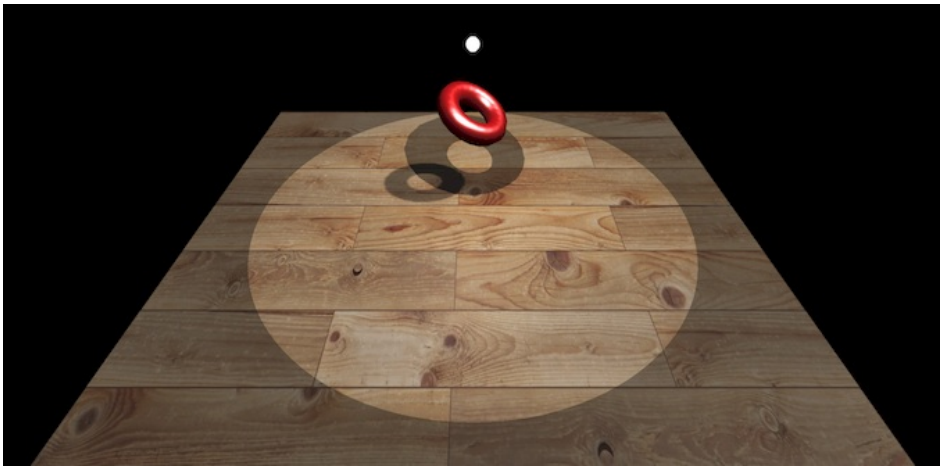
It’s usually not necessary to add a phony light for sunlight. The real Sun is way up high in the sky. We can pretend that the sun in our games is way up high as well.

Texture

This all looks pretty amazing. But we can still do one more thing to make it even better. Above the code for the ground, we'll load a "texture" image. Then, just after the cover is created, we assign that texture to the cover's `map` property.

```
» var texture = new THREE.TextureLoader().load("/textures/hardwood.png");
  var shape = new THREE.PlaneGeometry(1000, 1000, 10, 10);
  var cover = new THREE.MeshPhongMaterial();
» cover.map = texture;
  var ground = new THREE.Mesh(shape, cover);
  ground.rotation.x = -Math.PI/2;
  ground.receiveShadow = true;
  scene.add(ground);
```

That makes a very pretty wooden floor for our scene.



Calling the property "map" might seem a little strange. It gets the name from the way that 3D code applies square images—like `hardwood.png`—to shapes that are not squares. The image has to be "mapped" as best as possible onto spheres and cylinders. We'll see an example of that in Chapter 13, *Project: Phases of the Moon*.

Our 3D code collection includes several texture images that can replace `hardwood.png`: `brick.png`, `floor.png`, `grass.png`, `ground.png`, `metal.png`, `rock.png`, and `wood.png`. Try them out!

Further Exploration

It's easy to get lost playing with lights and materials. It is very fun. It can be frustrating, too, trying to get everything just right. This is an important part of game programming, but not nearly as important as gameplay.

The next sections are included in case you really enjoyed this chapter and want to play some more. Have fun, but don't go overboard with it!

Getting a Better View

This is an impressive creation. To get a better view, we can add "orbit" mouse controls to the scene. These controls let us click and drag with the mouse to spin the scene around the center.

Start by loading in the orbit controls code collection at the top of our code.

```
    <script src="/three.js"></script>
»   <script src="/controls/OrbitControls.js"></script>
```

Then, just above the `animate()` function, add the controls as follows:

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

That's it! If you hide the code, you should be able to click and drag with the mouse to rotate the scene. You can also scroll with the mouse or touchpad to zoom in and out.

Final Tweaks

To make a scene as realistic as possible without too much computing power, the first thing we'll do is remove our point light. Of the three lights that can cast shadows, point lights are the hardest on computers. Since they shine in all directions, the computer has to look in all directions for objects that cast shadows. That's a lot of work.

Point lights can still be useful, especially if they don't need to cast a shadow. But

we'll remove it here by commenting out the line that adds the point light to the scene.

```
// scene.add(point);
```

Let's also comment out the sunlight:

```
// scene.add(sunlight);
```

We've already seen all of the options that have the most impact on directional lights, so we don't need to tweak them further here.

Add the spotlight back to the scene, increase its brightness to `0.7`, and add the `angle` and `penumbra` settings:

```
» var spot = new THREE.SpotLight('white', 0.7);
  spot.position.set(200, 300, 0);
  spot.castShadow = true;
  spot.shadow.camera.far = 750;
» spot.angle = Math.PI/4;
» spot.penumbra = 0.1;
» scene.add(spot);
```

The angle describes how narrow the spotlight should be. The *penumbra* describes how fuzzy the edges of the spotlight should be.

	Let's Play!
	Play with those numbers. The angle cannot be bigger than `Math.PI/2`, so try numbers between that and `Math.PI/100`. The penumbra can be a number between 0 (not fuzzy light edges) and 1 (very fuzzy). Which numbers might be best for a search light? Which number would make the spookiest scene?

The end result, after you've rotated the scene, might look something like:

That, my fellow 3D programmers, is beautiful!

The Code So Far

If you'd like to double-check the code in this chapter, flip to *Code: Working with Lights and Materials*.

What’s Next

Lights and materials are advanced topics and we’ve only scratched the surface of what’s possible. They can have a big impact on games. Just don’t get too crazy with them. They can make the computer work hard, slowing down your game. Plus gameplay is more important than anything else.

This is an important lesson in any kind of programming, not just JavaScript gaming: just because you *can* doesn’t mean you *should*. The best programmers in the world know this rule well. And now you do, too!

Let’s put our new lighting skills to good use in the next chapter as we simulate the phases of the Moon.

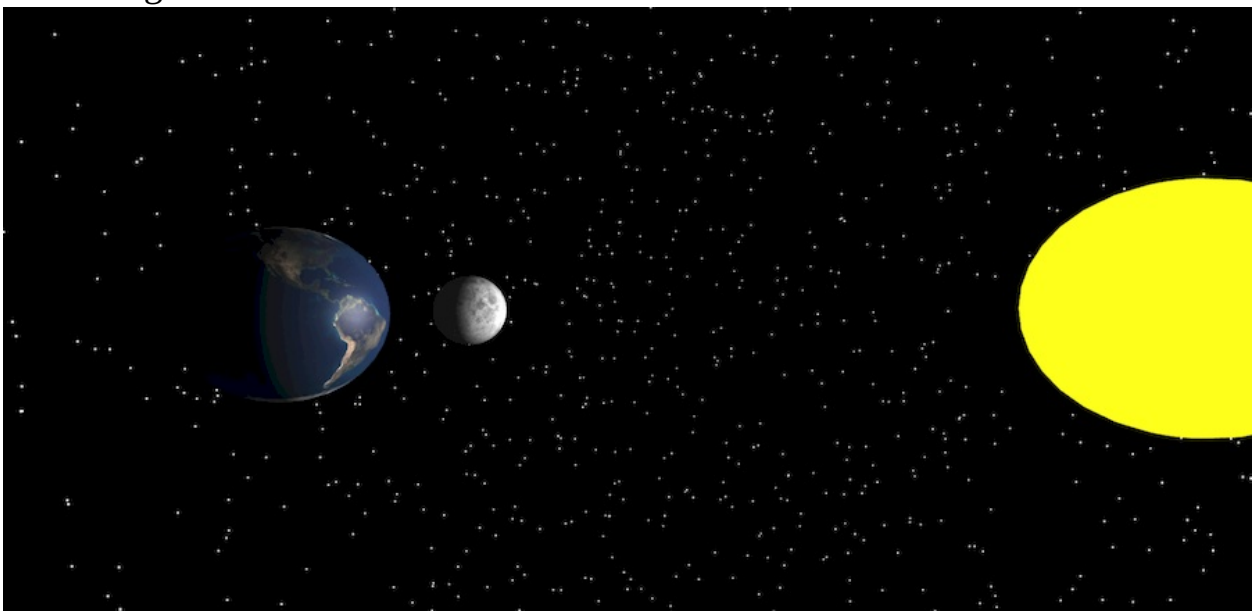
When you're done with this chapter, you will

- *Have an important trick for your 3D programmer's virtual toolbox*
- *Be able to switch between multiple cameras*
- *Know how and why to keep game and animation code separate*
- *Understand Moon phases better than most people (but not the Toy Story animators)*

Chapter 13

Project: Phases of the Moon

In this chapter we'll cover something every 3D programmer must learn at some point: how to visualize the Moon and its phases. It will end up looking something like this:



Why is the phase of the Moon important for 3D programmers? First, it's a simple enough problem—the Sun shines on the Moon, which revolves around Earth. Plus it lets us use our knowledge of lights and materials. It also lets us play the relative-positioning tricks that we practiced previously with an avatar's hands and feet.

Not important enough? Go watch *Toy Story*. It was the first full-length, computer-animated movie, and the programmers behind the project made sure they got it right. A waxing crescent Moon is visible behind Woody and Buzz as

they argue over being left by Andy at the gas station. If it was important enough for those moviemakers, it’s important enough for us!

Getting Started

Start a new project in 3DE. Choose the **3D starter project (with Animation)** template from the menu. Then save it with the name **Moon Phases**.

Before coding, let's dial down the brightness of the ambient light that's automatically added to our scene.

```
  var scene = new THREE.Scene();
  var flat = {flatShading: true};
» var light = new THREE.AmbientLight('white', 0.1);
  scene.add(light);
```

There is some ambient light in space—especially that reflected light near planets and moons. But nearly all of the light in the solar system comes from the Sun. We'll add the Sun shortly.

In addition to changing the light, let's also switch to an orthographic camera. When we talked about these cameras in Chapter 9, *What's All That Other Code?*, we mentioned that space games were a really good use for them. And here we are in a space game! Well, a space simulation, but close enough.

Comment out the **PerspectiveCamera** line (or delete it) and add the orthographic camera as shown:

```
  var aspectRatio = window.innerWidth / window.innerHeight;
» //var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
» var w = window.innerWidth / 2;
» var h = window.innerHeight / 2;
» var camera = new THREE.OrthographicCamera(-w, w, h, -h, 1, 10000);
» camera.position.y = 500;
» camera.rotation.x = -Math.PI/2;
  scene.add(camera);
» var aboveCam = camera;
```

We use the browser window's width and height to create the orthographic camera just like we did in *A Quick Peek at a Weirdly Named Camera*. We also move the camera 500 above the solar system—500 in the Y direction—instead

of 500 out in the Z direction. After that, we rotate the camera to point down to the center of the scene.

Don’t forget the last line that assigns `camera` to a second variable, `aboveCam`. We do this so that we can change `camera` between this above-the-solar-system camera and another one. Right now `camera` and `aboveCam` point to the same camera, but we’re going to change that in a bit.

That’s it for the setup. Let’s get started by adding the Sun to the center of the solar system.

The Sun at the Center

You're getting to be an expert with 3D objects, so you know the drill. Below the START CODING line, let's create a cover and a shape, combine them in a mesh, and add it to the scene.

```
var cover = new THREE.MeshPhongMaterial({emissive: 'yellow'});
var shape = new THREE.SphereGeometry(50, 32, 16);
var sun = new THREE.Mesh(shape, cover);
scene.add(sun);
```

Now that we have a better understanding of these things from Chapter 12, *Working with Lights and Materials*, we're using a Phong material for the Sun. This lets the Sun emit yellow light—so it can glow yellow.

The Sun has to do more than glow. It has to shine light on everything else in the solar system. As we saw in the last chapter, emitting a color does not shine a light in 3D programming. For that, we add a point light—a "light bulb" light.

```
var sunlight = new THREE.PointLight('white', 1.7);
sun.add(sunlight);
```

Next, we add the Earth to the scene. Again we do the usual combining of cover and shape to make a mesh, but with a fun twist.

```
var texture = new THREE.TextureLoader().load("/textures/earth.png");
var cover = new THREE.MeshPhongMaterial({map: texture});
var shape = new THREE.SphereGeometry(20, 32, 16);
var earth = new THREE.Mesh(shape, cover);
earth.position.x = 300;
scene.add(earth);
```

As we saw in Chapter 12, *Working with Lights and Materials*, we load an image of the Earth. Then we map that image onto the sphere using the `map` property of the Phong material.

It’s a little hard to see the Earth. We’re looking down at the north pole of the Earth with the Pacific Ocean facing the Sun. It will be easier to see once the Earth starts rotating and orbiting. So let’s do that next.

Game and Simulation Logic

We're going to make a new function to move and rotate the Earth. We could put this inside the `animate()` function, but separating game or simulation code from animation code will make our code more flexible. Game programmers often refer to this as separating *game logic* from animation code.

The properties that we'll control in this simulation are the speed at which it runs, whether or not the simulation is paused, and the number of days that have passed inside the simulation. We'll also need another internal clock to help track the time inside the simulation.

Add the following at the very end of our code, below the call to `animate()`:

```
var speed = 10;
var pause = false;
var days = 0;
var clock2 = new THREE.Clock();
```

Below that, we add a `gameStep()` function.

```
function gameStep() {
  setTimeout(gameStep, 1000/30);

  if (pause) return;

  days = days + speed * clock2.getDelta();

  earth.rotation.y = days;

  var years = days / 365.25;
  earth.position.x = 300 * Math.cos(years);
  earth.position.z = -300 * Math.sin(years);
}
gameStep();
```

Inside `gameStep()`, the first thing we do is set a timeout to call `gameStep()` again. We used `setTimeout()` in Chapter 11, *Project: Fruit Hunt* to wait before shaking a new treasure tree. Here, we use it to wait a little bit before updating our

simulation.

If we waited 1000 milliseconds, the simulation would update once a second. Here, we wait 1000 divided by 30. In other words we update our simulation 30 times a second. That might seem like a lot, but the animation updates closer to 60 times a second thanks to `requestAnimationFrame()`, which is a special function that we saw at the end of Chapter 5, *Functions: Use and Use Again*. Moving the game logic into a separate function that is called less often will help the animation run smoother.

In the rest of `gameStep()`, we return immediately—we leave the function without doing anything—if the simulation is paused. Next we update the number of days that have passed. We ask `clock2` how much time has passed since the last time we asked. The answer from `getDelta()` is going to be very small—close to 1/30 of a second. We take that small number, multiply it by the speed, and add it to the number of days that have already passed.

We can use the number of days by assigning it to the Earth's rotation. That is, the rotation about the Y axis—the up and down axis—will be equal to the number of days that have passed.

It takes a day for the Earth to make one rotation. It takes 365.25 days to make a full orbit around the Sun. In other words, after one day, the Earth is `1/365.25` of the way through its orbit—or `1/365.25` of the way through a year. So we assign the partial amount of `years` that have passed to `days / 365.25`.

Then we use our old trigonometry buddies sine and cosine to convert the partial amount of years into X and Z positions. If you haven't taken trigonometry in school yet, don't worry too much about how sine and cosine work. Just know that they can work together like this to make circles. And pay attention in math class when you talk sines and cosines—they're pretty amazing!

With all of that, we should have the Earth rotating pretty quickly and slowly revolving in its orbit.

Let’s do one more thing with our game logic before adding the Moon to the simulation. Add a keyboard listener after the call to `gameStep()`.

```
document.addEventListener("keydown", sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;

  if (code == 'Digit1') speed = 1;
  if (code == 'Digit2') speed = 10;
  if (code == 'Digit3') speed = 100;
  if (code == 'Digit4') speed = 1000;
  if (code == 'KeyP') pauseUnpause();
}

function pauseUnpause() {
  pause = !pause;
  clock2.running = false;
}
```

The `sendKeyDown` function changes the speed of the simulation from slow to fast when the `1`, `2`, `3`, and `4` keys are pressed. This will only work with the number keys on the main part of the keyboard. To work with a number pad, we would have to use `Numpad1`, `Numpad2`, `Numpad3`, and `Numpad4` instead of the strings that start with “Digit.” When the `P` key is pressed, it calls the `pauseUnpause()` function.

That `pauseUnpause()` function uses the exclamation point that we saw in Chapter 7, *A Closer Look at JavaScript Fundamentals* to switch `pause` back and forth. We also tell `clock2` that it’s no longer running. We don’t worry about telling `clock2` when it’s running again—the next time we ask it for `getDelta()`, `clock2` will automatically start up again.

With that, we should be able to hide the code and pause and unpause the simulation of the Earth rotating and revolving around the Sun. We can make it go slow by pressing `1` and can increase the speed with `2`, `3`, and `4`.

That’s pretty cool, but the *really* cool stuff starts next!

Local Coordinates

Let's try to build the Moon. We won't add it to the scene yet—just build it at first. Add the following below the code that adds the Earth to the scene:

```
var texture = new THREE.TextureLoader().load("/textures/moon.png");
var cover = new THREE.MeshPhongMaterial({map: texture, specular: 'black'});
var shape = new THREE.SphereGeometry(15, 32, 16);
var moon = new THREE.Mesh(shape, cover);
moon.position.set(0, 0, 100);
moon.rotation.set(0, Math.PI/2, 0);
```

That's similar to what we did with the Earth, but with a few changes. Obviously, we use an image of the Moon instead of the Earth for the mapped texture. We also make the specular color black because the Moon has almost no shininess. It's smaller than the Earth's geometry, and we rotate it so that the correct side of the Moon is facing us.

Let's try adding the Moon to the scene.

```
scene.add(moon);
```

You probably knew that wasn't going to work quite right. The Moon is added to the scene, but just sits there a little away from the center without moving.

Instead, we can change that last line so that the Moon is added to the Earth.

```
earth.add(moon);
```

That isn't right either. The Moon is now next to the Earth. It even stays with the Earth as the Earth revolves around the Sun. But the Moon is flying around the Earth as fast as the Earth rotates. Since we added the Moon to the Earth, whenever the Earth spins, the Moon goes right along with it. Since the real Moon takes a little over 29 days to go around the Earth, this isn't right.

To make this work, we need to borrow the ideas that we used in Chapter 8, *Project: Turning Our Avatar* to spin our avatar without moving the camera. In that game, we added the camera and player to a marker. We're going to do the

same here, but we'll call it *local coordinates* instead of marker. They mean the same thing, but local coordinates is the more common way of saying it in 3D programming.

So, above the code for the Earth, we need to add `earthLocal`.

```
var earthLocal = new THREE.Object3D();
earthLocal.position.x = 300;
scene.add(earthLocal);
```

Once we have local coordinates for the Earth, we add the Earth mesh to it. Change the code as shown to add the Earth to the new `earthLocal`:

```
var texture = new THREE.TextureLoader().load("/textures/earth.png");
var cover = new THREE.MeshPhongMaterial({map: texture});
var shape = new THREE.SphereGeometry(20, 32, 16);
var earth = new THREE.Mesh(shape, cover);
» //earth.position.x = 300;
» earthLocal.add(earth);
```

You may briefly lose sight of the Earth after making this change. That's OK.

We also comment out (or delete) the line that sets the Earth's X position. We no longer change the Earth's position. Instead we add the Earth to a local coordinate system, which we already positioned 300 away from the Sun.

Next, let's change the simulation logic in `gameStep()` to move the Earth's local coordinates instead of the Earth.

```
function gameStep() {
  setTimeout(gameStep, 1000/30);

  if (pause) return;

  days = days + speed * clock2.getDelta();

  earth.rotation.y = days;

  var years = days / 365.25;
» earthLocal.position.x = 300 * Math.cos(years);
» earthLocal.position.z = -300 * Math.sin(years);
```

```
}
```

At this point, the Moon is again spinning wildly around the Earth once a day. It might not seem like we made any progress at all, but this was a huge improvement. We can actually make the Moon orbit the Earth with just four lines of code.

Above the code for the Moon, add another local coordinate system—this one's for the Moon's orbit.

```
var moonOrbit = new THREE.Object3D();
earthLocal.add(moonOrbit);
```

This also adds the Moon's orbit to the Earth's local coordinate system. As the Earth's local coordinate system changes position, the Moon's orbit goes right along with it—just like our avatar did when its marker was moved.

That's two of the promised four lines. Next, we add the Moon to the Moon's orbit instead of to the Earth.

```
var texture = new THREE.TextureLoader().load("/textures/moon.png");
var cover = new THREE.MeshPhongMaterial({map: texture, specular: 'black'});
var shape = new THREE.SphereGeometry(15, 32, 16);
var moon = new THREE.Mesh(shape, cover);
moon.position.set(0, 0, 100);
moon.rotation.set(0, Math.PI/2, 0);
» moonOrbit.add(moon);
```

With that, the Moon is no longer revolving wildly around the Earth. It's not moving around the Earth at all—it always stays "below" the Earth as we're viewing the solar system from above.

That's three lines of code. Are we really going to get the Moon to revolve around the Earth with one line? Yup!

Down in the `gameStep()` function, we rotate the Moon's orbit just like we're rotating the Earth, only 29.5 times slower.

```
function gameStep() {
  setTimeout(gameStep, 1000/30);
```

```
    if (pause) return;

    days = days + speed * clock2.getDelta();

    earth.rotation.y = days;

    var years = days / 365.25;
    earthLocal.position.x = 300 * Math.cos(years);
    earthLocal.position.z = -300 * Math.sin(years);
»   moonOrbit.rotation.y = days / 29.5;
  }
```

With that, the Moon should be traveling with AND around Earth!

	Don’t Underestimate the Power of Local Coordinates
	This is the second time we’ve used local coordinates in our 3D programming. It won’t be the last. Moving a bunch of things together and still being able to move and rotate just some of the things is powerful. So powerful, it almost feels like cheating. If it feels like cheating in programming, you’re probably doing something right!

Multi-Camera Action!

We have a nice simulation of the Sun, Earth, and Moon from above the solar system. Let's add a second camera to the scene so that we can switch the view from above the solar system to a view from the Earth. This will help us better understand the phases of the Moon.

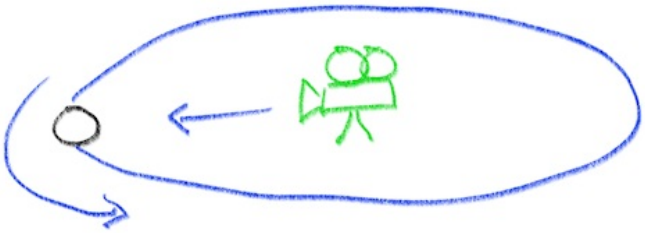
Remember from the Getting Started part of this chapter that we named our current camera `aboveCam`. Let's add `moonCam`, which will show the Moon from the Earth. Add the following code below the code for the Moon:

```
var moonCam = new THREE.PerspectiveCamera(70, aspectRatio, 1, 10000);
moonCam.position.z = 25;
moonCam.rotation.y = Math.PI;
moonOrbit.add(moonCam);

camera = moonCam;
```

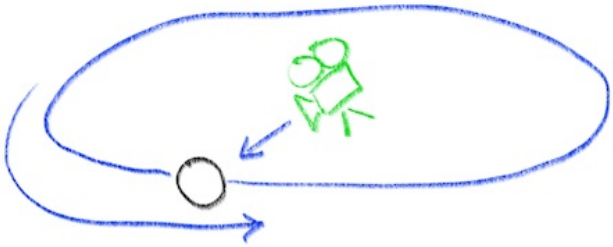
That last line assigns `camera` to our new `moonCam`. The `animate()` function uses that `camera` variable to render the scene. By assigning `camera` to `moonCam`, our view will switch to look at the Moon instead of looking at the solar system from overhead.

The *moon-cam* is a perspective camera to get a better view of the Moon. We position it 25 away from the center of the Earth's local coordinate system—that puts it just above the Earth's surface, which we gave a radius of 20. We rotate the camera to face the Moon and add the camera to the Moon's orbit.

We add the moon-cam to the Moon's orbit so that it always faces the Moon. As the Moon's orbit rotates, the moon-cam will rotate right along with it. You can think of the Moon's orbit—another local coordinate system—as a plate that we glue the camera and the Moon to.



When we spin the plate, the Moon and camera spin along with it:



Now that we have two cameras, let's add the code to switch back and forth. In the `sendKeyDown()` function that we wrote at the bottom of our code, add another `if` statement.

```
function sendKeyDown(event) {
  var code = event.code;

  if (code == 'Digit1') speed = 1;
  if (code == 'Digit2') speed = 10;
  if (code == 'Digit3') speed = 100;
  if (code == 'Digit4') speed = 1000;
  if (code == 'KeyP') pauseUnpause();
» if (code == 'KeyC') switchCamera();
}
```

When the `C` key is pressed, the `switchCamera()` function is called. We have to add that function next, which we do below the `pauseUnpause()` function at the very bottom of our code.

```
function switchCamera() {
  if (camera == moonCam) camera = aboveCam;
  else camera = moonCam;
}
```

It's pretty neat to see the Moon revolve around Earth while Earth revolves

around the sun and switch back and forth between above-cam and moon-cam. It’s especially helpful to slow the simulation to `1` or to pause the simulation and switch between cameras.

Bonus #1: Stars

What is a space simulation without stars? Stars in 3D games are pretty interesting. It would be a lot of work on the computer to simulate stars by generating 500 sphere meshes and moving them really far away. Instead of that, 3D programmers use a special material to create one shape with a lot of points in it. To make it easy on the computer, this *points* material only shows the points in the shape instead of smoothly covering the entire shape like regular materials.

Below the Moon code, start by creating the cover and shape for our star points.

```
var cover = new THREE.PointsMaterial({color: 'white', size: 15});
var shape = new THREE.Geometry();
```

The `PointsMaterial` is similar to other materials that we've seen—we can even set the color to white. The size that we specify is how big the points will be once we add the mesh to the scene. The points will be far away so we make them 15 to be big enough to see.

The shape that we're using is a basic geometry. It's not a cube. It's not a cylinder. It's not a sphere. It is not even really a shape yet. We have to add points to it before it has any structure at all.

We add that structure with a lot of help from our math friends.

```
var distance = 4000;
for (var i = 0; i < 500; i++) {
  var ra = 2 * Math.PI * Math.random();
  var dec = 2 * Math.PI * Math.random();

  var point = new THREE.Vector3();
  point.x = distance * Math.cos(dec) * Math.cos(ra);
  point.y = distance * Math.sin(dec);
  point.z = distance * Math.cos(dec) * Math.sin(ra);

  shape.vertices.push(point);
}
```

That might look like some daunting math, but it's not *that* bad. It loops over 500 numbers, creating a point for each at a distance of 4000. It does so randomly—picking angles, converting the angles to X-Y-Z coordinates, and adding those coordinates to our basic geometry shape.

If you're curious, the `ra` and `dec` variables are *right ascension* and *declination*. Astronomers use these two values to describe locations in the sky. Right ascension describes how far east or west a star or planet is—like longitude, but for the night sky. Declination describes how far north or south a thing is—like latitude. Using right ascension and declination, astronomers can pinpoint anything in the sky. We randomly pick values for both and then use sines and cosines to convert those angles into points very far away, making them seem like stars.

Points in a geometry are called *vertices*. After adding 500 random points to the vertices in our shape, we create a *points mesh* and add it to the scene.

```
var stars = new THREE.Points(shape, cover);
scene.add(stars);
```

With that, we have stars!

Bonus #2: Flying Controls

If we make a space simulation, we're going to want to fly through it, right? We can use the same controls that we used to fly through the planets in Chapter 5, *Functions: Use and Use Again*. Start by loading the fly controls code collection at the very top of our code:

```
  <body></body>
  <script src="/three.js"></script>
» <script src="/controls/FlyControls.js"></script>
```

We will need yet another camera for this. Just below the moon-cam code, add a ship-cam:

```
var shipCam = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
shipCam.position.set(0, 0, 500);
scene.add(shipCam);
```

Add the fly controls code just below that. (See *The Hitchhiker's Guide to the Galaxy* [Ada95]) to understand the importance of the constant 42.

```
var controls = new THREE.FlyControls(shipCam, renderer.domElement);
controls.movementSpeed = 42;
controls.rollSpeed = 0.15;
controls.dragToLook = true;
controls.autoForward = false;
```

In the `animate()` function, comment out (or delete) the line that gets the elapsed time. Also, add lines to get the change in time and to update our controls.

```
var clock = new THREE.Clock();
function animate() {
  requestAnimationFrame(animate);
» // var t = clock.getElapsedTime();

  // Animation code goes here...
» var delta = clock.getDelta();
» controls.update(delta);

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

```
}
animate();
```

Finally, add an `if` statement inside `sendKeyDown` that calls `fly()` if the `F` key is pressed.

```
function sendKeyDown(event) {
  var code = event.code;

  if (code == 'Digit1') speed = 1;
  if (code == 'Digit2') speed = 10;
  if (code == 'Digit3') speed = 100;
  if (code == 'Digit4') speed = 1000;
  if (code == 'KeyP') pauseUnpause();
  if (code == 'KeyC') switchCamera();
» if (code == 'KeyF') fly();
}
```

Put the `fly()` function at the very bottom of our code and have it switch the camera to `shipCam`.

```
function fly() {
  camera = shipCam;
}
```

We get a nice payoff here from when we put our simulation logic inside the `gameStep()` function instead of the `animate()` function. Since our fly controls are inside `animate()`, we can pause the simulation and still fly around exploring the paused simulation. Animation still works even though the planets are paused!

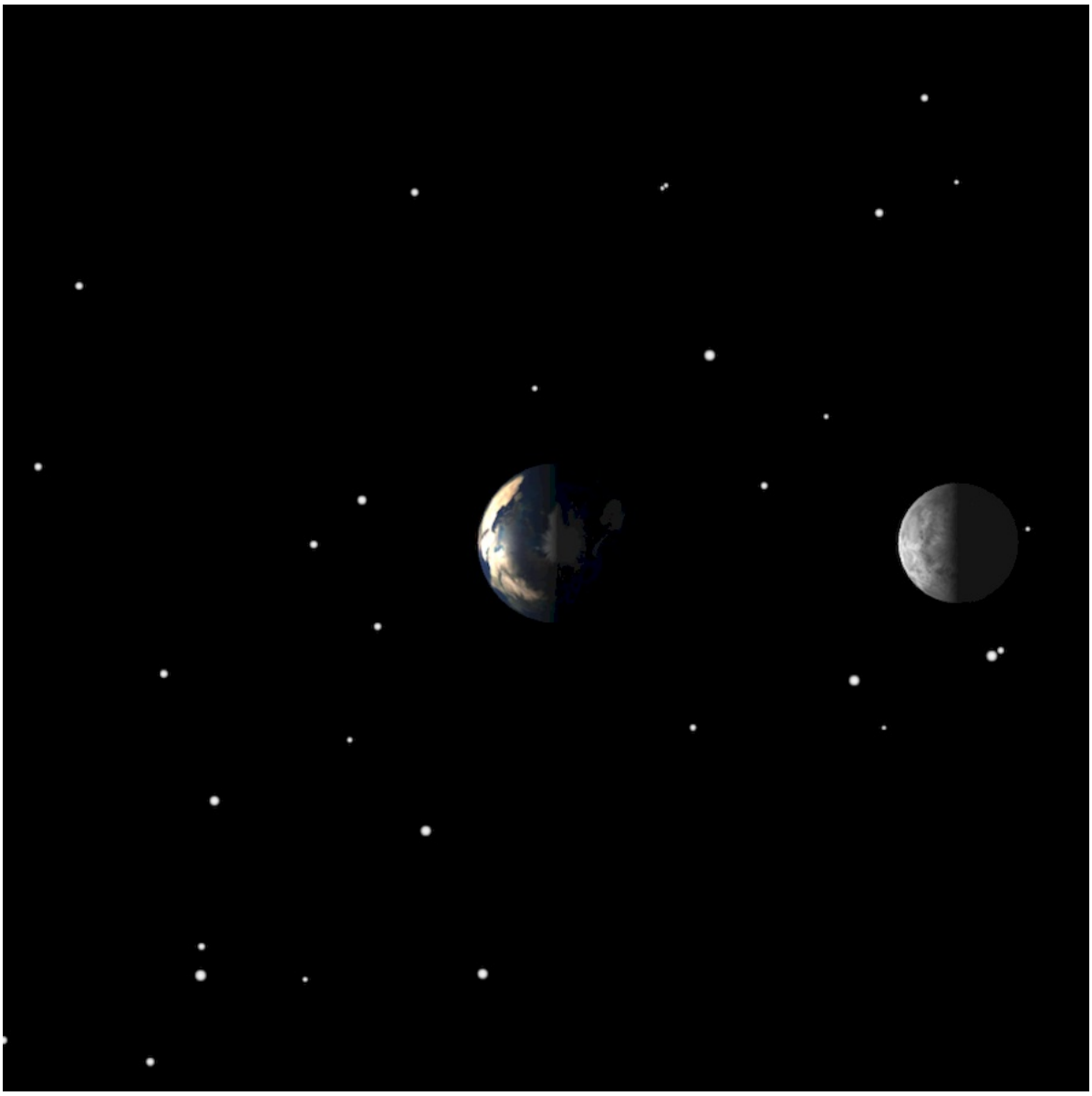
You can still use the `W`, `A`, `S`, `D`, `Q`, `E`, `R`, `F`, and arrow keys as described in *Bonus #2: Flight Controls*. Just don’t crash into the Earth!

Understanding the Phases

The Moon has four main phases: new, first quarter, full, and third quarter. *New* is when the Moon is in between Earth and the Sun. Since the Sun is shining on the side of the Moon that we can’t see, we don’t see the Moon at this time (also, it’s in the same part of the sky as the Sun).

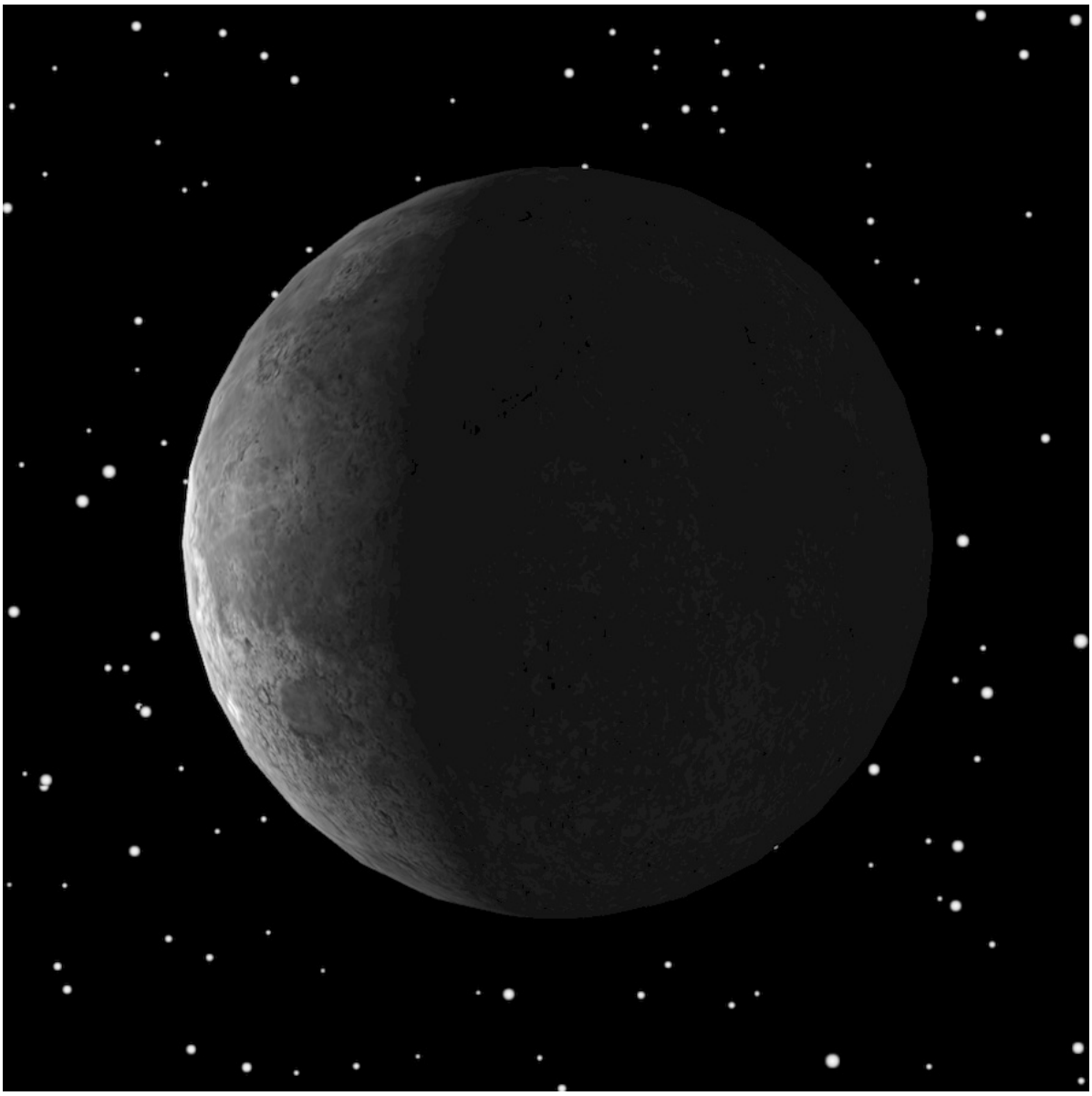
First quarter means that the moon is one-quarter of the way around its orbit. It does *not* mean that it’s a quarter lit up. As you can see, the first quarter Moon is actually half full.

When the Moon is two-quarters (or one-half) of the way around Earth, it's called *full*. The part of the Moon that we see is completely lit up.

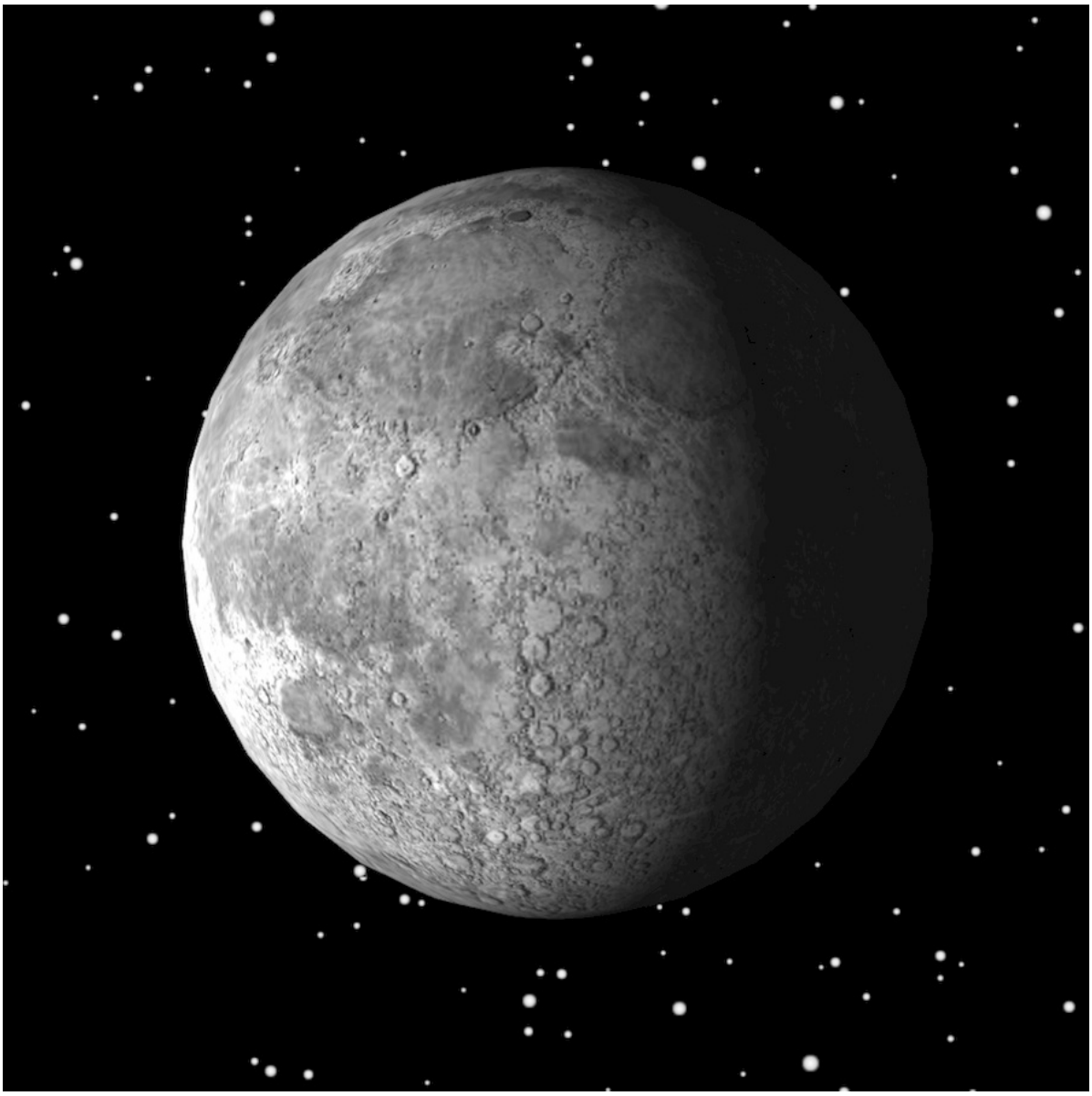


You know what *third quarter* is. The moon is three-quarters of the way around Earth, and again it’s half lit, though it’s the other half from the first quarter Moon.

Between the new Moon and the quarters, the Moon is a *crescent*.



Between the quarters and full Moon, the Moon is called *gibbous*.



When the lit side is growing, it’s said to be *waxing*. When it’s getting smaller, it’s said to be *waning*. And now you know just about everything there is to know about the Moon’s phases. Better yet, you have your own simulation!

Not Perfect, But Still a Great Simulation

You may have noticed that the Moon completely covers up the Sun after it moves through a waning crescent. That is, our simulation shows a total solar eclipse every month. This is a pretty good clue that our simulation is not perfect since solar eclipses are rare. What would we need to make it better?

First, the sizes and distances of the Earth and Moon in our simulation are way off. The Sun in our simulation has size 100. A correctly sized Earth would have a size of less than 1—ours is 40! Even if the Earth had the correct size, it's still way too close to the Sun. Our simulated Earth is 300 away from the Sun. To be accurate, for a Sun of size 100, the Earth should be 11,000 away! The Moon is too large as well. In the simulation, it's 75 percent the size of our Earth, but it should be 25 percent.

We did not make our simulation more accurate for three reasons. First, everything would have to be tiny to fit on the screen. The Earth and Moon would be tiny dots and even the Sun would be small if we zoomed out far enough to see everything. Second, 3D software would not be able to keep up with lights and shapes that have distances like 11,000 and sizes like 1. The code is designed to work with things that are easy to see. Third, you probably don't have a monitor that can show 11,000 pixels—giant 4K screens *only* have 3840!

Some other problems with the simulation are not as big. The Earth's orbit is not a circle—it's an ellipse/oval. This means that sometimes the Earth can be slightly closer or farther away from the Sun. Also the Moon's orbit is tilted compared to the Earth's, so sometimes it's above or below the Sun instead of causing an eclipse.

And despite all of that, this is still a *great* simulation. It helps us to understand how phases of the Moon work and it looks beautiful while doing so.

Sometimes Good Enough Is Great

It’s tempting as a programmer to make everything perfect. As you get experience, you’ll be amazed at how great “good enough” can be!

The Code So Far

If you'd like to double-check the code in this chapter, it's included in *Code: Phases of the Moon*.

What’s Next

This ends the space simulations in the book. Congratulations—you’ve made it through a grand tradition in 3D programming. Hopefully you also picked up a thing or two about space. More importantly for your computer skills, you’ve been introduced to the concept of local coordinates, which we’re definitely going to use in our games.

Speaking of games, let’s get started on some in the next chapter!

When you're done with this chapter, you will

- *Be able to build games with life-like motion including: jumping, falling, and colliding*
- *Understand how to build 2D games*
- *Have a challenging mini-game!*

Chapter 14

Project: The Purple Fruit Monster Game

In this chapter we'll make a two-dimensional jumping game. The player will use keyboard controls to make the Purple Fruit Monster jump and move to capture as much rolling fruit as possible, without touching the ground. It will end up looking something like this:

This might seem like a simple game to write, but we're going to use a lot of the skills and knowledge that we've been building up in the book. And to get the jumping and rolling and capturing, we're going to introduce a whole new level of sophistication to our code. This is going to be a fun one!

Getting Started

Start a new project in 3DE. Choose the `3D starter project (with Animation)` template and name this project `Purple Fruit Monster`. Do *not* use the template with physics for this project—we'll use that in later chapters.

Let's Make Physics!

This game will need two JavaScript code collections and some settings to go along with them. At the very top of the file, add two new `<script>` tags:

```
    <body></body>
    <script src="/three.js"></script>
①   <script src="/physi.js"></script>
②   <script src="/scoreboard.js"></script>
```

① We're going to use code to simulate real-life motion like falling, rolling, and colliding. We use the Physijs (physics + JavaScript) code collection so we don't have to write all the physics code ourselves.

② To keep score, we again use the scoreboard code collection.

At the top of the code from the `3D starter project` template, just below the `<script>` tag without an `src` attribute, make the changes noted below.

```
    // Physics settings
①   Physijs.scripts.ammo = '/ammo.js';
②   Physijs.scripts.worker = '/physijs_worker.js';

    // The "scene" is where stuff in our game will happen:
③   var scene = new Physijs.Scene();
④   scene.setGravity(new THREE.Vector3( 0, -250, 0 ));
    var flat = {flatShading: true};
    var light = new THREE.AmbientLight('white', 0.8);
    scene.add(light);
```

① A setting that enables Physijs to decide when things bump into each other.

② "Worker" code that runs in the background, performing all of the physics calculations.

③ Instead of a `THREEscene`, we need to use a `Physijsscene`.

④ Even with physics, we won't have gravity unless we add it to the scene. In this case, we add gravity in the negative Y direction, which is down.

Just one last bit of setup remains to get our scene to actively simulate physical activity. We'll wait until after we add some objects to the scene before working on that. First, let's convert from a 3D scene to a two-dimensional scene.

Vectors Are Direction and Magnitude

We're using `THREE.Vector3` to set gravity. We're going to use these a lot in this chapter. If you saw the first *Despicable Me* movie, then you already know what this is!

The bad guy in that movie is Vector. He chose his super villain name because a *vector* is an arrow with direction and magnitude (Oh, yeah!). That means a vector includes two pieces of information: the direction in which it points and how strongly it points in that direction.

The vector that describes gravity in this game points in the negative Y direction (down). It has a high magnitude (250), which means that things will fall down fairly quickly.

Let's Make 2D

The most important change to make for a 2D game is to use an orthographic camera. Back in Chapter 9, *What's All That Other Code?*, we talked about two

uses for these cameras: long distance views and 2D games. We used an orthographic camera for the long distances of space in Chapter 13, *Project: Phases of the Moon*. Now we use one for a 2D game.

Still working above the START CODING line, comment out (or delete) the code for the usual perspective camera. Then add an OrthographicCamera as shown.

```
» // var aspectRatio = window.innerWidth / window.innerHeight;
» // var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
» var w = window.innerWidth / 2;
» var h = window.innerHeight / 2;
» var camera = new THREE.OrthographicCamera(-w, w, h, -h, 1, 10000);
  camera.position.z = 500;
  scene.add(camera);
```

One other change that we'll make is a blue sky. To change the color of the entire scene, set the "clear" color—the color that's drawn when the scene is clear of anything else—to sky blue.

```
  var renderer = new THREE.WebGLRenderer({antialias: true});
  renderer.setSize(window.innerWidth, window.innerHeight);
» renderer.setClearColor('skyblue');
  document.body.appendChild(renderer.domElement);
```

With that, we're ready to start coding our jumping game.

Outline the Game

Let's think about how we can organize our code. To have made it this far in the book, you've written a lot of code. At times, it must have gotten difficult to move through the code to see what you've done. You're not the first programmer to run into this problem, and you won't be the last. Thankfully, you can learn from the mistakes of programmers before you.

	Keep Your Code Organized
	Programming is hard enough on its own. Don't make it harder by writing messy code. Organizing code doesn't matter too much with short programs. But code grows as new stuff is added. Organized code—indented and with functions defined in the order that they are called—is code that can grow.

One of the easiest ways to organize code is to treat it a little bit like writing. When you write an essay, it helps to start with an outline. After you have the outline, you can fill in the details.

When organizing code, it helps to write the outline first, then add the code below it. Since we're programming, our outlines are also written in code. Type in the following, including the double slashes, below START CODING ON THE NEXT LINE.

```
//var ground = addGround();
//var avatar = addAvatar();
//var scoreboard = addScoreboard();
```

This outline doesn't include everything in the game, but it's a lot of it. The ground will be the playing area. The avatar is the player in the game. The scoreboard will keep score and display useful information.

The double slashes at the beginning of each of those lines introduce a JavaScript

comment, which we first saw in *Code Is for Computers and Humans, Comments Are Only for Humans*. This means JavaScript will ignore those lines. This is a good thing since we haven't defined those functions yet.

Programmers call this "commenting out" code so it won't run. Programmers do this for many reasons. Here, we're doing it to outline code without causing errors.

We'll define these functions in the same order as they are in the code outline. This makes it easier to find code. By looking at the code outline, we know that the `addGround` function will be defined before the `addAvatar` function, which will be followed by `addScoreboard()`. The faster we can find code, the faster we can fix it or add things to it. When you write a lot of code, tricks like this can really help keep things straight.

After we build each function, we'll come back to this code outline to remove the double slashes before the function call—we'll "uncomment" the calls when they're ready.

Let's get started writing the code that matches this outline.

Adding Ground for the Game

The first function call in our code outline is to the `addGround` function. Just below the code outline (after the commented-out `//addScoreboard()` line), define that function as follows:

```
function addGround() {
  var shape = new THREE.BoxGeometry(2*w, h, 10);
  var cover = new THREE.MeshBasicMaterial({color: 'lawngreen'});
  var ground = new Physijs.BoxMesh(shape, cover, 0);
  ground.position.y = -h/2;
  scene.add(ground);
  return ground;
}
```

Our ground is a giant box. It is just like other boxes that we've built—with one twist. Instead of a plain, old `Mesh`, we use a `Physijs.BoxMesh` here. Meshes from the Physijs code collection are just like regular meshes, except that they can also behave like real, physical objects—they fall down and bounce off of each other.

When creating a Physijs mesh, we can pass a third argument in addition to the geometry and material. That third argument is the object's mass, which lets us make things very heavy or very light. In this case, we set the mass to a special number: 0. The 0 means that the shape never moves. If we didn't set the ground's mass to 0, the ground would fall down like anything else!

Unlike regular meshes, the different shapes have different physical meshes. The list includes `Physijs.BoxMesh`, `Physijs.CylinderMesh`, `Physijs.ConeMesh`, `Physijs.PlaneMesh`, `Physijs.SphereMesh`, and for all other shapes, `Physijs.ConvexMesh`.

Once this function is defined, we uncomment the call to `addGround()` in our code outline.

```
» var ground = addGround();
  //var avatar = addAvatar();
  //var scoreboard = addScoreboard();
```

If everything is working, we should see green ground with blue sky in the background as shown in the figure.

Build a Simple Avatar

In 3D programming, you can make simple graphics in two ways. We'll use both in this game—one kind for the Purple Fruit Monster and the other kind for the fruit. The simple graphic technique that we use for the Purple Fruit Monster is called a *sprite*.

In the `addAvatar()` function, we create an invisible, physics-enabled box mesh, then we add the sprite to the box mesh. Add this function below the `addGround()` function.

```
function addAvatar() {
  var shape = new THREE.CubeGeometry(100, 100, 1);
  var cover = new THREE.MeshBasicMaterial({visible: false});
  var avatar = new Physijs.BoxMesh(shape, cover, 1);
  scene.add(avatar);

  var image = new THREE.TextureLoader().load("/images/monster.png");
  var material = new THREE.SpriteMaterial({map: image});
  var sprite = new THREE.Sprite(material);
  sprite.scale.set(100, 100, 1);
  avatar.add(sprite);

  avatar.setLinearFactor(new THREE.Vector3(1, 1, 0));
  avatar.setAngularFactor(new THREE.Vector3(0, 0, 0));

  return avatar;
}
```

Sprites are graphics that always face the camera, which is exactly what we want our 2D avatar to do in this game. Sprites are super-efficient in graphics code. Any time we can use them, we make it much easier for the computer to do everything it needs to do to keep the game running smoothly.

Sprites start as tiny 1 by 1 things in a scene. To see this sprite, we scale it by 100 in the X and Y directions—we stretch it in the left/right and up/down directions.

The box mesh at the beginning of `addAvatar()` is doing all the work of falling

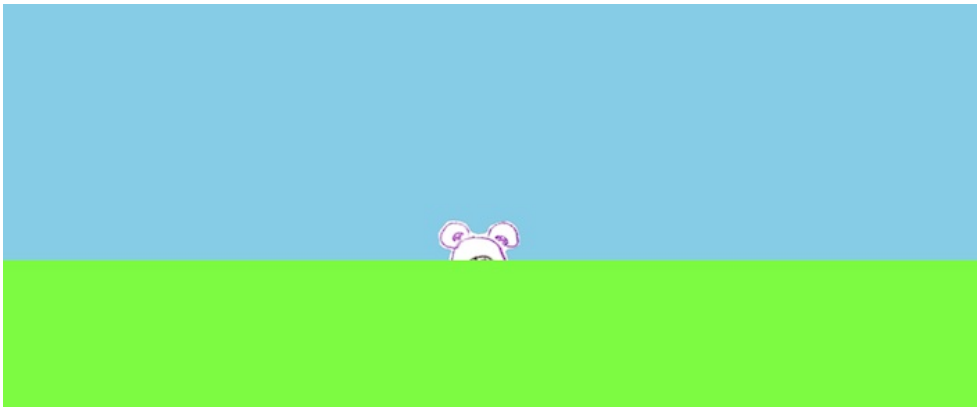
down, colliding with fruit, and colliding with the ground. We give it a small mass of `1` so it'll be easy to push with the controls that we'll add in a bit. It's invisible because we set `visible: false` in its material, but it's still there. We add the sprite to the box mesh so we know where the avatar is.

The last thing we do in `addAvatar()` is to set the angular and linear "factors." These factors say how much an object can rotate or move in certain directions. By setting the angular factor to all 0s, we're saying that our avatar cannot rotate in any direction. Even if it bounces off of spinning fruit, the avatar will always stay straight up and down. By setting the linear factor to two 1s and a 0, we're saying that the avatar can move in the X and Y directions, but not the Z direction. In other words, we're telling our 3D code that even though we're creating a three-dimensional shape, it will only move in two dimensions.

Move back up to the code outline and uncomment the `addAvatar` call.

```
    var ground = addGround();
»   var avatar = addAvatar();
    //var scoreboard = addScoreboard();
```

With that, we have a Purple Fruit Monster avatar…that's stuck in the ground.



Resetting the Position

We could have positioned the avatar in `addAvatar()`, but we just added it to the scene. Instead, we'll create a separate function to set the position. Why use a separate function? So we can re-use it!

When we talked about functions in Chapter 5, *Functions: Use and Use Again*, we said that some functions tell part of a story. The functions in our code outline do that—they tell the story of setting up the game.

Another kind of function is one that gets called over and over again. Let's create one of those functions that can start—or restart—the game by moving the avatar to its start position. Add the following after the `addAvatar()` function:

```
function reset() {
  avatar.__dirtyPosition = true;
  avatar.position.set(-0.6*w, 200, 0);
  avatar.setLinearVelocity(new THREE.Vector3(0, 250, 0));
}
```

The name of the `__dirtyPosition` property is something of a programmer's joke. We're making a mess here, so we say that it's "dirty." What's the mess? Since the avatar is physics-enabled, we normally couldn't change its position. We could push it to a new location, but we're not allowed to instantly move it from one place to another. But to reset the game, we need to do just that—immediately change the avatar's position back to start. Setting `__dirtyPosition` to `true` lets us do it.

`dirty` Starts with Two Underscores

Be sure to add two underscores before `dirtyPosition`. It's not `_dirtyPosition`. The setting is `__dirtyPosition`. If you use only one underscore, there will be no errors, but the movement controls won't work.

We move the avatar 60 percent of the way to the left: `-0.6` times the distance from the center to the left edge of the window. We also move it 200 above the ground. Finally we set the speed—the velocity—of the avatar. We use a vector to start it

with a speed of 250 straight up in the Y direction.

Add a call to `reset()` just below the code outline.

```
var ground = addGround();
var avatar = addAvatar();
//var scoreboard = addScoreboard();

» reset();
```

That should leave our avatar hovering above the ground, to the left of the screen.

Before adding controls to move the avatar, we have to tell our physics engine to actively simulate gravity and collisions.

Actively Simulate Physics

As we did in Chapter 13, *Project: Phases of the Moon*, we put our game code—our physics simulation—inside a function named `gameStep()`. Add it just below the `reset()` function.

```
function gameStep() {
  scene.simulate();
  setTimeout(gameStep, 1000/30);
}
gameStep();
```

Don't forget to call the `gameStep()` immediately after the function definition. Once that's coded, our avatar should start a little above ground, jump up a bit, then fall back down to the ground. We're now simulating real-world physics in

our game!

The `setTimeout()` inside `gameStep()` calls `gameStep()` after waiting for `1000/30` milliseconds—roughly 30 milliseconds. That will ask the Physijs code to update the positions of everything in the scene every 30 milliseconds. That sounds like a lot, but it's a nice balance. It's not so often that computers will start running slow. It's often enough so that the updates look smooth when animated.

Next, let's add some controls to move the Purple Fruit Monster about.

Movement Controls

To control the avatar, we use the `keydown` event listener that we saw in earlier chapters. Add the following code below the `animate` function:

```
document.addEventListener("keydown", sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;

  if (code == 'ArrowLeft') left();
  if (code == 'ArrowRight') right();
  if (code == 'ArrowUp') up();
  if (code == 'ArrowDown') down();
  if (code == 'Space') up();
  if (code == 'KeyR') reset();
}

function left()  { move(-100, 0); }
function right() { move(100,  0); }
function up()    { move(0, 250); }
function down()  { move(0, -50); }

function move(x, y) {
  if (x > 0) avatar.scale.x = 1;
  if (x < 0) avatar.scale.x = -1;

  var dir = new THREE.Vector3(x, y, 0);
  avatar.applyCentralImpulse(dir);
}
```

There's nothing fancy with the `sendKeyDown()` event listener. The `left()`, `right()`, `up()`, and `down()` functions that get called are pretty simple as well. They call a `move()` function with different amounts to move in the X and Y directions.

The `move()` function is a little interesting. The first two lines set the avatar's X scale if we're moving in the X direction. This flips the Purple Fruit Monster's image to face left or right, depending on the direction in which we're moving.

The last two lines of `move()` push the avatar in the proper direction. First, we calculate the direction. For example, when the left arrow key is pressed, the `left()` function is called. The `left()` function calls `move(-100, 0)`, which tells move to set `x` to `-100` and `y` to 0. The `dir` value is then set to a vector pointing (-100, 0, 0), which is 100 to the left. Applying a central impulse in that direction means a quick push in the center of the avatar. Pushing in the center of an object is easier than pushing an edge. Since we're programmers and we like easy, we push in the center.

With that, we should be able to hide the code and move the avatar up, down, left, and right.

Add Scoring

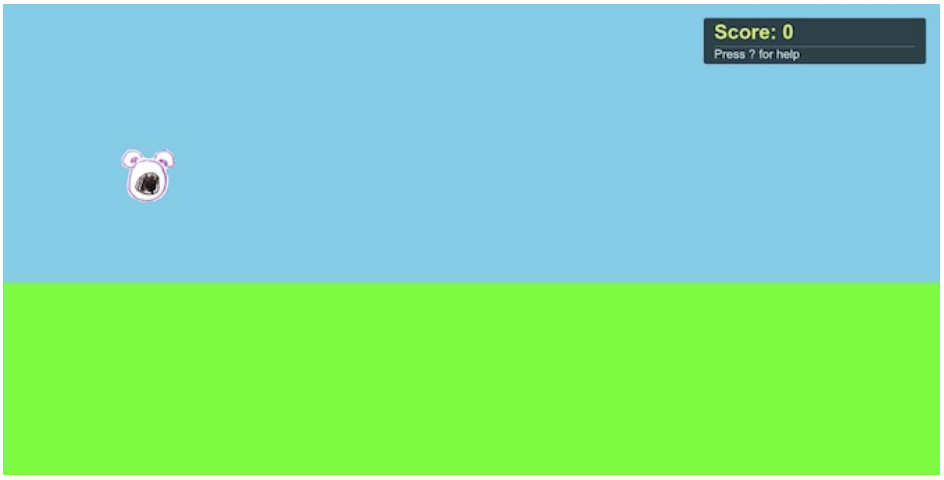
To complete the code outline, we next add the scoreboard. Put this below the `addAvatar()` function and above the `reset()` function.

```
function addScoreboard() {
  var scoreboard = new Scoreboard();
  scoreboard.score();
  scoreboard.help(
    "Use arrow keys to move and the space bar to jump. " +
    "Don't let the fruit get past you!!!"
  );
  return scoreboard;
}
```

This is similar to the scoreboard we used in Chapter 11, *Project: Fruit Hunt*, so the code should look familiar. Uncomment the `addScoreboard` function in the code outline.

```
var ground = addGround();
var avatar = addAvatar();
» var scoreboard = addScoreboard();

reset();
```

You should now see a scoreboard showing 0 points.



Gameplay

At this point, we're done with the code outline and we have the basics for a solid 2D game. We have the playing area, the avatar (including controls), and a way to keep score. To make the game interesting, we still need to do a bit more work. Next, we'll add gameplay. We're going to roll out some fruit and challenge the player to make the avatar eat as much as possible without touching the ground.

Launching Fruit

First, to create fruit, add a function below the `reset()` function.

```
function makeFruit() {
  var shape = new THREE.SphereGeometry(40, 16, 24);
  var cover = new THREE.MeshBasicMaterial({visible: false});
  var fruit = new Physijs.SphereMesh(shape, cover);
  fruit.position.set(w, 40, 0);
  scene.add(fruit);

  var image = new THREE.TextureLoader().load("/images/fruit.png");
  cover = new THREE.MeshBasicMaterial({map: image, transparent: true});
  shape = new THREE.PlaneGeometry(80, 80);
  var picturePlane = new THREE.Mesh(shape, cover);
  fruit.add(picturePlane);

  fruit.setAngularFactor(new THREE.Vector3(0, 0, 1));
  fruit.setLinearFactor(new THREE.Vector3(1, 1, 0));
  fruit.isFruit = true;

  return fruit;
}
```

That's a fair amount of typing, but it should be familiar. We're creating a sphere as a physics-enabled stand-in for fruit. As with the avatar, we make this sphere invisible. To enable physics on the sphere, we create a `Physijs.SphereMesh`. Before adding it to the scene, we position it off to the right of the window and just above the ground.

For the fruit image, we use the second way of adding simple, 2D graphics. After

loading the image, we map it into a basic material and add that to a simple plane geometry. We can't use a sprite here as we did with the avatar because the image will need to rotate as the ball rotates.

We again set angular and linear factors so that the fruit can only move and rotate two-dimensionally. The angular factor only sets `1` for the Z axis. This means that the fruit will be able to spin like the hands on an analog clock.

Before returning the fruit from the function, we set an `isFruit` property. That will help us later when we need to decide whether the avatar is colliding with the ground or one of these pieces of fruit.

To make sure that all of this is typed in correctly, add a call to `makeFruit()` after the function definition. You should see the fruit on the very right edge of the screen and there should be no errors in the JavaScript console. If everything is OK, remove the `makeFruit()` call.

Next, we need to launch the fruit. Add the `launchFruit()` function *above* the `makeFruit()` function definition.

```
function launchFruit() {
  var speed = 500 + (10 * Math.random() * scoreboard.getScore());
  var fruit = makeFruit();
  fruit.setLinearVelocity(new THREE.Vector3(-speed, 0, 0));
  fruit.setAngularVelocity(new THREE.Vector3(0, 0, 10));
}
```

We start by making fruit with the `makeFruit()` function we just wrote. We calculate the speed as 500 plus a little extra. The little extra is a random number that gets bigger as the score gets bigger. A game should get harder the longer the player plays. We make it harder by rolling the fruit faster and faster!

We apply the speed with `setLinearVelocity()`. The motion needs to be from right to left, so we set the X direction to the negative of the speed setting. Last, we give the fruit a little spin.

We still need to call this function. So, below the `launchFruit()` function, add two

calls.

```
launchFruit();
setInterval(launchFruit, 3*1000);
```

The first call to `launchFruit()` launches a single fruit right away. Next, we use `setInterval()` to keep calling `launchFruit()` every 3 seconds. The `setTimeout()` function that we've already seen calls a function once after a delay. The `setInterval()` function does the same thing, but keeps calling over and over.

With that, we have lots of fruit heading at the Purple Fruit Monster. We can even use the keyboard controls to bounce off the fruit. Next, we need to keep score when that happens.

Eating Fruit and Keeping Score

All the way at the bottom of our code—below the `move()` function, add code to send collisions to the rest of our code.

```
avatar.addEventListener('collision', sendCollision);
function sendCollision(object) {
  if (object.isFruit) {
    scoreboard.addPoints(10);
    avatar.setLinearVelocity(new THREE.Vector3(0, 250, 0));
    scene.remove(object);
  }
}
```

This looks a lot like the keydown event listener that we're using to respond to keyboard actions. Instead of processing and sending keyboard events to the game, here we send collision events.

If the avatar collides with fruit, we add 10 points to the score, give the avatar a little bump up, and remove the fruit from the screen (because the Purple Fruit Monster ate the fruit).

Hide the code and try it out!

Game Over

We have all the elements we need for this game except one: a way to lose. Let's add code so the game ends when the Purple Fruit Monster touches the ground.

When the game ends, we need a way to tell various parts of the code to stop doing what they're doing—at least until the game is reset. We'll use a `gameOver` variable for that. Add it above the code outline.

```
» var gameOver = false;

  var ground = addGround();
  var avatar = addAvatar();
  var scoreboard = addScoreboard();

  reset();
```

The game is not over when it first starts, so we set `gameOver` to `false` here.

One of the ways for the game to end is when the avatar hits the ground or when it *collides* with the ground. So in the `sendCollision()` function at the bottom of our code, add a second collision check as shown:

```
  avatar.addEventListener('collision', sendCollision);
  function sendCollision(object) {
»   if (gameOver) return;

    if (object.isFruit) {
      scoreboard.addPoints(10);
      avatar.setLinearVelocity(new THREE.Vector3(0, 250, 0));
      scene.remove(object);
    }
»   if (object == ground) {
»     gameOver = true;
»     scoreboard.message(
»       "Purple Fruit Monster crashed! " +
»       "Press R to try again."
»     );
»   }
  }
```

If the object the avatar is colliding with is the ground, then the game is over. We also update the scoreboard with a helpful message. And note that we're returning

immediately from the function if the game is over. There's no reason to check for collisions when the game is over!

At this point, we have a way to say that the game is over, but nothing stops. Animation still happens and fruit still rolls. We need to teach our code what it means when the game is over.

Back up in the `animate()` function, add a check that returns right away—before anything is animated—if the game is over.

```
  var clock = new THREE.Clock();
  function animate() {
»   if (gameOver) return;

    requestAnimationFrame(animate);
    var t = clock.getElapsedTime();

    // Animation code goes here...
    renderer.render(scene, camera);
  }
  animate();
```

Do the same thing above `animate()`, in the `launchFruit()` function.

```
  function launchFruit() {
»   if (gameOver) return;

    var speed = 500 + (10 * Math.random() * scoreboard.getScore());
    var fruit = makeFruit();
    fruit.setLinearVelocity(new THREE.Vector3(-speed, 0, 0));
    fruit.setAngularVelocity(new THREE.Vector3(0, 0, 10));
  }
```

Then, we have to teach the `reset()` function how to start things back up.

```
  function reset() {
    avatar.__dirtyPosition = true;
    avatar.position.set(-0.6*w, 200, 0);
    avatar.setLinearVelocity(new THREE.Vector3(0, 250, 0));

    scoreboard.score(0);
    scoreboard.message('');
```

```
»   var last = scene.children.length - 1;
»   for (var i=last; i>=0; i--) {
»     var obj = scene.children[i];
»     if (obj.isFruit) scene.remove(obj);
»   }
»
»   if (gameOver) {
»     gameOver = false;
»     animate();
»   }
  }
```

We're doing two things here: removing old fruit and restarting the game.

To remove the fruit, we loop over all of the "children" in the scene—all of the objects that we've added to the scene. At any step in the loop, if we find that the object is a piece of fruit, we tell the scene to remove it. This way, when we reset the game, no old fruit is left.

Note that we have to go backward when removing things from a list in JavaScript—otherwise we risk skipping things that we didn't mean to skip. The list of things in the scene might be:

```
0: Fruit #1
1: Fruit #2
2: Ground
```

If we start the for-loop from 0, the first time through the list, `i` would be 0, and we would remove Fruit #1 from the scene. Then, the list would look like this:

```
0: Fruit #2
1: Ground
```

The next time through the loop, `i` would increase to `1`. Checking object number `1`, we would find the ground. Since that's not fruit, our code would not remove it from the scene. And then our loop would stop since there're no more items in the list.

But wait! We skipped right over Fruit #2. By removing something at the

beginning of the list, we shift everything else in the list down by one. By the time the loop starts again, we have skipped an old fruit.

We don't have this problem doing the reverse. Instead of starting with the first item in the list and increasing the `i` variable each time, we start with the last item and *decrease* `i` after each loop. The example in reverse would start like this:

```
2: Ground
1: Fruit #2
0: Fruit #1
```

When `i` is 2, we don't do anything because the ground is not a piece of fruit. When `i` is `1`, we remove Fruit #2, leaving the list like this:

```
1: Ground
0: Fruit #1
```

Then, in the next loop, `i` would be 0 and we'd remove Fruit #1, leaving us with:

```
0: Ground
```

Ending with just the ground and no fruit is what we want. The moral of this story is that we need to be careful removing things from JavaScript lists.

Happily, the other thing we do inside the `reset()` function is easier. We set `gameOver` back to `false` and restart the `animate()` function. With that, our game is ready!

Improvements

Congratulations! You just wrote another game from scratch. And it's a fun and challenging one. You can still use other ways to improve things. Some suggestions:

- Add things that the Purple Fruit Monster doesn't like, to make the score go down when she eats them. *Hint:* An `/images/rotten_banana.png` image is also available!

- Stop the game if too much fruit gets past the Purple Fruit Monster. *Hint:* create a `checkMissedFruit()` function that's called at the very beginning of `launchFruit()`. The 'checkedFruit()' function should loop through all fruit in the scene, counting the number whose X position is too low. If that count is too high, it's game over!

This is *your* code, so make the game better as only you can!

The Code So Far

If you'd like to double-check the code in this chapter, turn to *Code: The Purple Fruit Monster Game*. That code includes the `checkMissedFruit()` improvement, but try it on your own first.

What’s Next

This was an impressive game to make. In the upcoming chapters, we’ll practice the physics skills that we developed here. We’ll also build on the concept of a `gameStep` function, which was fairly simple in this game. But before that, what high score can you get with Purple Fruit Monster?

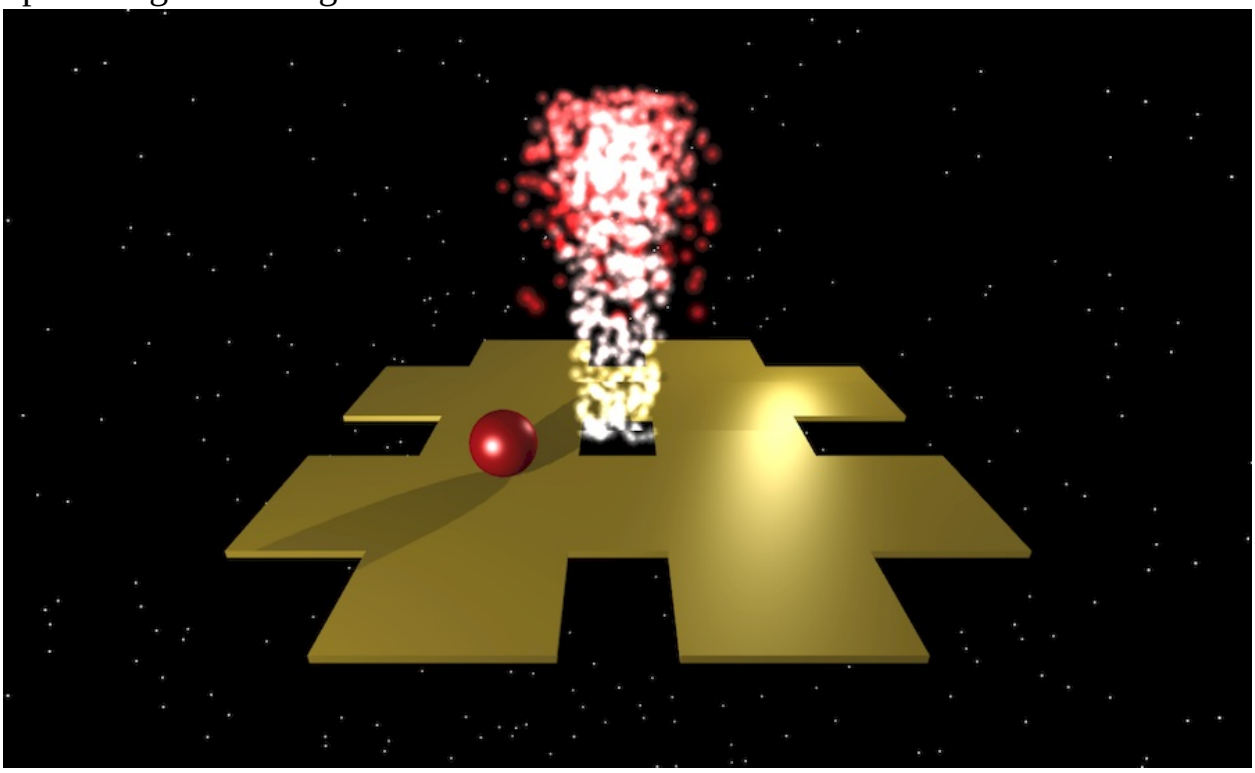
When you're done with this chapter, you will

- *Have built a complete 3D mini-game*
- *Understand how to build complex 3D game pieces*
- *Be able to create cool particle effects like fire*
- *Be able to combine shapes, materials, lights, and physics together in a game*

Chapter 15

Project: Tilt-a-Board

Let’s build a 3D game in which a ball lands on a small maze in space. The object of the game is to use the arrow keys to tilt the maze so that the ball falls through a small hole in the center of the board—*without* falling off the edge. It will end up looking something like this:



We’ll make this game pretty, so we’ll be using skills from Chapter 12, *Working with Lights and Materials*. We’ll need physics to make the ball fall, to make it slide back and forth on the game board, and to detect when it hits the goal, so we’ll use some of the skills from Chapter 14, *Project: The Purple Fruit Monster*

Game. And we'll be adding a lot of shapes and moving them around, so we'll need the skills from the first half of the book, as well.

A word to the wise: a ton of things are going on in this game, which means we'll be typing a lot of code. To save time, we won't talk much about ideas and approaches that we introduced in earlier chapters. If you haven't already worked through those earlier chapters, coding this game may be frustrating!

Getting Started

Start a new project in 3DE. Choose the **3D starter project (with Physics)** template and name this project **Tilt-a-Board**.

Gravity and Other Setup

To get “physics” working in the Purple Fruit Monster game, we needed to do a little work before the **START CODING ON THE NEXT LINE** line. Without enabling physics, shapes won’t fall, roll, bounce, slide, or do anything like real-world things do.

In this project, we’ve chosen a template that already includes the setup needed for the physics engine to work. Double-check that you see the physics code in the “Tilt-a-Board” project that you just created.

```
    <body></body>
    <script src="/three.js"></script>
①
    <script src="/physi.js"></script>
    <script>
      // Physics settings
②     Physijs.scripts.ammo = '/ammo.js';
③     Physijs.scripts.worker = '/physijs_worker.js';

      // The "scene" is where stuff in our game will happen:
④     var scene = new Physijs.Scene();
⑤     scene.setGravity(new THREE.Vector3( 0, -100, 0 ));
      var flat = {flatShading: true};
      var light = new THREE.AmbientLight('white', 0.8);
      scene.add(light);
```

① Load the physics library.

② Tell the physics library where it can find additional help to detect collisions.

③ Set up a **worker** to perform all of the physics calculations.

④

Create a physics-enabled `Physijs.scene`.

⑤ Enable gravity.

Lights, Camera, Shadows!

Most of the light in this game will come from point lights that can cast shadows. So let's dial back the ambient light brightness at the top of the code from `0.8` to `0.2`.

```
var light = new THREE.AmbientLight('white', 0.2);
```

To get the best view of this game, we'll want the camera a little above and back from the game board, but still looking down at the center of the scene. We also need to enable shadows in the renderer as we did in Chapter 12, *Working with Lights and Materials*. Add the following setup code above the `START CODING` line:

```
camera.position.set(0, 100, 200);
camera.lookAt(new THREE.Vector3(0, 0, 0));
renderer.shadowMap.enabled = true;
```

That's it for the setup. Let's get started with the code that goes after `START CODING ON THE NEXT LINE`.

Outline the Game

In addition to lights, this game will have a ball, a game board, and a goal. Let's start with the following code outline, including the double slashes:

```
//var lights = addLights();
//var ball = addBall();
//var board = addBoard();
//var goal = addGoal();
```

Just as we did in Chapter 14, *Project: The Purple Fruit Monster Game*, we'll uncomment these function calls as we define the functions.

Add Lights

Before doing anything else, let's add some lights to the scene.

Below the commented-out code outline, add the following function definition of `addLights`:

```
function addLights() {
  var lights = new THREE.Object3D();

  var light1 = new THREE.PointLight('white', 0.4);
  light1.position.set(50, 50, -100);
  light1.castShadow = true;
  lights.add(light1);

  var light2 = new THREE.PointLight('white', 0.5);
  light2.position.set(-50, 50, 175);
  light2.castShadow = true;
  lights.add(light2);

  scene.add(lights);
  return lights;
}
```

We're adding two point lights here. We saw in Chapter 12, *Working with Lights and Materials* that these are similar to light bulbs. We group both in a 3D object, `lights`, which is added to the scene and returned from the function. Both lights

will cast shadows.

Now that we've added the function definition, uncomment the call to `addLights` in the code outline.

```
» var lights = addLights();
  //var ball = addBall();
  //var board = addBoard();
  //var goal = addGoal();
```

Add the Game Ball

Add the `addBall` function below the function definition for `addLights`.

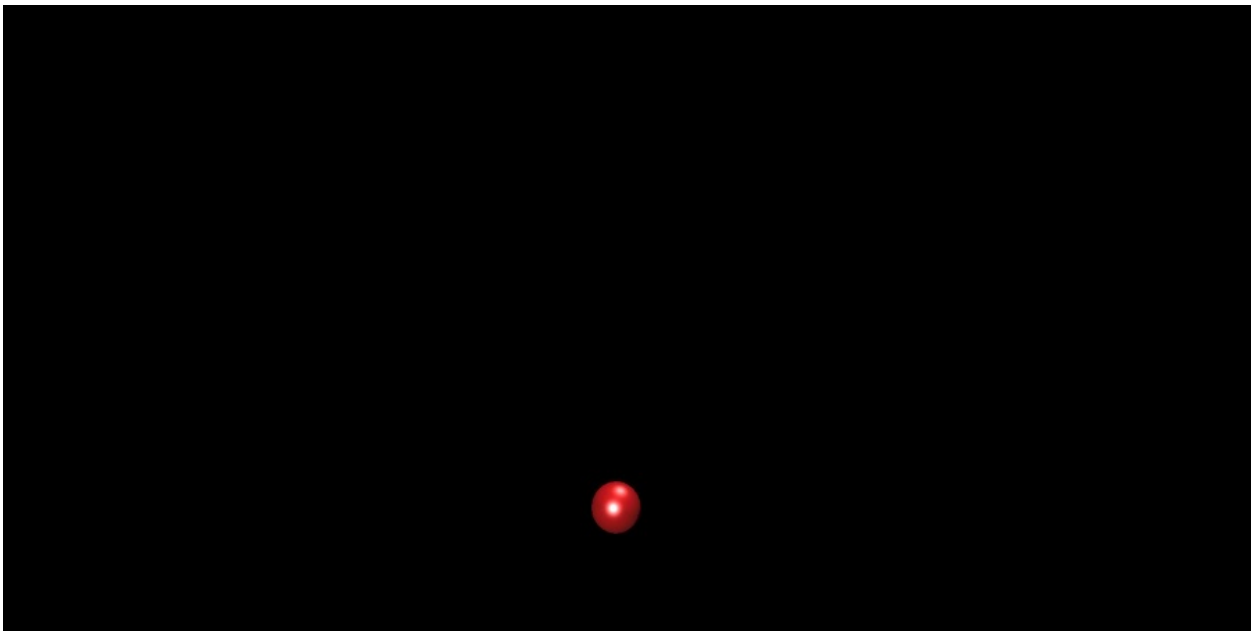
```
  function addBall() {
    var shape = new THREE.SphereGeometry(10, 25, 21);
    var cover = new THREE.MeshPhongMaterial({color: 'red'});
    cover.specular.setRGB(0.6, 0.6, 0.6);

    var ball = new Physijs.SphereMesh(shape, cover);
    ball.castShadow = true;

    scene.add(ball);
    return ball;
  }
```

This function adds a physics-enabled, red ball to the scene. We give it a little specular shininess and enable it to cast shadows.

After finishing the function, uncomment the call to `addBall()` in the code outline.

```
  var lights = addLights();
» var ball = addBall();
  //var board = addBoard();
  //var goal = addGoal();
```

Since the ball is physics-enabled, it falls right out of the scene never to be seen again as shown in the figure.



We'll fix that, starting with a game board to catch the ball.

Add the Game Board

Add the `addBoard` function after the `addBall()` function. (Warning: this one requires a lot of typing.)

```
function addBoard() {
  var cover = new THREE.MeshPhongMaterial({color: 'gold'});
  cover.specular.setRGB(0.9, 0.9, 0.9);

  var shape = new THREE.CubeGeometry(50, 2, 200);
  var beam1 = new Physijs.BoxMesh(shape, cover, 0);
  beam1.position.set(-37, 0, 0);
  beam1.receiveShadow = true;

  var beam2 = new Physijs.BoxMesh(shape, cover, 0);
  beam2.position.set(75, 0, 0);
  beam2.receiveShadow = true;
  beam1.add(beam2);

  shape = new THREE.CubeGeometry(200, 2, 50);
  var beam3 = new Physijs.BoxMesh(shape, cover, 0);
  beam3.position.set(40, 0, -40);
  beam3.receiveShadow = true;
  beam1.add(beam3);
```

```
    var beam4 = new Physijs.BoxMesh(shape, cover, 0);
    beam4.position.set(40, 0, 40);
    beam4.receiveShadow = true;
    beam1.add(beam4);

    beam1.rotation.set(0.1, 0, 0);
    scene.add(beam1);
    return beam1;
}
```

We create four beams and combine them all together to make the game board. At the very end, we tilt the board a bit (to get the ball rolling) and add it to the scene. Note that we mark each of the beams to make them able to have shadows on them.

One thing that's new is the `0` in each of the `BoxMesh`es.

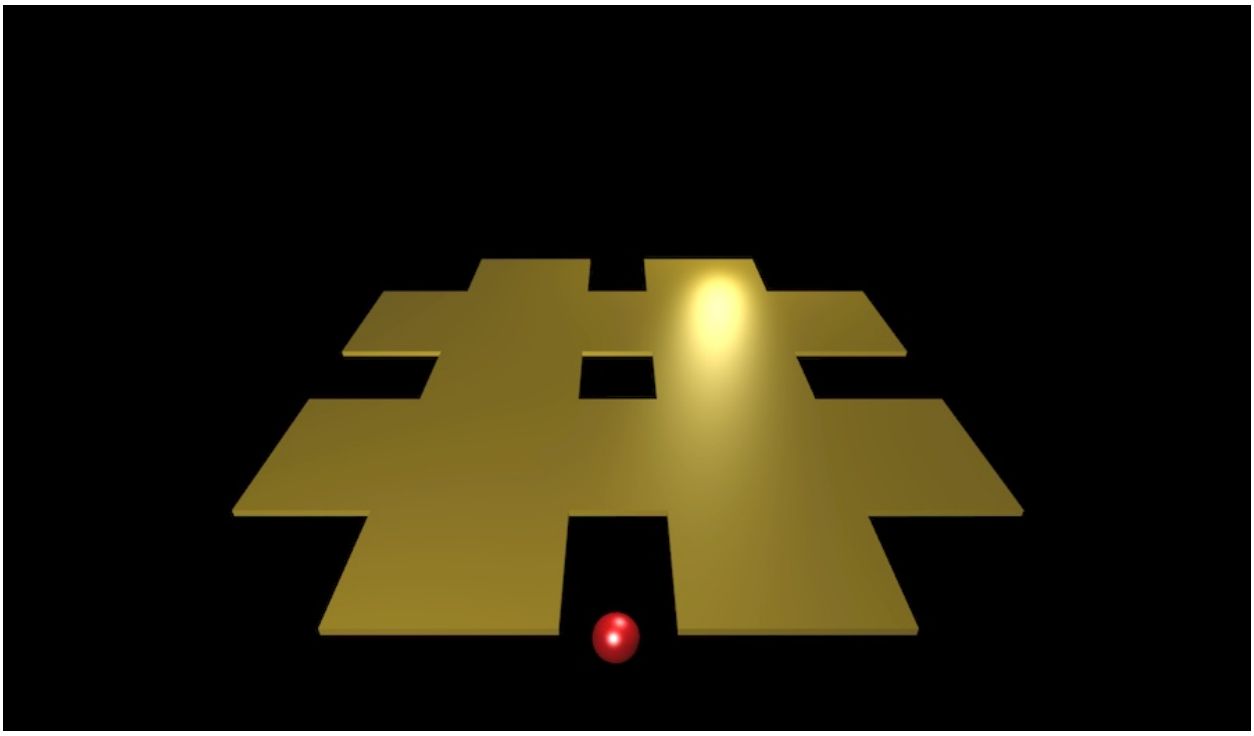
```
var beam1 = new Physijs.BoxMesh(shape, cover, 0);
```

As we saw in Chapter 14, *Project: The Purple Fruit Monster Game*, the `0` tells the physics library that the board does not move. Without the 0, our game board would fall right off the screen just like the ball does.

Uncomment the call to `addBoard` in the code outline.

```
var lights = addLights();
var ball = addBall();
» var board = addBoard();
//var goal = addGoal();
```

With that, the ball should fall right through the middle of the game board.



That’s not right since the goal of the game is get the ball to fall through the middle of the game board. So before we add the goal, let’s first reset the ball’s location.

Reset the Game

When the game starts—or restarts—the game board should have a slight tilt and the ball should fall down on the edge of the far left beam. Add the following `reset()` function below the `addBoard()` function:

```
function reset() {
  ball.__dirtyPosition = true;
  ball.__dirtyRotation = true;
  ball.position.set(-33, 200, -65);
  ball.setLinearVelocity(new THREE.Vector3(0, 0, 0));
  ball.setAngularVelocity(new THREE.Vector3(0, 0, 0));


  board.__dirtyRotation = true;
  board.rotation.set(0.1, 0, 0);
}
```

Don’t forget the two underscores before `dirtyPosition` and `dirtyRotation`!

We first used “dirty” positions back in *Resetting the Position*. We use both `__dirtyPosition` and `__dirtyRotation` here, so we can change the position and spin of the ball.

Add a call to the `reset()` function below the code outline.

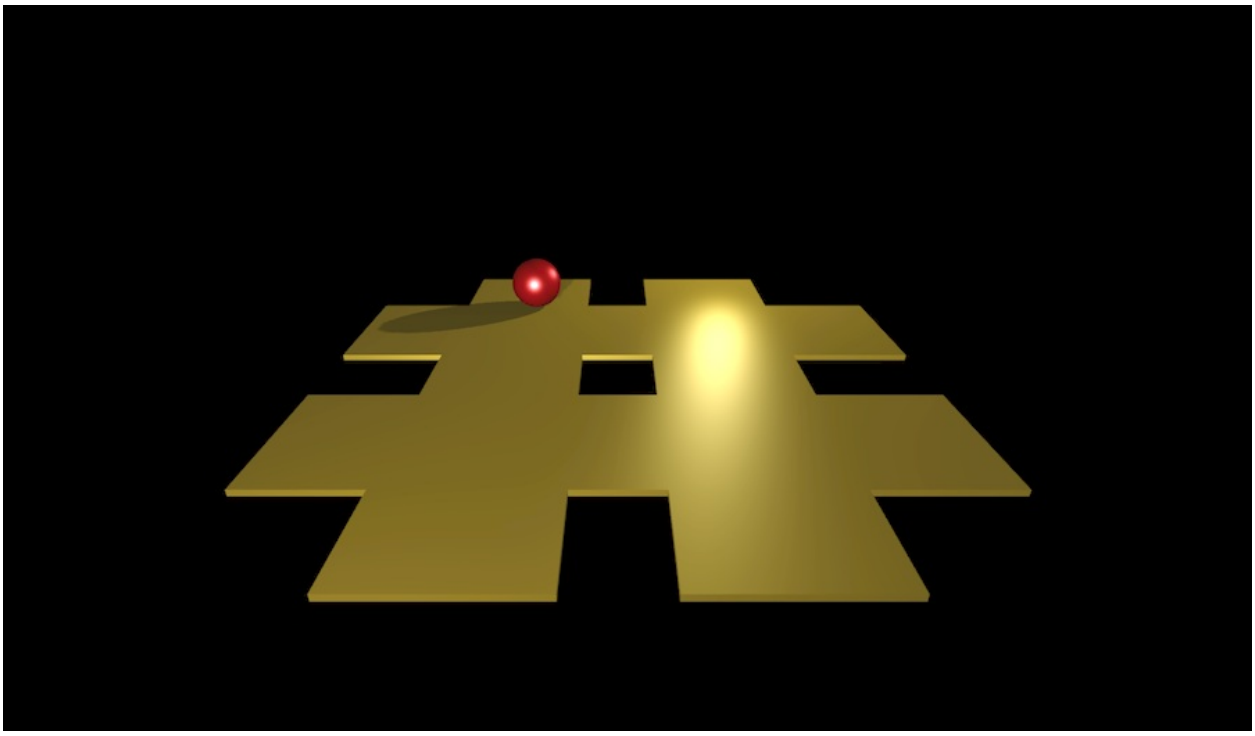
```
  var lights = addLights();
  var ball = addBall();
  var board = addBoard();
  //var goal = addGoal();

» reset();
```

Our code outline now adds lights, the game ball, and the game board. Then it resets everything to the start position. With that, the ball should start on the back of the game board instead of falling through the hole in the middle.

Add a call to `reset()` down in the `gameStep()` function.

```
  function gameStep() {
»   if (ball.position.y < -500) reset();

    // Update physics 60 times a second so that motion is smooth
    setTimeout(gameStep, 1000/60);
  }
```

That `if` statement checks to see if the ball has fallen down really far. If the ball’s Y position is below -500, then we reset the ball back to the start position so the player can try again as shown in the figure.



OK, we have our ball and game board and a way to start and restart the game. Before adding the goal, let's add keyboard controls for the game board.

Add Game Controls

We'll add the game controls at the very bottom of our code. Add the following "keydown" listener and `sendKeyDown()` function below the `gameStep()` function and the call to `gameStep()`:

```
document.addEventListener("keydown", sendKeyDown);

function sendKeyDown(event){
  var code = event.code;
  if (code == 'ArrowLeft') left();
  if (code == 'ArrowRight') right();

  if (code == 'ArrowUp') up();
  if (code == 'ArrowDown') down();
}
```

By now we're familiar with using JavaScript keyboard events to control

gameplay like this. Here, we're calling functions to tilt the game board left, right, up, and down. We'll add those function definitions next, after the `sendKeyDown()` function.

```
function left()  { tilt('z',  0.02); }
function right() { tilt('z', -0.02); }
function up()    { tilt('x', -0.02); }
function down()  { tilt('x',  0.02); }

function tilt(dir, amount) {
  board.__dirtyRotation = true;
  board.rotation[dir] = board.rotation[dir] + amount;
}
```

The `left`, `right`, `up`, and `down` functions are pretty easy to understand. They're so short that we can put the entire function definition on one line! What we're doing in the `tilt` function called by each of those is a little trickier.

We already know `__dirtyRotation` from the `reset()` function. We have to set it here because otherwise the board doesn't move. Remember when we added the 0 to the `BoxMesh` in `addBoard()`? That 0 says the board doesn't move or rotate…unless we set a dirty property.

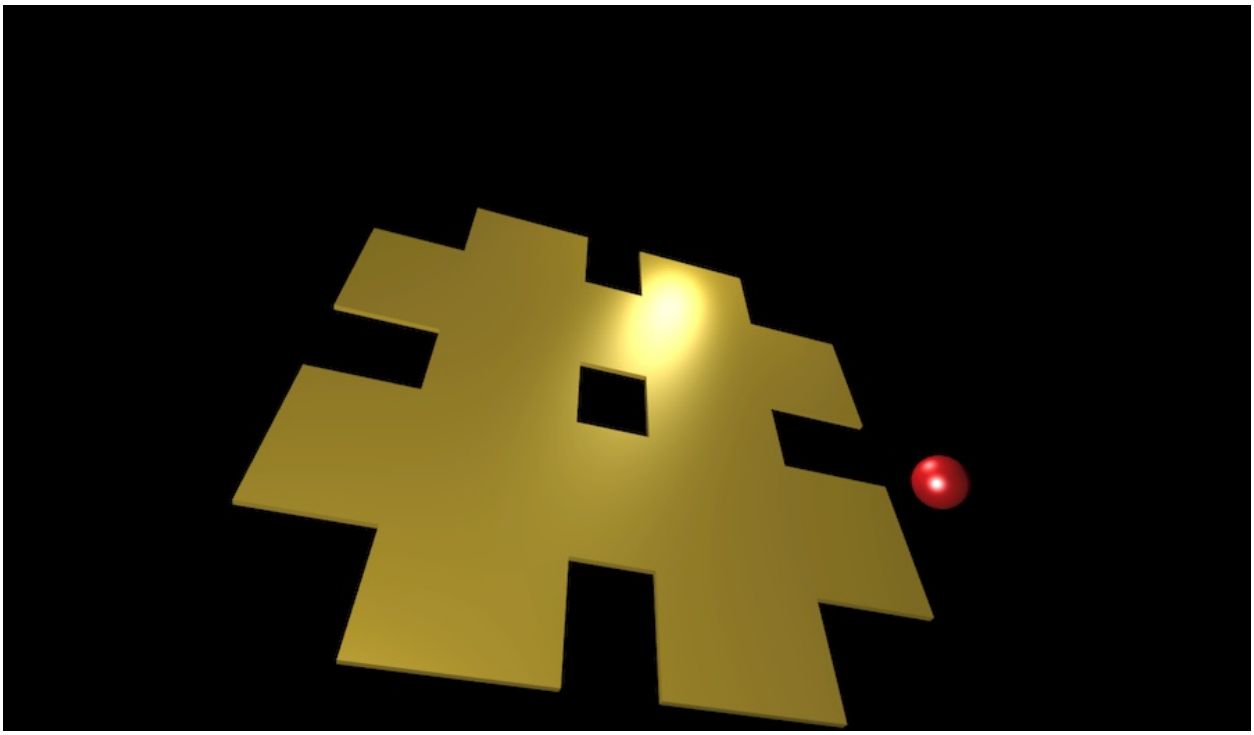
What's really sneaky in `tilt` is `board.rotation[dir]`. When the `left` function is called, it calls `tilt`, setting the `dir` argument to `'z'`. Since `dir` is `'z'`, setting `board.rotation[dir]` is the same thing as setting `board.rotation['z']`. This is something new! We've seen stuff like `board.rotation.z`, but we've never seen square brackets and a string like that.

Well, it turns out that `board.rotation['z']` is the same as `board.rotation.z`. JavaScript sees both as changing the `z` property of the rotation. Using this trick, we write just one line that can update all different directions in `tilt`.

```
board.rotation[dir] = board.rotation[dir] + amount;
```

Without a trick like that, we would probably have to use four different `if` statements. So we lazy programmers like this trick!

Give the game board a try! You should be able to tilt it left, right, up, and down using the arrow keys.



You might even be able to get the ball through the center goal, but since nothing is there yet, nothing happens. Let's add a goal next.

Add the Goal

To keep our code organized, we continue to define functions in the same order they're called. So place the `addGoal()` function below `addBoard()`, but above `reset()`.

```
function addGoal() {
  shape = new THREE.CubeGeometry(100, 2, 100);
  cover = new THREE.MeshNormalMaterial({wireframe: true});
  var goal = new Physijs.BoxMesh(shape, cover,  0);
  goal.position.y = -50;
  scene.add(goal);

  return goal;
}
```

This is just a small, flat box that we place below the game board. This won't do anything until we add a collision event listener.

Wireframing

You might have noticed that we set `wireframe` to `true` when we created the goal. A wireframe lets us see the geometry without a material to wrap it. It's a useful tool to explore shapes and to draw planes as we've done here.

Normally you should remove the `wireframe` property in finished game code (you can remove the enclosing curly braces, too). In this game, it probably makes the most sense to change `wireframe: true` to `visible: false` so that the goal is invisible to the player.

Before we get to the collision detection for winning the game, let's add another function to add a "goal light." This light will do two things: highlight the goal for the player and flash when the player wins the game.

Add the `addGoalLight()` function below the `reset()` function.

```
var goalLight1, goalLight2;
function addGoalLight(){
  var shape = new THREE.CylinderGeometry(20, 20, 1000);

  var cover =  new THREE.MeshPhongMaterial({
    emissive: 'white',
    opacity: 0.15,
    transparent: true,
    color: 'black'
  });
  goalLight1 = new THREE.Mesh(shape, cover);
  scene.add(goalLight1);

  var cover2 =  new THREE.MeshPhongMaterial({
    visible: false,
    emissive: 'red',
    opacity: 0.4,
    transparent: true,
```

```
      color: 'black'
    });
    goalLight2 = new THREE.Mesh(shape, cover2);

    scene.add(goalLight2);
  }
```

This function is a little strange. It doesn't add an actual light that shines light in our scene. Plus, it adds two fake lights, not just one "goal light."

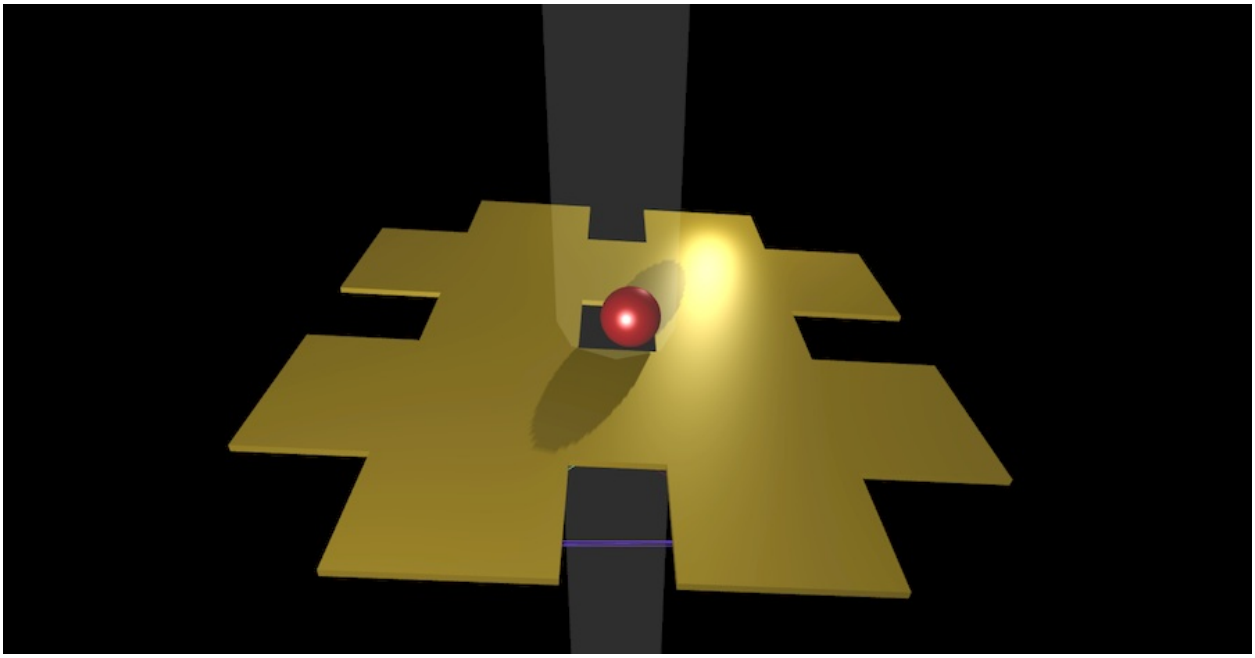
Remember that we're using the goal light to highlight the goal for the player. It's meant to look a little like a spotlight. To give it the look of a spotlight shining on something important, we mark it as transparent and give it a low opacity. Being *transparent* means that we can see through it. The *opacity* determines how easy it is to see through—a number close to 0, like `0.15`, is almost completely transparent.

For the second goal light, we reuse the same shape, but use a slightly different material. It's still transparent, but with an opacity of `0.4`, it is not quite as see-through as the first light. This light will emit a red color instead of the white of the first one. Most important, we make this material invisible with `visible: false`.

Only one of these goal lights will be visible at a time. To see that, let's add this function to our code outline:

```
  var lights = addLights();
  var ball = addBall();
  var board = addBoard();

  var goal = addGoal();

  reset();
» addGoalLight();
```

With that, you should see the mostly transparent, white goal light.



Usually this first goal light is the one that will be seen. When the player wins, we'll switch back and forth. First the red one will be visible and the white will be invisible. Then, half a second later, the white one will be visible and the red invisible. To make that happen, add the `win()` function below `addGoalLight()`:

```
function win(flashCount) {
  if (!flashCount) flashCount = 0;

  goalLight1.material.visible = !goalLight1.material.visible;
  goalLight2.material.visible = !goalLight1.material.visible;

  flashCount++;
  if (flashCount > 10) {
    reset();
    return;
  }

  setTimeout(win, 500, flashCount);
}
```

This is a fun little function! The first line checks to see if `win()` was called with an argument. If that `flashCount` is not set, then we set it to 0.

The next line changes the `visible` property on the first goal light's material. To do this, we use the Boolean "not" operator from *Booleans*. If the `visible` property is

`true`, then `visible` is updated to "not" `true`—or `false`. If the `visible` property is `false`, then `visible` is updated to `true`.

The next line takes whatever new value is there for goal light 1's `visible` property, and sets goal light 2's `visible` property to the opposite. So if goal light 1 is visible, then goal light 2 will be invisible. If goal light 1 is not visible, goal light 2 will be visible. Cool, right? But wait, the fun doesn't stop there!

Next we increase the flash count—the number of times the light has flashed—by one. If the number of flashes is more than 10, then we reset the game and return out of the `win()` function.

If the number of flashes is still less than 10, we skip this `if` statement and move to a `setTimeout()`. We tell `setTimeout()` to call this same `win()` function after a wait of half a second (500 milliseconds). We also do something a little different. We pass the value of `flashCount`, which we just increased by 1 a few lines earlier, to `win()` after waiting half a second.

In other words, we've written `win()` so that it keeps calling itself. Each time `win()` is called, it simulates a flashing light by switching which goal light is visible. Then it waits half a second to do it all over again, with a `flashCount` of one more than it started with. That's tricky to understand, but it's yet another demonstration of the power of functions.

We have one more thing to add before we're finished with the goal. After the `win()` function, add an event listener to the `goal`.

```
goal.addEventListener('collision', win);
```

This calls `win()` when the ball collides with the goal.

With that, we're done with the goal. You should be able to hide the code and use the arrow keys on the keyboard to tilt the board until the ball drops through the hole and onto the goal.

If the goal doesn't flash, check the JavaScript console. You can also try adding a

call to `win()` just below the code outline. That will call `win()` without needing to tilt the board. You can then see what's wrong in the JavaScript console and try fixing it in the `addGoalLight()` or `win()` functions. Just remember to remove the call to `win()` from the code outline when you're done debugging. We don't want to give players a win unless they earn it!

That's It!

You should have a fully functioning tilt-a-game working at this point. Use the arrow keys to tilt the board and score.

Bonus #1: Add a Background

We can give this mini-game a space-age feel with the starry background from Chapter 13, *Project: Phases of the Moon*. Below the `win` function definition and the collision event listener, add the following code:

```
function addBackground() {
  var cover = new THREE.PointsMaterial({color: 'white', size: 2});
  var shape = new THREE.Geometry();

  var distance = 500;
  for (var i = 0; i < 2000; i++) {
    var ra = 2 * Math.PI * Math.random();
    var dec = 2 * Math.PI * Math.random();

    var point = new THREE.Vector3();
    point.x = distance * Math.cos(dec) * Math.cos(ra);
    point.y = distance * Math.sin(dec);
    point.z = distance * Math.cos(dec) * Math.sin(ra);

    shape.vertices.push(point);
  }

  var stars = new THREE.Points(shape, cover);
  scene.add(stars);
}
```

Once you have that, you can add a call to the `addBackground` function in the code outline.

```
var lights = addLights();
var ball = addBall();
var board = addBoard();
var goal = addGoal();

reset();
addGoalLight();
» addBackground();
```

Building stars from a points mesh like this is pretty cool. This is just the beginning of what is possible with points. Let’s see something amazing when

we…

Bonus #2: Make Fire!

All modern computers are extremely complex systems. Nearly all of them have a part that's dedicated to processing graphics: a Graphics Processing Unit or GPU. Any time we programmers can get the GPU to do work is a win because that frees up the Central Processing Unit, or CPU, to perform more game logic.

One of the many things GPUs are good at are "shader particles"—special points that can blend into a scene. Regular points, like those we used in the star field, won't work for shader particles. We need a new kind of point. But first, we have to load a new code collection.

```
  <body></body>
  <script src="/three.js"></script>
  <script src="/physi.js"></script>
» <script src="/spe.js"></script>
```

Next, we completely replace the `addGoalLight()` function. Instead of two fake lights that we switch between, we will add a "shader particle emitter."

```
var fire, goalFire;
function addGoalLight(){
  var material = new THREE.TextureLoader().load('/textures/spe/star.png');
  fire = new SPE.Group({texture: {value: material}});
  goalFire = new SPE.Emitter({particleCount: 1000, maxAge: {value: 4}});
  fire.addEmitter(goalFire);

  scene.add(fire.mesh);

  goalFire.velocity.value = new THREE.Vector3(0, 75, 0);
  goalFire.velocity.spread = new THREE.Vector3(10, 7.5, 5);
  goalFire.acceleration.value = new THREE.Vector3(0, -15, 0);
  goalFire.position.spread = new THREE.Vector3(25, 0, 0);
  goalFire.size.value = 25;
  goalFire.size.spread = 10;
  goalFire.color.value = [new THREE.Color('white'), new THREE.Color('red')];
  goalFire.disable();
}
```

This shader particle starts off disabled. We will enable it shortly, but first here's a

quick description of what the various properties in this function do.

This function loads a “spark” image that will make up our fire. We group these particles into the `fire` value. Next we build the `goalFire` “emitter,” which will emit —or throw out—1000 fire particles, each of which will exist for four seconds before dying out. We add the emitter to the group of fire particles and add the group to the scene. Then we set a bunch of properties for the emitter particles including:

- their speed up in the Y direction
- how much the speed of each particle might vary
- how fast the particles get pulled back down
- how spread out the bottom of the fire is
- how big particles are
- how much the particle size can vary

The last property says to start particles as white and end as red.

Shader particles need to be updated regularly. It’s best to do this when animating the scene, so add the highlighted code to the `animate()` function.

```
var clock = new THREE.Clock();
function animate() {
  requestAnimationFrame(animate);
  renderer.render(scene, camera);

» var dt = clock.getDelta();
»
» fire.tick(dt);
  lights.rotation.y = lights.rotation.y + dt/2;
}
animate();
```

You can also animate the lights in the scene as shown on the last line, but that is up to you!

To make sure that everything’s typed correctly, move back to the `addGoalLight()` function. Change the last line of that function to `goalFire.enable()`. If everything is working, you should see a fire in the middle of the game board. Just make sure

to switch it back—we only want the goal fire enabled after a win.

Let's Play!	
	There are a lot of particle properties. The best way to understand them is to take some time to play with them. What happens if you change the `spread` properties to be all 0s? What happens if you change the `acceleration` values? What happens if you add another color to `color`? Can you make a better fire than this one?

After disabling the fire at the end of `addGoalLight()`, we need to enable it when the player wins the game. Change the `win()` function as shown:

```
  function win(flashCount) {
    if (!flashCount) flashCount = 0;

»   goalFire.enable();

    flashCount++;
    if (flashCount > 10) {
      reset();
»     goalFire.disable();
      return;
    }

    setTimeout(win, 500, flashCount);
  }
  goal.addEventListener('collision', win);
```

This removes the code that switches the visibility of the two fake goal lights. Instead, it enables the `goalFire` emitter when it's first called—when the ball first collides with the goal. Then, when the `win()` function has been called enough, we again disable the fire until the next time the player wins the game.

That is pretty cool. And maybe your fire is even better. Just remember that cool

looking games are no substitute for making games fun. Right now, the game is pretty easy to win. Can you make it harder?

Challenge

To make the game harder, try making the beam on the left longer. Try adding another beam in the back that runs from left to right. Finally, have the ball land on the edge of that new beam in the back.

You might also try adding a scoreboard like the one in Chapter 11, *Project: Fruit Hunt* that resets the game if the player can't finish in a certain amount of time.

Can you think of other ways to improve this mini-game? Get creative!

The Code So Far

If you want to double-check the code in this chapter, turn to *Code: Tilt-a-Board*.

What’s Next

That was our prettiest game yet. We combined our skills with writing 3D games with our new skills of making shadows and materials. The tilt-a-board game is fun to play and cool looking. It took a lot of time to code, but it was worth it.

In the next chapters, we’ll dig a little more into JavaScript. Specifically, we’ll cover objects, which we’ve been using all along but haven’t talked about making. Once we have that skill, we’ll build a couple more nifty games.

When you're done with this chapter, you will

- *Know what that* `new` *keyword we keep using means*
- *Be able to define your own objects*
- *Have seen the worst thing that JavaScript does*

Chapter 16

Learning about JavaScript Objects

We've made some pretty incredible progress so far. We have an avatar that can walk around the screen and bump into obstacles. We built an animated model of the moon's movements. We also tried out our new skills to create a couple of pretty cool games.

We've made so much progress, in fact, that we've reached the limit of what we can do with JavaScript—at least without introducing something new. To understand why we need to learn about this new concept, consider our avatar. We can make plenty of games where our avatar could play by itself, but what if the player wants to play with others?

If two avatars were on the screen at the same time, how would we add all those hands, feet, and bodies to the screen and not mix them up? How would we make each one move independently? How would we assign different colors and shapes to each avatar?

Things quickly get out of control if we try to accomplish all these things with what we know so far. So it's time to learn about objects and see what we can do with them.

This Is a Challenging Chapter
A lot of new concepts are in this chapter. You may find it best to skim through this chapter the first time and then come back to explore it in more depth later.

Getting Started

Create a new project in the 3DE Code Editor. For this, let's use the `Empty project` template—NOT the usual animation template!—and call it `JavaScript Objects`.

We won't create visualizations in this chapter. Instead we'll create objects in 3DE and look at them in the JavaScript console. So be sure to have the JavaScript console open.

Simple Objects

Anything that we can touch or talk about in the real or virtual world can be described in computer programming: an avatar, a car, a tree, a book, a movie. Anything. When programmers talk about things in the computer world, we call these things *objects*. Let's think about movies. I think we can all agree that *Star Wars* is the greatest movie of all time, right? Of course we can.

Well, let's describe *Star Wars* as a JavaScript object. Add the following below "Your code goes here…":

```
var bestMovie = {
  title: 'Star Wars',
  year: 1977,
};
```

Hey, wait. Isn't that just a map, which we first met all the way back in Chapter 7, *A Closer Look at JavaScript Fundamentals*? The answer is, Yes! But it can be more that just a map.

	Put a comma after the last entry in maps and map-like things.
	In maps, it's helpful to place a comma after the last entry. This is not required and it does look a little weird, but we do this to make it easier to add another entry. Without a comma, it's easy to break the map when adding another item. It is easy to just start typing the next entry without remembering to first put a comma after the previous entry. If we already have that comma, then we don't have to worry about forgetting it!

In JavaScript, another name for a plain old map is an *object literal*. There's no difference between the two, so most of the time we just call them maps. We call

them object literals when we feel fancy. Or when we're about to turn them into "real" objects.

The difference between an object literal and a real object is the values. Right now, the `title` and `year` keys point to simple values—a string and a number. Even if we use a list or anything else we talked about in Chapter 7, *A Closer Look at JavaScript Fundamentals*, we still have an object literal.

```
var bestMovie = {
  title: 'Star Wars',
  year: 1977,
  stars: ['Mark Hamill', 'Harrison Ford', 'Carrie Fisher'],
};
```

Things get *real* when we point the keys to… functions!

```
var bestMovie = {
  title: 'Star Wars',
  year: 1977,
  stars: ['Mark Hamill', 'Harrison Ford', 'Carrie Fisher'],
  logMe: function() {
    console.log(this.title + ', starring: ' + this.stars);
  },
};
```

That's wild. The `logMe` key is pointing to a function—without a name. And inside the function are two references to `this`. What's going on here?

The best way to answer that is to call that `logMe` function. Believe it or not, we already know how to call that function. We've been calling similar functions since Chapter 1, when we added things to a scene with `scene.add(ball)` or set the positions with `position.set(-250,250,-250)`.

For the `logMe()` function in `bestMovie`, we call `bestMovie.logMe()`. Add that call below the closing curly brace of `bestMovie`.

```
bestMovie.logMe();
```

In the JavaScript console, you should see the message: "Star Wars, starring:

Mark Hamill, Harrison Ford, Carrie Fisher.”

That’s kind of neat, right? For that to work, JavaScript does two things to our object function:

1. It lets us call the key like it’s a function. The key is `logMe`, but we can call it as `logMe()`.
2. It does some pretty special work with the `this` variable inside the function, because we didn’t create it.

The `this` variable turns out to be one of *the* most powerful things in JavaScript. It has a number of uses, but we’re going to use it to refer to the current object. Anytime we create an object, JavaScript automatically defines `this` to mean the current object, which is why `this.title` and `this.stars` work in `logMe()`.

	The `this` Only Works Inside Functions.
	The `this` variable is special. JavaScript only lets us use it inside a function. If you try to use it elsewhere, you’ll get an error.

In addition to accessing simple values with `this`, we can even access other functions. Let’s change `logMe()` so that it’s not building a string. Instead, it will get that string from another function, `about()`.

```
  var bestMovie = {
    title: 'Star Wars',
    year: 1977,
    stars: ['Mark Hamill', 'Harrison Ford', 'Carrie Fisher'],
    logMe: function() {
»     var me = this.about();
»     console.log(me);
    },
```

```
»   about: function() {
»     return this.title + ', starring: ' + this.stars;
»   },
};
bestMovie.logMe();
```

With that, the `about()` function is building a string from the keys in `bestMovie` and the `logMe()` function is calling `about()` to get that string. Accessing both keys and methods works through `this`.

Properties and Methods

One of the annoying things about being a programmer is keeping the names of things straight: variables, strings, keys, lists, functions. It can get overwhelming—especially when you learn that programmers have two names for the same thing like maps and object literals.

Well, we're going to do it again!

When talking about objects, we don't refer to keys and functions. Instead, we call them *properties* and *methods*.

It might seem unimportant what names we use for these, but it matters for three reasons:

1. These names better describe how we use them with objects.
2. Other programming languages use these names for their objects, so it's good to be familiar with them.
3. That's what we're going to call them from now on!

If we access `title`, `year`, or `stars` on `bestMovie`, it's kind of like asking about different properties or characteristics of the best movie ever made. So `bestMovie.year` is one property of the best movie. And `bestMovie.stars` is another property of the same thing.

The name *method* describes how an object goes about doing something. It's the manner—or *method*—in which it performs some kind of action. So `bestMovie.logMe()` is the method in which `bestMovie` sends its information to the JavaScript console. And `bestMovie.about()` is how `bestMovie` combines its most important information to describe itself.

Hopefully the reason for the names makes sense, because we're going to use them in the very next section…

Copying Objects

In real life, you copy a cool idea or an interesting device by duplicating everything it does and changing a few things here and there to make it even better. The thing you're copying becomes the *prototype* for the new way of doing it. JavaScript handles copying objects in a similar way.

To describe another movie, we can copy the *prototypical* `bestMovie` object with `Object.create()`.

```
var greatMovie = Object.create(bestMovie);
greatMovie.logMe();
// => Star Wars, starring: Mark Hamill,Harrison Ford,Carrie Fisher
```

`Object.create()` will create a new object with all the same properties and methods as the prototypical object we created earlier. So the new object, `greatMovie`, has the same title and actors as the original `bestMovie`. It also has the same `logMe` and `about` methods.

Let's make this new object refer to a movie that's a favorite of all 3D programmers like us, *Toy Story*.

```
greatMovie.title = 'Toy Story';
greatMovie.year = 1995;
greatMovie.stars = ['Tom Hanks', 'Tim Allen'];
```

This changes the `title`, `year`, and `stars` properties of the new `greatMovie` object. What makes programming with objects so nice is what happens when we call the `logMe()` method on `greatMovie`.

```
greatMovie.logMe();
// => Toy Story, starring: Tom Hanks,Tim Allen
```

In the JavaScript console, we see information being logged about Toy Story.

What happens when we log `bestMovie`'s information?

```
bestMovie.logMe();
// => Star Wars, starring: Mark Hamill,Harrison Ford,Carrie Fisher
```

In the JavaScript console, we see information about Star Wars.

The new `greatMovie` and the prototypical `bestMovie` from which it was copied are different objects. They now have completely different properties. But they still use the same `logMe()` and `about()` methods—we didn't change them in `greatMovie`. And even though they use the same methods, they produce different results.

These methods produce different results because we used `this.title` and `this.stars` in the `about()` method.

```
about: function() {
  return this.title + ', starring: ' + this.stars;
},
```

Thanks to `this`, the `about()` method always uses the title and list of stars from the current object.

Note that updating properties on the new `greatMovie` object doesn't affect the `bestMovie` object. `bestMovie` has all of its properties unchanged and its `logMe` method still displays the original results.

	Let's Play!
	Play with your own objects. Create a `favoriteMovie` of your own. Can you create a `logFullTitle()` method that displays the title followed by the year in parentheses? It should look something like "Star Wars (1977)" in the JavaScript console. Make sure it works for the prototype object as well as your `favoriteMovie`.

All this talk of prototypes and prototypical objects is not just an excuse to throw fancy words around. In fact, the concept of a prototype is very important in

JavaScript and it helps to answer a question you may have had since the very first chapter in this book: what's that `new` keyword that we keep typing?

Constructing New Objects

We now have a good idea of what an object is in JavaScript. We also now see how an object can be a prototypical object and act as a template for creating similar objects. Creating new objects like this can be pretty tedious and mistake-prone. Consider this: if we forget to assign the `year` property on `greatMovie`, then the object will think *Toy Story* was made back in 1977. Unless we tell the object differently, it copies all properties from the original (`bestMovie`) object, including the year, 1977!

Another way to create objects in JavaScript is using simple functions. Yes, the same simple functions that we first saw all the way back in Chapter 5, *Functions: Use and Use Again*. Surprisingly, we don't have to do anything special to a function to create new objects.

Because we think it looks better, programmers usually capitalize the name of a function if it creates new objects. For example, a function that creates movie objects might be called `Movie`.

```
function Movie(title, stars) {
  this.title = title;
  this.stars = stars;
  this.year = (new Date()).getFullYear();
}
```

This is just a normal function using the `function` keyword, a name (`Movie`), and a list of arguments (such as the movie title and the list of stars in the movie).

However, we do something different inside this function definition than in typical functions. Instead of performing calculations or changing values, we assign the current object's properties. In this case, we assign the current object's title to `this.title`, the names of the actors and actresses who starred in the movie to `this.stars`, and even the year the movie was made to `this.year`.

Aside from assigning the `this` values, nothing is really special about this function.

So how does it create objects? What makes it an object creator and not a regular function?

The answer is that thing we saw in the very first chapter of this book: the `new` keyword. We don't call `Movie` the way we would a regular function. It's an object *constructor* (yes, programmers really love fancy names). So we construct new objects with it by placing `new` before the constructor's name.

```
var kungFuMovie = new Movie('Kung Fu Panda', ['Jack Black', 'Angelina Jolie']);
```

The `Movie` in `new Movie` is the constructor function we defined. It needs two arguments: the title (*Kung Fu Panda*), and a list of stars (Jack Black and Angelina Jolie).

Then, thanks to the property assignments we made in the constructor function, we can access these properties just like we did with our previous objects.

```
console.log(kungFuMovie.title);
// => Kung Fu Panda
console.log(kungFuMovie.stars);
// => ['Jack Black', 'Angelina Jolie']
console.log(kungFuMovie.year);
// => 2018
```

You might notice that the year of the Kung Fu Panda movie is wrong—it came out in 2008, not 2018. This is because our constructor only knows to set the `year` property to the current year.

Let's Play!

If you're up for a challenge, change the constructor so that it takes a third argument—the year. *If* the year is given as the third argument, then use that for `this.year`—*else* assign `this.year` to the current year as we are doing above.

Now we know how the creators of our 3D JavaScript code collection write all of their code, so that we can write things like:

```
var shape = new THREE.SphereGeometry(100);
var cover = new THREE.MeshNormalMaterial();
var ball = new THREE.Mesh(shape, cover);
```

`SphereGeometry`, `MeshNormalMaterial`, and `Mesh` are all constructor functions in the Three.js code collection.

One mystery is solved, but one remains: if we're using function constructors to build objects, how can we make methods for those objects? How could we define a `logMe()` method for `Movie` objects?

The answer to that is why we emphasized the word "prototype" in the previous section. To create a `logMe` method for the objects created with our `Movie` constructor, we define the method on the constructor's prototype. That is, for a prototypical movie, we want the `logMe` method to look like the following:

```
Movie.prototype.logMe = function() {
  console.log(this.title + ', starring: ' + this.stars);
};
```

With that method in place, we can ask the `kungFuMovie` to log itself.

```
kungFuMovie.logMe();
// => Kung Fu Panda, starring: Jack Black,Angelina Jolie
```

JavaScript objects can have any number of methods, like `logMe`, but it's good to keep the number of them small. If you find yourself writing more than 12 or so methods, then it may be time for a second object with a new constructor.

The Worst Thing in JavaScript: Losing `this`

JavaScript is a great programming language. Except when it's not. What happens to `this` in certain cases is a good example of JavaScript being not so great.

Consider using `setTimeout()` to call a function after a delay. We've done this several times in the book. In Chapter 11, *Project: Fruit Hunt*, we used `setTimeout()` to shake the treasure tree after a two-second delay.

```
setTimeout(shakeTreasureTree, 2*1000);
```

After 2 seconds (2000 milliseconds), `setTimeout()` called our `shakeTreasureTree()` function. We've also done this several times for the `gameStep()` function.

```
setTimeout(gameStep, 1000/30);
```

What happens if we try to do this with our `logMe()` method?

```
setTimeout(kungFuMovie.logMe, 500);
```

Try this out and see what happens.

What we expect is, after a delay of half a second, we can look in the JavaScript console to see the "about" message: "Kung Fu Panda, starring: Jack Black,Angelina Jolie."

What we actually see, however, is the message: "undefined, starring: undefined." Hunh?

What happens if we try this with `bestMovie`?

```
setTimeout(bestMovie.logMe, 500);
```

After that, we see an error in the JavaScript console similar to: "this.about is not a function."

What's going on here?

As crazy as this is going to sound, JavaScript has forgotten what `this` is! When we have `setTimeout()` call `logMe()`, JavaScript treats it like a plain old function instead of a method that knows about `kungFuMovie` or `bestMovie`.

When it tries to call the `logMe()` method for `kungFuMovie`, JavaScript knows that the method is the one that we defined on `Movie`'s prototype.

```
Movie.prototype.logMe = function() {
  console.log(this.title + ', starring: ' + this.stars);
};
```

But `this` is no longer assigned to `kungFuMovie`. So, when `setTimeout()` runs it, `this.title` and `this.stars` are both undefined. This is why we get the message, "undefined, starring: undefined."

When `setTimeout()` tries to call `logMe()` on `bestMovie`, JavaScript has also forgotten what `this` is. JavaScript doesn't even remember that we added an `about()` method to `bestMovie()`, so we get an error message that "this.about is not a function."

The problem of JavaScript forgetting about `this` is one of the hardest parts of JavaScript programming. People that have been programming JavaScript for *years* still get confused by this. Part of the problem is that JavaScript is just weird—this behavior doesn't seem to make much sense. This is also a hard problem because the error messages aren't helpful—something might be `undefined` for lots of reasons, and it's not always obvious that JavaScript is just being strange.

Happily, once we know what the problem is, it's an easy fix. We remind JavaScript what `this` should be with the `bind()` method.

```
setTimeout(kungFuMovie.logMe.bind(kungFuMovie), 500);
setTimeout(bestMovie.logMe.bind(bestMovie), 500);
```

Yes, it is super weird to write `kungFuMovie` or `bestMovie` twice to call a method. But this is really the worst thing about programming with JavaScript, and it's not

that bad. Plus we only have to do this when working with something like `setTimeout()`, or as we'll see in the next chapter, with collision methods.

	Let's Play!
	What happens if you `bind()` `bestMovie` to `kungFuMovie.logMe`? What happens if you `bind(kungFuMovie)` to `bestMovie.logMe`?

Challenge

This was a tough chapter. If you'd like a challenge to make sure you understand how to add methods to prototypes, give this a try.

Can you define an `about()` function on the `Movie.prototype` that returns the same information as `about()` from `bestMovie`? Then have `logMe()` log the results from `this.about()`.

If you can get that working, you're well on your way to working with objects!

The Code So Far

To double-check the code in this chapter, go to *Code: Learning about JavaScript Objects*.

What’s Next

Programming with objects is a tough thing to wrap your brain around. If you understood everything in this chapter, then you’re doing way better than I did when I first tried to learn it. If not everything made sense, don’t worry. Examples we’ll play with in the next few games should help to make things a little clearer.

After you’ve written a game or two with objects, it might help to reread this chapter. As the games you invent on your own get more and more complex, you’ll want to rely on objects to help organize your code.

When you're done with this chapter, you will

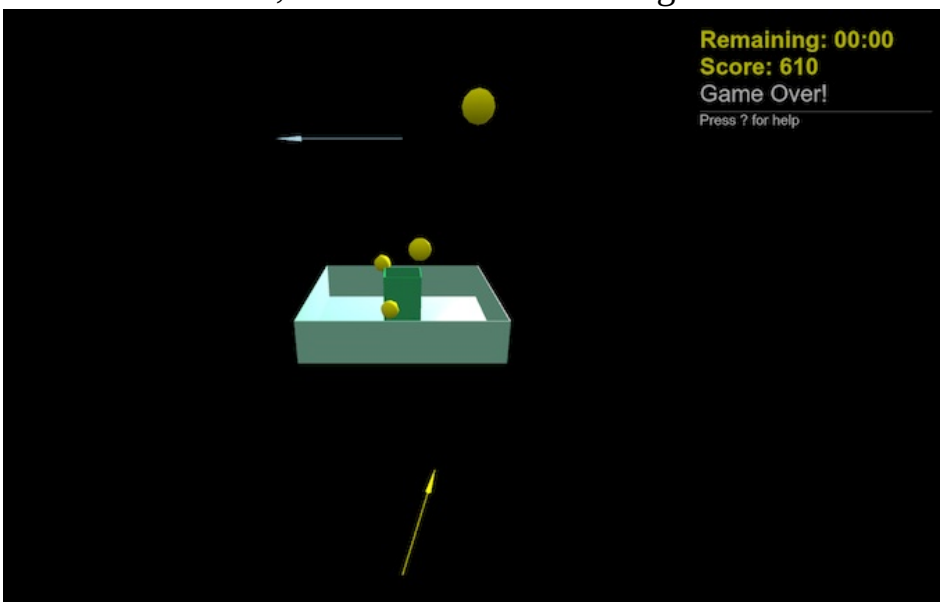
- *Have a fun game of skill and chance*
- *Better understand how to use JavaScript objects to build games*
- *Have even more experience with cool graphics and physics*

Chapter 17

Project: Ready, Steady, Launch

In this chapter, we build a game that allows us to exercise nearly all of our new-found skills. We'll use collisions, physics, lighting, materials, and even JavaScript objects.

We'll use them to create a nifty mini-game. The best games combine luck and skill to challenge players. "Ready, Steady, Launch" is just that kind of game. When we're done, it should look something like this:



JavaScript Objects Are... Weird

I said it in Chapter 16, *Learning about JavaScript Objects*, and I'll say it again here. JavaScript objects can get weird. They even trip up experienced JavaScript programmers, so don't get down if you don't quite "get" parts of this. Work through the chapter and you'll understand enough. Then maybe go back and reread Chapter 16, *Learning about JavaScript Objects*.

Following are the three important parts to this game:

1. The launcher
2. Baskets
3. Wind

The launcher will throw balls into the air in an attempt to get them in the baskets to score points. The wind will keep changing during the game, making it harder for players to use the launcher.

We will make each of these important parts a JavaScript object:

1. The launcher object will know how to move left or right and how to launch a ball.
2. The basket object will wait for balls to collide and score points.
3. The wind object will change direction and speed over time.

And all three will know how to draw themselves in the scene.

So let's get to it!

Getting Started

We begin by creating a new project in the 3DE Code Editor. Let's use the **3D starter project (with Physics)** template (you need to change the template this time) and call it **Ready, Steady, Launch**.

As you might guess, this template includes much of the physics-engine work we manually added in Chapter 14, *Project: The Purple Fruit Monster Game*.

We still need to make a couple of changes before the **START CODING** line. First, we'll want to keep score in this mini-game, so we need to include the **scoreboard.js** code collection. Start a new line after line 3, just before the plain **<script>** tag, and add the following **<script>** tag:

```
    <script src="/scoreboard.js"></script>
```

Next, we need to adjust the camera so that it's a little higher and looking down on the center of the scene. To do so, add the highlighted code just before the camera is added to the scene.

```
    camera.position.z = 500;
»   camera.position.y = 200;
»   camera.lookAt(new THREE.Vector3(0,0,0));
    scene.add(camera);
```

Since we haven't added anything to the scene yet, this won't make a noticeable difference. But it will definitely help players see the game better.

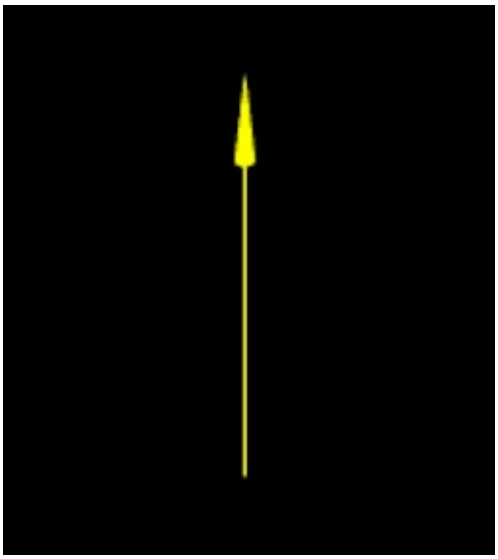
Let's start by adding the launcher to the game.

The Launcher

The launcher will be an arrow that points in the direction we want to launch the ball. Add the launcher arrow code just below the line with **START CODING ON THE NEXT LINE**.

```
var direction = new THREE.Vector3(0, 1, 0);
var position = new THREE.Vector3(0, -100, 250);
var length = 100;
this.arrow = new THREE.ArrowHelper(
  direction,
  position,
  length,
  'yellow'
);
scene.add(this.arrow);
```

We use an “arrow helper” to draw our launcher. We might replace it with an animated slingshot or something similar later, but an arrow will be just fine for our mini-game. The arrow needs a direction to point in (straight up), a position (a little down and a little back from center), a length, and a color. With those values, we create a new arrow and add it to the scene.



The launcher is going to do a lot: draw this arrow, move left and right, pull back to power up, and launch balls. Lots of features like that often lead to messy code.

So we're going to build a JavaScript object named `Launcher` to control all of that.

Add the the `Launcher` constructor *above* the arrow code that we just added.

```
function Launcher() {
  this.angle = 0;
  this.power = 0;
  this.draw();
}
```

A `Launcher` will have two properties: an angle that describes where it's pointing and the amount of power it uses for a launch. When we construct the launcher object, it assigns these properties and draws itself on the screen.

The `draw()` method should add the arrow to the screen. So let's convert our arrow code to a `draw()` method by adding the highlighted lines above and below the arrow code.

```
» Launcher.prototype.draw = function() {
    var direction = new THREE.Vector3(0, 1, 0);
    var position = new THREE.Vector3(0, -100, 250);
    var length = 100;
    this.arrow = new THREE.ArrowHelper(
      direction,
      position,
      length,
      'yellow'
    );
    scene.add(this.arrow);
» };
```

It's a good idea to indent the arrow code as shown to make it easier to read.

The arrow disappears once you add that. The arrow code in the `draw()` method is called whenever a new launcher object is created, so let's create one.

```
var launcher = new Launcher();
```

With that, the arrow should be back. If not, check the JavaScript console as described in Chapter 2, *Debugging: Fixing Code When Things Go Wrong*.

We're making small changes like we did in Chapter 13, *Project: Phases of the Moon*—for the same reasons that we did there: to make it easier to notice where we make mistakes.

A launcher needs to do more than just draw itself, so we'll keep adding to the launcher prototype. To do so, add a few blank lines above the `new Launcher` line.

The `Launcher` needs a method that converts its angle to a vector. Both the arrow and the launched balls need this. Add the `vector()` method below the `draw()` method.

```
Launcher.prototype.vector = function() {
  return new THREE.Vector3(
    Math.sin(this.angle),
    Math.cos(this.angle),
    0
  );
};
```

Then add a `moveLeft()` method below `vector()`.

```
Launcher.prototype.moveLeft = function(){
  this.angle = this.angle - Math.PI / 100;
  this.arrow.setDirection(this.vector());
};
```

This changes the launcher's direction slightly to the left and adjusts the arrow on the screen. Before we add the `moveRight()` method, let's test out `moveLeft()` to make sure we haven't missed anything.

All the way at the bottom of our code—below the `gameStep()` function and above the closing `</script>` tag, add an event listener for key presses.

```
document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;
  if (code == 'ArrowLeft') launcher.moveLeft();
}
```

Hide the code and try this out. You should be able to press the left arrow key to

see the launcher's arrow rotate to the left. If not, you know the drill: check the JavaScript console! Once the arrow moves when the left arrow is pressed, show the code again.

Since we're already looking at the `keydown` event listener, let's go ahead and add an `if` statement to handle a right arrow key.

```
document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;
  if (code == 'ArrowLeft') launcher.moveLeft();
» if (code == 'ArrowRight') launcher.moveRight();
}
```

Now move back up to the `Launcher` prototype code. Below the `moveLeft()` method, add `moveRight()`.

```
Launcher.prototype.moveRight = function(){
  this.angle = this.angle + Math.PI / 100;
  this.arrow.setDirection(this.vector());
};
```

Make sure that both move right and move left work after you made that change. Now it's time for the fun stuff: powering up the launcher and launching the balls.

The `powerUp()` method adds 5 to the current power (to a maximum of 100). Then it adjusts the arrow length to show how much power will be used.

```
Launcher.prototype.powerUp = function(){
  if (this.power >= 100) return;
  this.power = this.power + 5;
  this.arrow.setLength(this.power);
};
```

Then we launch!

```
Launcher.prototype.launch = function(){
  var shape = new THREE.SphereGeometry(10);
  var material = new THREE.MeshPhongMaterial({color: 'yellow'});
  var ball = new Physijs.SphereMesh(shape, material, 1);
  ball.name = 'Game Ball';
```

```
    ball.position.set(0,0,300);
    scene.add(ball);

    var speedVector = new THREE.Vector3(
      2.5 * this.power * this.vector().x,
      2.5 * this.power * this.vector().y,
     -80
    );
    ball.setLinearVelocity(speedVector);

    this.power = 0;
    this.arrow.setLength(100);
  };
```

The first half of that method creates a game ball with a weight of `1`. It adds the ball to the scene near the launcher arrow. Then it calculates a speed vector and uses that to set the ball's velocity—its speed and direction. Finally it resets the power and the arrow to their original values.

To use the launcher, we need to move back to the event listener code at the bottom of everything. When a player presses the down arrow key (and holds it down), we want to power up.

```
document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;
  if (code == 'ArrowLeft') launcher.moveLeft();
  if (code == 'ArrowRight') launcher.moveRight();
» if (code == 'ArrowDown') launcher.powerUp();
}
```

Next, add a new event listener that will listen for a key going up (being let go).

```
document.addEventListener('keyup', sendKeyUp);
function sendKeyUp(event){
  var code = event.code;
  if (code == 'ArrowDown') launcher.launch();
}
```

With that, when the player lets go of the down arrow key, the launcher should launch a ball. Hide the code and try it out!

Scoreboard

We'll want to keep score in this mini-game, so let's add a scoreboard. Below the line that creates the `new Launcher()`, add the usual scoreboard code:

```
var scoreboard = new Scoreboard();
scoreboard.countdown(60);
scoreboard.score(0);
scoreboard.help(
  'Use right and left arrow keys to point the launcher. ' +
  'Press and hold the down arrow key to power up the launcher. ' +
  'Let go of the down arrow key to launch. ' +
  'Watch out for the wind!!!'
);
scoreboard.onTimeExpired(timeExpired);
function timeExpired() {
  scoreboard.message("Game Over!");
}
```

You can skip the help text for now if you're eager to get back to coding. Just don't forget to add it later—it's no fun playing a game where you don't know the rules.

Baskets and Goals

We can launch game balls, but there's no way to score yet. In this section, we'll build a `Basket` object that will add to the score when struck by a launched ball. We'll start with a little more code in `Basket` than we did with `Launcher`, but try to keep the steps smallish.

We'll add the `Basket` prototype code below the `Launcher` prototype code. So add a few spaces below the closing curly brace of `Launcher.prototype.launch`, then add the `Basket` constructor.

```
function Basket(size, points) {
  this.size = size;
  this.points = points;
  this.height = 100/Math.log10(size);

  var r = Math.random;
  this.color = new THREE.Color(r(), r(), r());

  this.draw();
}
```

This constructor takes two arguments: the size of the basket and the number of points it's worth. Inside the constructor, we remember those two values as `this.size` and `this.points`. We also calculate the height. Next we assign a random color to our basket. Lastly, the `Basket` constructor calls its own `draw()` method.

We're having a little math fun with the height of the basket. We want small baskets to rise above wider baskets. Narrow baskets should be tall. Wide baskets should be shorter. Division almost works for this. If we said the height of a basket is 100 divided by the size, we'd get the following heights:

- For size 10 (small), the height would be 100÷10 = 10
- For size 100 (wide), the height would be 100÷100 = 1

We don't want *that* big of a difference between the heights. Narrow baskets should rise—not tower—above wider baskets. Also, a height of 1 is too small.

Logarithms are good at smaller changes. Specifically, we use a *base 10* logarithm—`Math.log10()`. A base 10 logarithm returns numbers like this:

- `Math.log10(10)` is 1
- `Math.log10(40)` is 1.602
- `Math.log10(100)` is 2
- `Math.log10(10000)` is 4

For multiples of 10, it returns the number of 0s after the one. For other numbers, the base 10 logarithm is somewhere in between.

This is why we set the height to 100 divided by the logarithm of the height. Instead of heights 10 and 1, we get:

- For size 10 (small), the height is 100÷log(10) = 100÷1 = 100
- For size 100 (wide), the height is 100÷log(100) = 100÷2 = 50

There are other logarithms in addition to this base 10 version. There are other uses for logarithms than lessening changes. So pay attention when they come up in Math class!

After the constructor is done, add the `draw()` method just below the `Basket` constructor.

```
Basket.prototype.draw = function() {
  var cover = new THREE.MeshPhongMaterial({
    color: this.color,
    shininess: 50,
    specular: 'white'
  });

  var shape = new THREE.CubeGeometry(this.size, 1, this.size);
  var goal = new Physijs.BoxMesh(shape, cover, 0);
  goal.position.y = this.height / 100;
  scene.add(goal);
}
```

We start the `draw()` method by building a shiny material with our random color. Then we build a flat cube with a `y` height of `1` and `x` and `z` width and depth of

`this.size`. The physics mesh needs a small height—enough to keep it above other, shorter meshes. Finally, we add it to the scene.

Let's build a `Basket` object to make sure we have everything correct so far. Add the following below the scoreboard code:

```
var goal1 = new Basket(200, 10);
```

If everything is typed correctly, then you should see a square platform in the center of the scene. It's a size 200 square worth 10 points. You can even try launching a few balls at it—we haven't added scoring yet, but you should still be able to hit the target. And if it doesn't work… check the JavaScript console.

Before finishing up the baskets, add a second goal to the scene.

```
var goal1 = new Basket(200, 10);
» var goal2 = new Basket(40, 100);
```

This smaller one will be worth a whopping 100 points if we can hit it!

Back up in the `Basket.prototype.draw()` method, add sides to baskets so they look more like actual baskets. This is a lot of code, but it should all be familiar as we add back, front, right, and left sides.

```
Basket.prototype.draw = function() {
  var cover = new THREE.MeshPhongMaterial({
    color: this.color,
    shininess: 50,
    specular: 'white'
  });

  var shape = new THREE.CubeGeometry(this.size, 1, this.size);
  var goal = new Physijs.BoxMesh(shape, cover, 0);
  goal.position.y = this.height / 100;
  scene.add(goal);

» var halfSize = this.size/2;
» var halfHeight = this.height/2;
»
» shape = new THREE.CubeGeometry(this.size, this.height, 1);
» var side1 = new Physijs.BoxMesh(shape, cover, 0);
```

```
»   side1.position.set(0, halfHeight, halfSize);
»   scene.add(side1);
»
»   var side2 = new Physijs.BoxMesh(shape, cover, 0);
»   side2.position.set(0, halfHeight, -halfSize);
»   scene.add(side2);
»
»   shape = new THREE.CubeGeometry(1, this.height, this.size);
»   var side3 = new Physijs.BoxMesh(shape, cover, 0);
»   side3.position.set(halfSize, halfHeight, 0);
»   scene.add(side3);
»
»   var side4 = new Physijs.BoxMesh(shape, cover, 0);
»   side4.position.set(-halfSize, halfHeight, 0);
»   scene.add(side4);
»
»   this.waitForScore(goal);
};
```

Don’t forget that very last line!

The very last, new line inside the `draw()` method will break our code, but add it anyway. After all of the sides are added, we call the `waitForScore()` method. We’ll add that method next so that we can finally score points!

```
Basket.prototype.waitForScore = function(goal){
  goal.addEventListener('collision', this.score.bind(this));
};
```

That will then call the `score()` method (which we’ll add next) when a ball collides with the goal. If you’re sharp-eyed, you’ll notice that we had to bind `this` to the `score()` method. As described in Chapter 16, *Learning about JavaScript Objects*, this is a case of JavaScript making life hard for us programmers. Without `bind(this)`, collisions would call the `score()` method but would forget `this`. And without `this`, the `score()` method has no way to know how my points just got scored!

Yup. JavaScript really is crazy.

The very last thing to do here is define that `score()` method.

```
Basket.prototype.score = function(ball){
  if (scoreboard.getTimeRemaining() == 0) return;
  scoreboard.addPoints(this.points);
  scene.remove(ball);
};
```

If there’s no time left, then we return from the method without doing anything else. Otherwise, we add whatever points this goal is worth to the scoreboard. Lastly we remove the ball from the scene so a bounce won’t count twice.

Phew! That’s a fair amount of code, but if you have all of that correct, you should be able to play a fairly challenging game.

Wind!

Now that we're getting good with these JavaScript objects, let's try typing out the `Wind` object all at once. Start with the constructor, which can go below all of the methods for `Basket`—after the `score()` method. The constructor will call methods to draw the wind on the scene and start changing it.

```
function Wind() {
  this.draw();
  this.change();
}
```

As we've done with the other objects, the methods for `Wind` can follow after the constructor. First, add the `draw()` method, which will use the arrow helper again.

```
Wind.prototype.draw = function(){
  var dir = new THREE.Vector3(1, 0, 0);
  var start = new THREE.Vector3(0, 200, 250);
  this.arrow = new THREE.ArrowHelper(dir, start, 1, 'lightblue');
  scene.add(this.arrow);
};
```

This time, it places the arrow 200 above and 250 in front of the center. It also makes the arrow light blue to be more wind-like.

The `change()` method will use `Math.random()` to change the direction (-1 for left, 1 for right) and the strength of the wind.

```
Wind.prototype.change = function(){
  if (Math.random() < 0.5) this.direction = -1;
  else this.direction = 1;
  this.strength = 20*Math.random();

  this.arrow.setLength(5 * this.strength);
  this.arrow.setDirection(this.vector());

  setTimeout(this.change.bind(this), 10000);
};
```

After updating the arrow on the screen with a new length and direction, we use

`setTimeout()` to change the wind after 10 seconds. We first saw `setTimeout()` in Chapter 11, *Project: Fruit Hunt*.

And, yup. We had to bind `this` to the `change()` method. Otherwise we'd be asking JavaScript to change a ghost wind. Silly JavaScript.

We need one last method for the `Wind` object: a `vector()` that describes the wind in (x, y, z) format.

```
Wind.prototype.vector = function(){
  var x = this.direction * this.strength;
  return new THREE.Vector3(x, 0, 0);
};
```

With that, we're ready to add the wind to our game. Create the wind below the scoreboard and the two goals that we added in the previous sections.

```
var wind = new Wind();
```

We are nearly done. The wind arrow should now show up in our game and it should change every 10 seconds. We still need for the wind to affect the balls that have been launched.

For that, we first need a function that returns a list of all balls in the game. Add `allBalls()` below the new wind that we just added.

```
function allBalls() {
  var balls = [];
  for (var i=0; i<scene.children.length; i++) {
    if (scene.children[i].name.startsWith('Game Ball')) {
      balls.push(scene.children[i]);
    }
  }
  return balls;
}
```

This loops through everything in the scene, looking for just the game balls. We named the balls in `Launcher.prototype.launch()` "Game Ball." So the loop checks every object to see whether its name starts with "Game Ball." If so, it pushes the

object onto the list of balls. At the very end of `allBalls()`, that list is returned.

Finally, inside the `gameStep()` function, we loop through all of the balls in the game and apply a force to the center of each ball.

```
  function gameStep() {
    scene.simulate();

»   var balls = allBalls();
»   for (var i=0; i<balls.length; i++) {
»     balls[i].applyCentralForce(wind.vector());
»     if (balls[i].position.y < -100) scene.remove(balls[i]);
»   }
  }

    // Update physics 60 times a second so that motion is smooth
    setTimeout(gameStep, 1000/60);
  }
```

We also add a check to remove any ball that's fallen well below the baskets. We have no reason to keep track of them any more, so we might as well make it easier on the computer by removing them from the scene.

And that's it! Phew! That was a lot of work, but if everything is coded correctly, you should have a nice little game on your hands. If there are any problems, don't forget to check the JavaScript console.

This is a nice, challenging game. Try it to see if you can break 500 points!

	Let's Play!
	The difference between a good game and a great game is in the details. So play around with the code to make this even better. Add some more lights. Add shadows. Let the player press the R key to reset the game after it's finished.

The Code So Far

If you'd like to double-check the code in this chapter, it's included in *Code: Ready, Steady, Launch*. The code for resetting the game is included. You'll have to create the rest on your own!

What’s Next

This was not an easy chapter, so you’ve done quite well by making it this far. Congratulations!

At this point you’re starting to put it all together. JavaScript should feel a little more familiar to you and comfortable. It’s probably not so familiar that you could write a complex game from scratch without a lot of help from the web. But you’re getting closer.

And hopefully things like physics, lights, materials, and objects are starting to make sense. Let’s keep that progress moving in the next chapter.

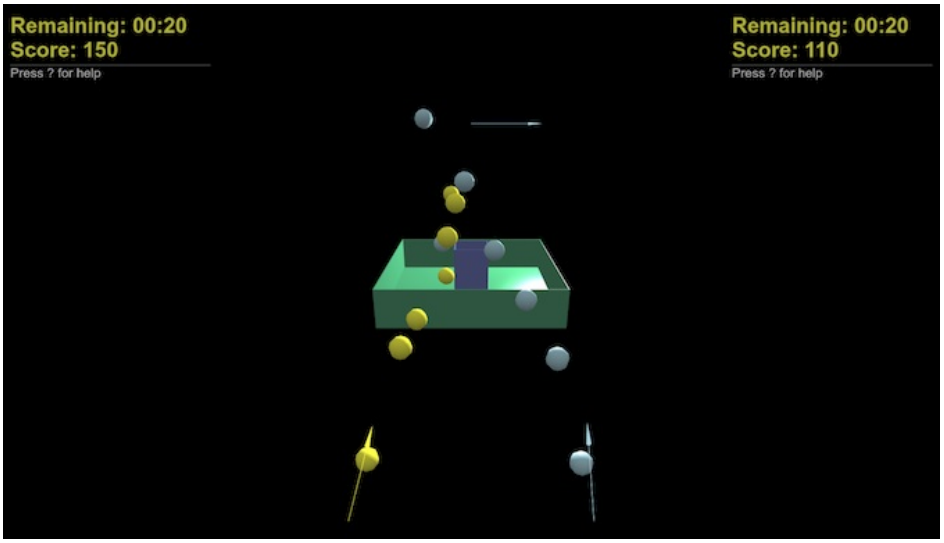
When you're done with this chapter, you will

- *Have a two-player game*
- *Better understand why objects are so important in programming*
- *Begin to understand what's involved in online multi-player games*

Chapter 18

Project: Two-Player Games

Multi-player games are hard to build. But we've done a lot of hard things in this book already, so we're not going to let that stop us!



In this chapter, we're going to turn the Ready, Steady, Launch game from the last chapter into a two-player game, which should look something like the picture above. This isn't going to be an online, multi-player game. Both players will use the same computer and keyboard to play. As we will see, just doing this is hard enough. But, we'll build a fun game (you can even try to knock the other player's balls out of the sky!) and begin to see how we might convert it into an online game someday.

The big changes that we need to make to the original Ready, Steady, Launch game are:

1. We’ll need to add two launchers instead of one.
2. Each launcher will need its own scoreboard.
3. Baskets will need to add points to the correct scoreboard.

As we’re doing this, pay special attention to how objects make this all possible.

Getting Started

We start by creating a copy of `Ready, Steady, Launch`, naming it `Ready, Steady, Launch 2`.

This is a gentle reminder of a tip from all the way back in Chapter 4, *Project: Moving Avatars*. If you have working code, always make sure you have a copy somewhere. You might think your next changes are small and couldn't possibly break things. It's super easy to break things badly when programming, though. When that happens, a backup is like gold. You can refer to your backup or delete your new code and start again. Which is why we do it again here.

Two Launchers

Thanks to the `Launcher` object that we built in the last chapter, we can actually create two launchers already. They won't work quite right, but sometimes you have to break things before you can put them back together a new way.

If you followed the code placement suggestions in the previous chapter, your code starts with the `Launcher` constructor, followed by methods on the `Launcher` prototype. Then we have the `Basket` constructor, followed by its methods. Last is the `Wind` constructor and its methods.

After the `Wind` methods, we have a line that creates a single launcher:

```
var launcher = new Launcher();
```

	Search Your Code If You Can't Find It
	`Ctrl+F` or `⌘+F` in 3DE will open the find feature. Type a few of the characters that you expect to find and 3DE will take you to the line that contains it. For the new launcher, try "new Laun".

Let's rename `launcher` to `launcher1` and tell this launcher that it'll live on the left side of the screen. Also, let's add a second launcher, which will live on the right side.

```
var launcher1 = new Launcher('left');
var launcher2 = new Launcher('right');
```

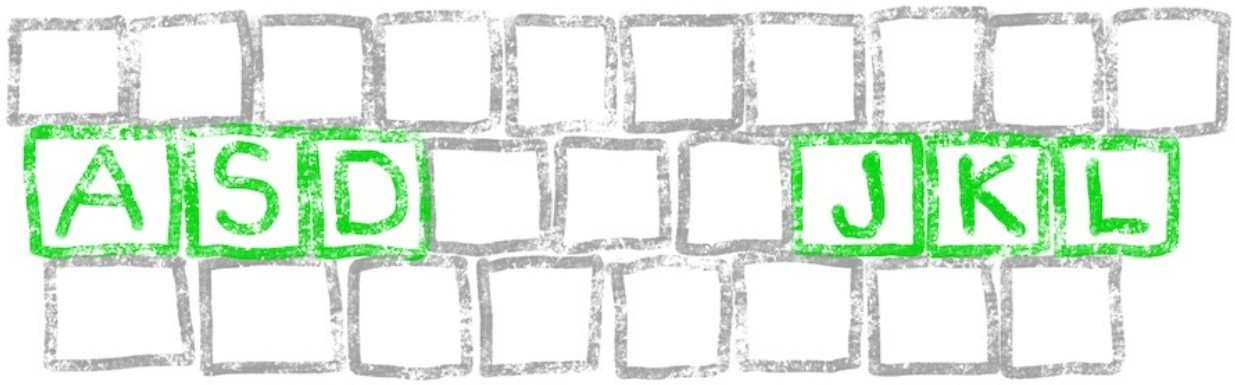
That change won't look like it changes anything. It does break something, however. If you hide the code and try to move or launch with the arrow keys,

nothing will happen. And the JavaScript console will have errors like: “launcher is not defined.”

This is because the keyboard code, which should be at the very bottom of our code, is trying to move the old `launcher`.

```
document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;
  if (code == 'ArrowLeft') launcher.moveLeft();
  if (code == 'ArrowRight') launcher.moveRight();
  if (code == 'ArrowDown') launcher.powerUp();
  if (code == 'KeyR') reset();
}
```

We need to change `launcher` to `launcher1`. We also have to add controls to move `launcher2`. Instead of letting one player use the arrow keys and the other player use something else, we’re going to move both controls to keys in the middle row of the keyboard. In the U.S., most keyboards start with the letters “A,” “S,” and “D” in the middle row. On the right side, these keyboards end with “J,” “K,” and “L.”



We’ll use A and D to move `launcher1` and S, which is between those two, to launch. For `launcher2`, J and L will move, and K will launch. To make that work, change the “keydown” listener as shown.

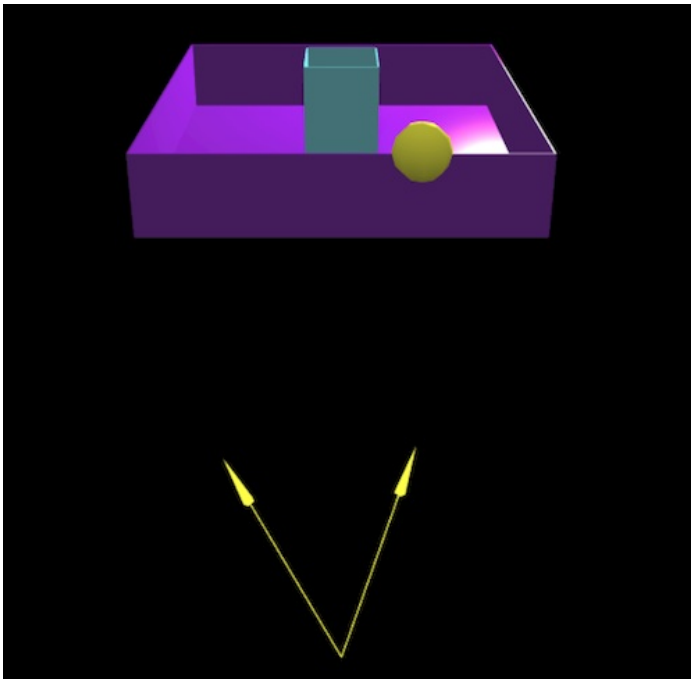
```
document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
```

```
  var code = event.code;
»   if (code == 'KeyA') launcher1.moveLeft();
»   if (code == 'KeyD') launcher1.moveRight();
»   if (code == 'KeyS') launcher1.powerUp();
»
»   if (code == 'KeyJ') launcher2.moveLeft();
»   if (code == 'KeyL') launcher2.moveRight();
»   if (code == 'KeyK') launcher2.powerUp();

  if (code == 'KeyR') reset();
}
```

And we also need to change the “keyup” listener.

```
document.addEventListener('keyup', sendKeyUp);
function sendKeyUp(event){
  var code = event.code;
»   if (code == 'KeyS') launcher1.launch();
»   if (code == 'KeyK') launcher2.launch();
}
```

After making these changes, the game shouldn’t look any different. But if you hide the code, the keyboard controls should be different. The arrow keys no longer work. And if you move one launcher with the A/D keys and the other with the J/L keys, you should be able to see that there are, in fact, two launchers that two people can control!



Be sure to try this out, and make sure there are no errors in the JavaScript console. It's a lot easier to correct an error right after it's introduced than when you're trying to add completely different code.

	Small Changes Win
	When adding features, take small steps and constantly check that no unexpected errors occurred.

Next, let's move the two launchers. We're already telling our launchers that they need to be on the left or right:

```
var launcher1 = new Launcher('left');
var launcher2 = new Launcher('right');
```

But our launcher prototype has no idea what that means. Let's teach it.

We start in the `Launcher` constructor, which should be at or near the top of the code. Right now, the constructor doesn't do anything with the `'left'` or `'right'` location that we're sending it.

```
function Launcher() {
  this.angle = 0;
  this.power = 0;
  this.draw();
}
```

For `Launcher` to know what to do, we make the following changes.

```
» function Launcher(location) {
»   this.location = location;
»   this.color = 'yellow';
»   if (location == 'right') this.color = 'lightblue';
  this.angle = 0;
  this.power = 0;
  this.draw();
}
```

The `location` argument on the first line tells our constructor to expect one argument, which we name "location" to make it easy to remember what it's supposed to do: set the location of the launcher. Then we set the color to yellow (which is what the launcher and balls were in the single player version of game). Last, we set the color to light blue if the launcher is on the right side of the screen.

After making those code changes, the game should look exactly the same with no errors. To actually see a difference, we need to draw the launcher arrows in different places, and we need to use different colors for the arrows and balls.

Change the `draw()` method as shown to move the arrow and change the color:

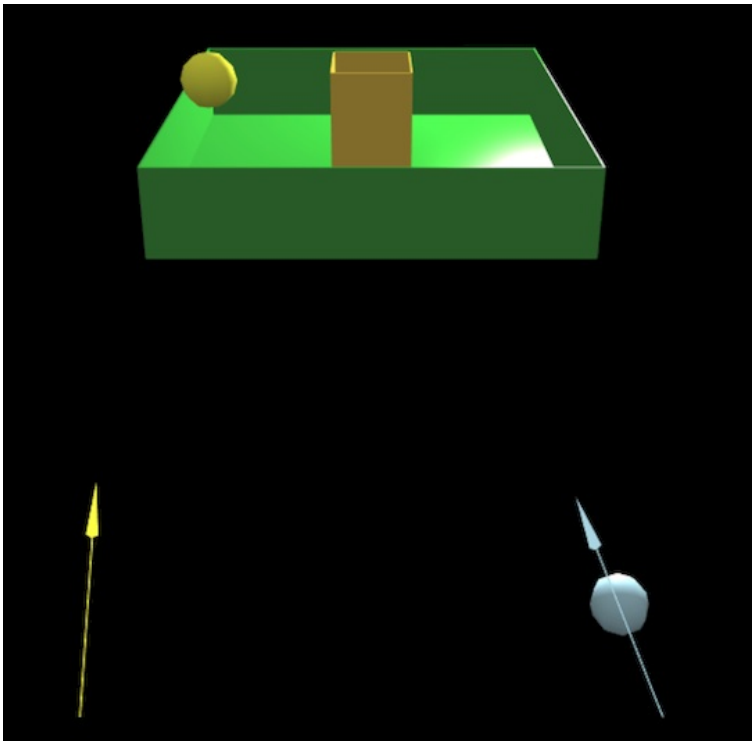
```
Launcher.prototype.draw = function() {
  var direction = new THREE.Vector3(0, 1, 0);
»   var x = 0;
»   if (this.location == 'left') x = -100;
»   if (this.location == 'right') x = 100;
»   var position = new THREE.Vector3( x, -100, 250 );
```

```
    var length = 100;
    this.arrow = new THREE.ArrowHelper(
      direction,
      position,
      length,
      this.color
    );
    scene.add(this.arrow);
  };
```

Change the `launch` method to start the balls from the new `location` and to change the color of the balls.

```
  Launcher.prototype.launch = function(){
    var shape = new THREE.SphereGeometry(10);
    var material = new THREE.MeshPhongMaterial({color: this.color});
    var ball = new Physijs.SphereMesh(shape, material, 1);
    ball.name = 'Game Ball';
    var p = this.arrow.position;
    ball.position.set(p.x, p.y, p.z);
    scene.add(ball);

    var speedVector = new THREE.Vector3(
      2.5 * this.power * this.vector().x,
      2.5 * this.power * this.vector().y,
     -80
    );
    ball.setLinearVelocity(speedVector);

    this.power = 0;
    this.arrow.setLength(100);
  };
```

If no errors are in the code, then we should have two launchers, in two different places, controlled by two different sets of keys, with two different colors!



That’s pretty cool. Going from one player to two is not a small thing, but thanks to the `Launcher` object—and taking small steps—we’re well on our way.

Even though we have two players, we still have one scoreboard. Let’s give each player their own scoreboard next.

Two Scoreboards

Happily, the `Scoreboard` code that we use is an object, which makes our job of adding two scoreboards easier—yay! The code that adds the one scoreboard should be below where we create `launcher1` and `launcher2`. It should look something like the following:

```
var scoreboard = new Scoreboard();
scoreboard.countdown(60);
scoreboard.score(0);
scoreboard.help(
  'Use right and left arrow keys to point the launcher. ' +
  'Press and hold the down arrow key to power up the launcher. ' +
  'Let go of the down arrow key to launch. ' +
  'Watch out for the wind!!!'
);
scoreboard.onTimeExpired(timeExpired);
function timeExpired() {
  scoreboard.message("Game Over!");
}
```

This code creates one scoreboard for the entire game. But we want one scoreboard for each `Launcher`. To do that, we will *carefully* cut and paste all of this code. Highlight this code in 3DE, starting with `var scoreboard` all the way to the closing curly brace after `"Game Over"`, then cut (`Ctrl+X` or `⌘+X`) from its current location. Then we want to paste (`Ctrl+V` or `⌘+V`) the code right below the last `Launcher` prototype method, which should be `launch()`.

Then, make this a method for `Launcher` by adding a `keepScore` method and a closing curly brace. Take some time to indent everything cleanly, otherwise this code will look strange or wrong when you see it later. This new `keepScore` method should look like the following:

```
» Launcher.prototype.keepScore = function(){
  var scoreboard = new Scoreboard();
  scoreboard.countdown(60);
  scoreboard.score(0);
  scoreboard.help(
```

```
      'Use right and left arrow keys to point the launcher. ' +
      'Press and hold the down arrow key to power up the launcher. ' +
      'Let go of the down arrow key to launch. ' +
      'Watch out for the wind!!!'
    );
    scoreboard.onTimeExpired(timeExpired);
    function timeExpired() {
      scoreboard.message("Game Over!");
    }
» };
```

After we make that change, the scoreboard disappears. And so does everything else. Checking the JavaScript console, we should see an error that "scoreboard is not defined." That's because the `animate()` and `gameStep()` functions use the scoreboard to stop the game when time runs out. Those two functions still need to know when time is up, so we have to let them know somehow.

The easiest way to do this is to have the `keepScore()` method save its scoreboard in the launcher's `scoreboard` property.

```
  Launcher.prototype.keepScore = function(){
    var scoreboard = new Scoreboard();
    scoreboard.countdown(60);
    scoreboard.score(0);
    scoreboard.help(
      'Use right and left arrow keys to point the launcher. ' +
      'Press and hold the down arrow key to power up the launcher. ' +
      'Let go of the down arrow key to launch. ' +
      'Watch out for the wind!!!'
    );
    scoreboard.onTimeExpired(timeExpired);
    function timeExpired() {
      scoreboard.message("Game Over!");
    }
»   this.scoreboard = scoreboard;
  };
```

Then, in the `Launcher` constructor above, call this `keepScore()` method:

```
  function Launcher(location) {
    this.location = location;
    this.color = 'yellow';
```

```
    if (location == 'right') this.color = 'lightblue';
    this.angle = 0;
    this.power = 0;
    this.draw();
»   this.keepScore();
  }
```

Finally, after the code that creates `launcher1` and `launcher2`, let's add a new `scoreboard` variable that comes from the `scoreboard` property of one of the launchers.

```
  var launcher1 = new Launcher('left');
  var launcher2 = new Launcher('right');
» var scoreboard = launcher1.scoreboard;
```

After making that change, we should be back to where we started. The baskets should again be in the middle of the screen. We should have two launchers. And we should see only one scoreboard.

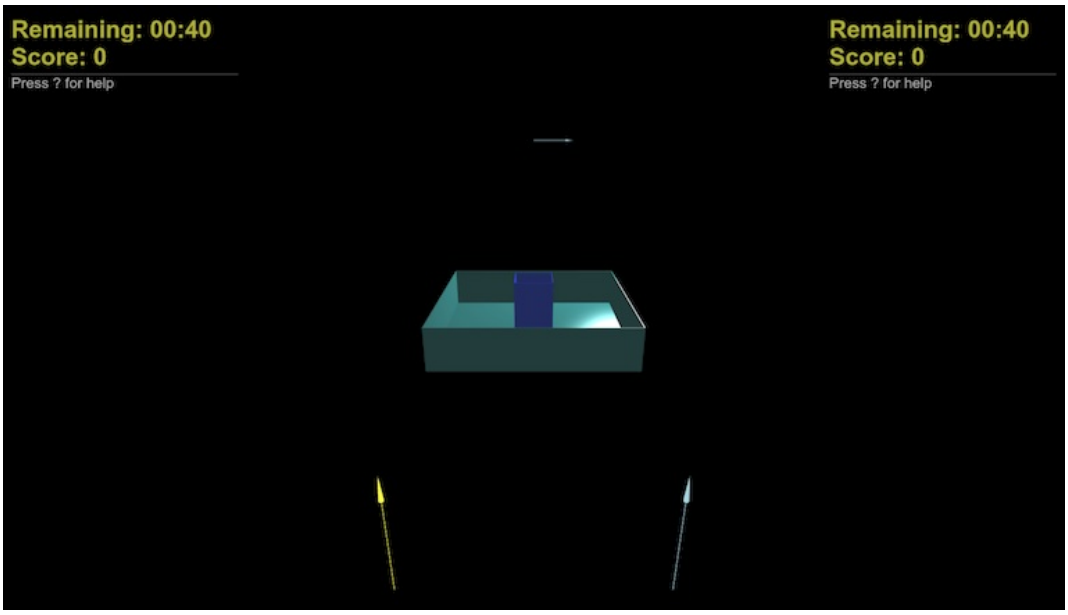
We only *see* one scoreboard, but there are actually two: right on top of each other. Next, we need to tell each launcher's scoreboard where it should live. So far, our scoreboards have all lived in the top-right of the scene. But the `Scoreboard` constructor can live in four different places: `'topleft'`, `'topright'`, `'bottomleft'`, and `'bottomright'`.

Since the `location` property of `Launcher` already holds either `'left'` or `'right'`, we can combine that with `'top'` to get the correct value to send to `Scoreboard`. The following change in the `keepScore()` method does the trick:

```
  Launcher.prototype.keepScore = function(){
»   var scoreboard = new Scoreboard('top' + this.location);
    scoreboard.countdown(60);
    scoreboard.score(0);
    scoreboard.help(
      'Use right and left arrow keys to point the launcher. ' +
      'Press and hold the down arrow key to power up the launcher. ' +
      'Let go of the down arrow key to launch. ' +
      'Watch out for the wind!!!'
    );
    scoreboard.onTimeExpired(timeExpired);
```

```
    function timeExpired() {
      scoreboard.message("Game Over!");
    }
    this.scoreboard = scoreboard;
  };
```

With that, two separate scoreboards should be there, one for each launcher.



That’s looking good, but we’re not done yet. We have two scoreboards, but scores are sent to only one of them. So next, we’ll update the basket code so that it can update the scoreboard for the proper launcher.

Teaching Baskets to Update the Correct Scoreboard

In the last chapter, we defined what a basket does when a score happens, as follows:

```
Basket.prototype.score = function(ball){
  if (scoreboard.getTimeRemaining() == 0) return;
  scoreboard.addPoints(this.points);
  scene.remove(ball);
};
```

The `score()` method knows about the ball that fell into it and the scoreboard that keeps the score. The ball is an argument to the method. The scoreboard is the one that used to be defined after the `launcher` (which became `launcher1` and `launcher2`).

We can't use that scoreboard anymore. The basket needs to know which scoreboard gets points added for the ball that just landed. But how can the basket know which scoreboard to use if all it knows is that the ball landed?

The answer to that question is that we can attach the scoreboard to the ball when it's launched. We do that with one change up in the `launch()` method of the `Launcher` prototype.

```
Launcher.prototype.launch = function(){
  var shape = new THREE.SphereGeometry(10);
  var material = new THREE.MeshPhongMaterial({color: this.color});
  var ball = new Physijs.SphereMesh(shape, material, 1);
  ball.name = 'Game Ball';
» ball.scoreboard = this.scoreboard;
  var p = this.arrow.position;
  ball.position.set(p.x, p.y, p.z);
  scene.add(ball);

  var speedVector = new THREE.Vector3(
    2.5 * this.power * this.vector().x,
    2.5 * this.power * this.vector().y,
   -80
  );
```

```
    ball.setLinearVelocity(speedVector);

    this.power = 0;
    this.arrow.setLength(100);
  };
```

This is an important little change. In this method—this `Launcher` method, we launch the ball that will eventually hit a basket. We add a new property to the `ball`, a `scoreboard` property. We assign `this.scoreboard` to that `ball.scoreboard` property. Since we're in the `Launcher`, `this.scoreboard` is the same scoreboard that we created in `keepScore()`. And just like that, the ball now knows which scoreboard should get points when it lands.

So, down in the `score()` method of `Basket`, we assign the `scoreboard` variable to `ball.scoreboard`.

```
  Basket.prototype.score = function(ball){
»   var scoreboard = ball.scoreboard;
    if (scoreboard.getTimeRemaining() == 0) return;
    scoreboard.addPoints(this.points);
    scene.remove(ball);
  };
```

Now, when a ball makes it into a basket, the basket tells the ball's scoreboard to add points! The game is really shaping up now, but there are still two small bugs —two problems—that we ought to fix before playing someone else.

Sharing a Keyboard

The first bug has to do with powering up the launcher. If player #1 holds down the `S` key, the launcher powers up like normal. But, if player #2 then presses the `K` key, player #1's power up stops.

This bug happens because there's only one keyboard and keyboards expect that only one person is going to press and hold a key. If you press and hold the `S` key in a word processor, you'll get a bunch of `S`s in your document—the keyboard repeats the `S` letter. If you keep the `S` key down and then hold down the `K` key at the same time, the computer figures you now want a bunch of `K`s added to the document, and switches to repeating the `K` key.

To make power-ups work when two people are sharing the keyboard, we need to replace keyboard repeats with JavaScript repeats. We learned JavaScript repeats in Chapter 14, *Project: The Purple Fruit Monster Game*. JavaScript repeats are called *intervals*. We need two intervals, one for each player. We'll start each interval when the power-up key is pressed. To do this, make the highlighted changes in the `keydown` listener.

```
» var powerUp1;
» var powerUp2;
» function powerUpLauncher1(){ launcher1.powerUp(); }
» function powerUpLauncher2(){ launcher2.powerUp(); }

  document.addEventListener('keydown', sendKeyDown);
  function sendKeyDown(event) {
»   if (event.repeat) return;

    var code = event.code;
    if (code == 'KeyA') launcher1.moveLeft();
    if (code == 'KeyD') launcher1.moveRight();
»   if (code == 'KeyS') {
»     clearInterval(powerUp1);
»     powerUp1 = setInterval(powerUpLauncher1, 20);
»   }
```

```
    if (code == 'KeyJ') launcher2.moveLeft();
    if (code == 'KeyL') launcher2.moveRight();
»   if (code == 'KeyK') {
»     clearInterval(powerUp2);
»     powerUp2 = setInterval(powerUpLauncher2, 20);
»   }

    if (code == 'KeyR') reset();
  }
```

There are identical power-ups for players #1 and #2. Looking closely at player #1's power-ups, we see:

① A variable to hold an active power-up interval.

② A function that tells the launcher to power up. This only adds a tiny bit of power to the launcher—we'll have to call it a lot to fully power up.

③ Plain old keyboard repeats are not allowed, thanks to the special `repeat` property. If this key-down event is a keyboard repeat, we return out of the keyboard listener function without doing anything.

④ If the power-up interval is already active, stop it.

⑤ Start the `powerUp1` interval, running the `powerUpLauncher1()` function once every 20 milliseconds. That's 50 times a second, which should power up the launcher nice and fast.

That fixes power-ups for two players, but we still need to launch after powering up. To do that, make the highlighted changes in the `keyup` listener.

```
  document.addEventListener('keyup', sendKeyUp);
  function sendKeyUp(event){
    var code = event.code;
»   if (code == 'KeyS') {
»     launcher1.launch();
»     clearInterval(powerUp1);
»   }
»   if (code == 'KeyK') {
»     launcher2.launch();
```

```
»     clearInterval(powerUp2);
»   }
  }
```

We still launch when the power-up key is released. We also need to clear the power-up interval so that launcher is ready to start all over again. Now, both player #1 and player #2 can use the same keyboard to power up and launch at the same time.

A Complete Reset

The last bug is in our `reset()` function. We're only resetting player #1's scoreboard.

```
  function reset() {
    if (scoreboard.getTimeRemaining() > 0) return;
»   scoreboard.score(0);
»   scoreboard.countdown(60);

    var balls = allBalls();
    for (var i=0; i<balls.length; i++) {
      scene.remove(balls[i]);
    }

    animate();
    gameStep();
  }
```

We want to reset the score to 0 and the countdown to 60 seconds for both players, not one. So let's cut those two lines with `Ctrl+X` or `⌘+X`.

Paste them below the last method for `Launcher`, which should be `keepScore()`, and above the `Basket()` function constructor. Then make a `reset()` method for `Launcher` by adding the highlighted code:

```
» Launcher.prototype.reset = function(){
»   var scoreboard = this.scoreboard;
»   if (scoreboard.getTimeRemaining() > 0) return;
    scoreboard.score(0);
    scoreboard.countdown(60);
» };
```

That teaches every launcher that gets created—like the two for player #1 and player #2—how to reset themselves and their scoreboards.

All that's left is to move back down to the `reset()` function. In there, we call that `reset()` method on both launchers.

```
  function reset() {
    if (scoreboard.getTimeRemaining() > 0) return;

»   launcher1.reset();
»   launcher2.reset();

    var balls = allBalls();
    for (var i=0; i<balls.length; i++) {
      scene.remove(balls[i]);
    }

    animate();
    gameStep();
  }
```

Now our game is *really* ready for playing!

Let's Play!	
	The help message for the scoreboards still tells players to use the arrow keys. Can you change the messages so they show the new keys? Can you get the two different scoreboards to show the correct message for their launcher?

The Code So Far

In case you'd like to double-check the code in this chapter, you can find it in *Code: Two-Player Ready, Steady, Launch*. The code for the updated help messages is included if you need some hints.

What’s Next

Phew! That was quite a bit of work. But if you look back, it didn’t take *that* many changes to get there. We had to play a few object tricks—especially to get baskets to tell the correct scoreboard to add points. Mostly, we found that having the launcher and scoreboards as objects made the switch from one player to two fairly easy.

As easy as it was, it still required effort. And things only get harder when games move online, which is why we won’t cover them in this book. But you’ve now seen enough of multi-player games to have a sense of how they work. Keep working at programming and you’ll be able to build incredible multi-player games before you know it!

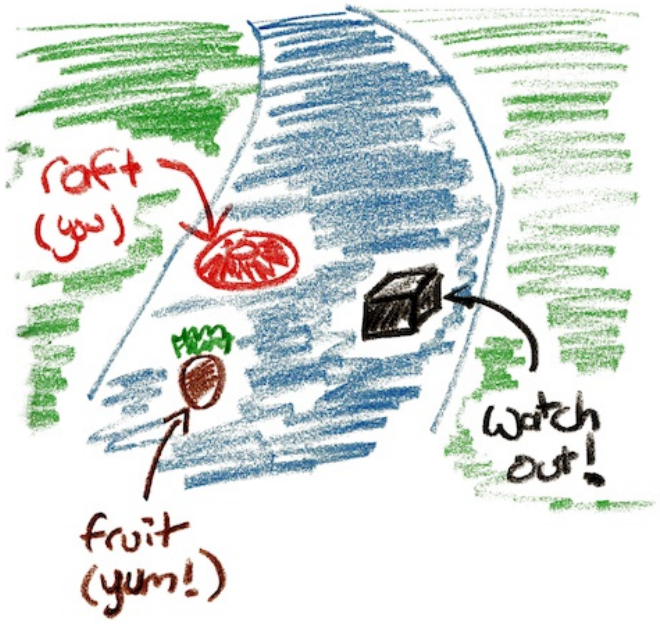
When you're done with this chapter, you will

- *Understand how to warp shapes into something completely different*
- *Be able to design a world—with terrain as rough or smooth as you like*
- *Complete another fun 3D game*

Chapter 19

Project: River Rafter

For our final project, let's build a river rafter game in which the player needs to navigate a raft along a raging river, dodging obstacles and picking up bonus points wherever possible. The game will look something like this sketch:



This is going to be a hard chapter. We're using all of the skills we've learned in this book and then learning a bunch of new things. So it's a really good idea to save this chapter for last.

That said, this is going to be a blast, so let's get started!

Getting Started

We begin by creating a new project in the 3DE Code Editor. We use the **3D starter project (with Physics)** template and call it **River Rafter**.

At the top of our code, let's pull in some code collections that are going to help us out.

```
    <body></body>
    <script src="/three.js"></script>
    <script src="/physi.js"></script>
»   <script src="/controls/OrbitControls.js"></script>
»   <script src="/scoreboard.js"></script>
»   <script src="/noise.js"></script>
```

The orbit controls, which we used in Chapter 12, *Working with Lights and Materials*, will let us look around the world that we'll build. We know the scoreboard code well by now. The *noise* code is new. It will let us do some pretty wild stuff to the shapes in the scene. First up, let's decrease the gravity in this game from **-100** to **-10**. This will make the raft move a little slower down the river.

```
    var scene = new Physijs.Scene();
»   scene.setGravity(new THREE.Vector3( 0, -10, 0 ));
```

We'll want shadows in this game, so decrease the strength of the ambient light to **0.2**.

```
»   var light = new THREE.AmbientLight('white', 0.2);
    scene.add(light);
```

Just below those lines, add the directional light from Chapter 12, *Working with Lights and Materials*.

```
    var sunlight = new THREE.DirectionalLight('white', 0.8);
    sunlight.position.set(4, 6, 0);
    sunlight.castShadow = true;
    scene.add(sunlight);
    var d = 10;
```

```
sunlight.shadow.camera.left = -d;
sunlight.shadow.camera.right = d;
sunlight.shadow.camera.top = d;
sunlight.shadow.camera.bottom = -d;
```

To better see what we're building, let's change the camera settings slightly.

```
  var aspectRatio = window.innerWidth / window.innerHeight;
» var camera = new THREE.PerspectiveCamera(75, aspectRatio, 0.1, 100);
» camera.position.set(-8, 8, 8);
  scene.add(camera);
```

When we create the camera, we change the last two numbers to `0.1` and `100`. We want to see a little closer in this game—as close as `0.1` away from things. And we won't need to see very far away—`100` should be more than enough. We also move the camera to the left by 8, up 8, and back 8.

This is an outdoor, daytime game, so let's make a blue sky. As we did in Chapter 14, *Project: The Purple Fruit Monster Game*, change the color of the entire scene by setting the "clear" color to sky blue. While we're making a change to the renderer, let's also enable shadows in this scene.

```
  var renderer = new THREE.WebGLRenderer({antialias: true});
» renderer.setClearColor('skyblue');
» renderer.shadowMap.enabled = true;
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);
```

The last bit of setup is to add the orbit controls from the code collection. Add this just above the `START CODING` line.

```
new THREE.OrbitControls(camera, renderer.domElement);
```

We'll remove the orbit controls in a bit, but they're handy for getting a better look as we build the river. Remember, these controls let us click and drag the scene. Scrolling with the mouse or touchpad will zoom in and out. And the arrow keys move up, down, left, and right.

As long as the scene is now blue and no errors are in the JavaScript console, that's all we need to do above the `START CODING` line. Let's move down into the

rest of the code and start making stuff!

Pushing and Pulling Shapes

In 3D programming, shapes are much easier to change than you might guess. As far back as Chapter 1, *Project: Creating Simple Shapes*, we've known how to change a shape's size, chunkiness, and rotation. It turns out that we can do even more. Lots more.

To see this, add a call to `addGround()` below the `START CODING` line. Then define the `addGround()` function as shown:

```
var ground = addGround();

function addGround() {
  var faces = 99;
  var shape = new THREE.PlaneGeometry(10, 20, faces, faces);

  var _cover = new THREE.MeshPhongMaterial({color: 'green', shininess: 0});
  var cover = Physijs.createMaterial(_cover, 0.8, 0.1);

  var mesh = new Physijs.HeightfieldMesh(shape, cover, 0);
  mesh.rotation.set(-0.475 * Math.PI, 0, 0);
  mesh.receiveShadow = true;
  mesh.castShadow = true;

  scene.add(mesh);
  return mesh;
}
```

Once you have that code typed in and the JavaScript console shows no errors, we should see a flat, green square in the middle of the scene.

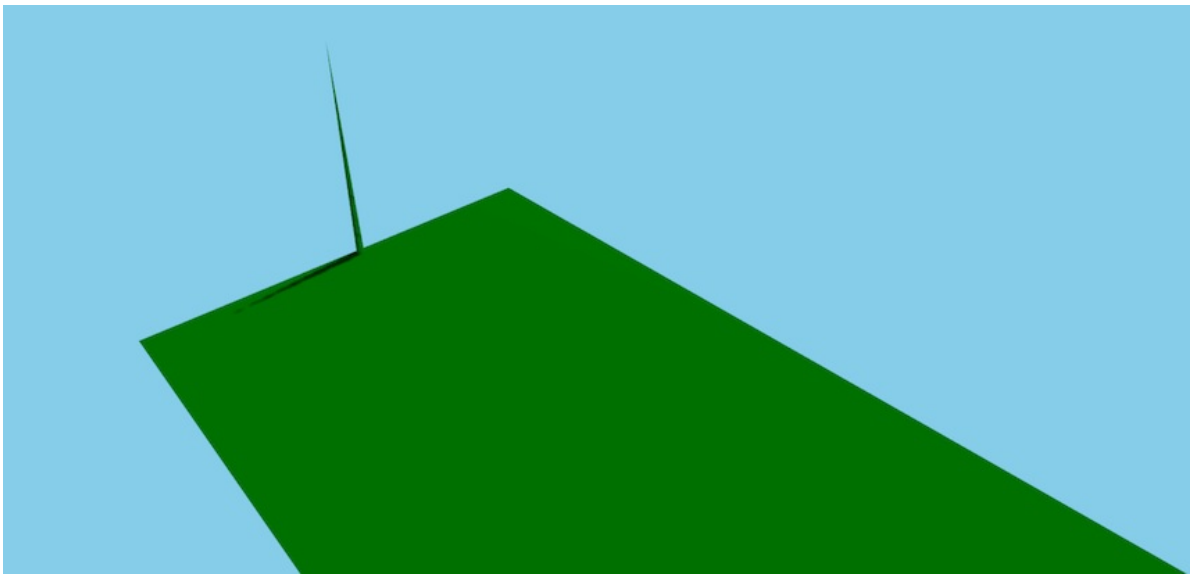
```
var ground = addGround();

function addGround() {
  var faces = 99;
  var shape = new THREE.PlaneGeometry(10, 20, faces, faces);

  var _cover = new THREE.MeshPhongMaterial({color: 'green', shininess: 0});
  var cover = Physijs.createMaterial(_cover, 0.8, 0.1);

  var mesh = new Physijs.HeightfieldMesh(shape, cover, 0);
  mesh.rotation.set(-0.475 * Math.PI, 0, 0);
  mesh.receiveShadow = true;
  mesh.castShadow = true;

  scene.add(mesh);
  return mesh;
}
```

There's some new stuff in there—and we'll talk about that in a bit—but mostly this code should be familiar. We create a rectangle/plane shape and a green cover, combine them into a physics-enabled mesh, and then rotate it flat and add it to the scene.

Now try this… Inside the `addGround()` function, right below the line that creates `shape`, add the following line:

```
    var shape = new THREE.PlaneGeometry(10, 20, faces, faces);
»   shape.vertices[50].z = 5;
```

After entering that code, we see our flat plane change.

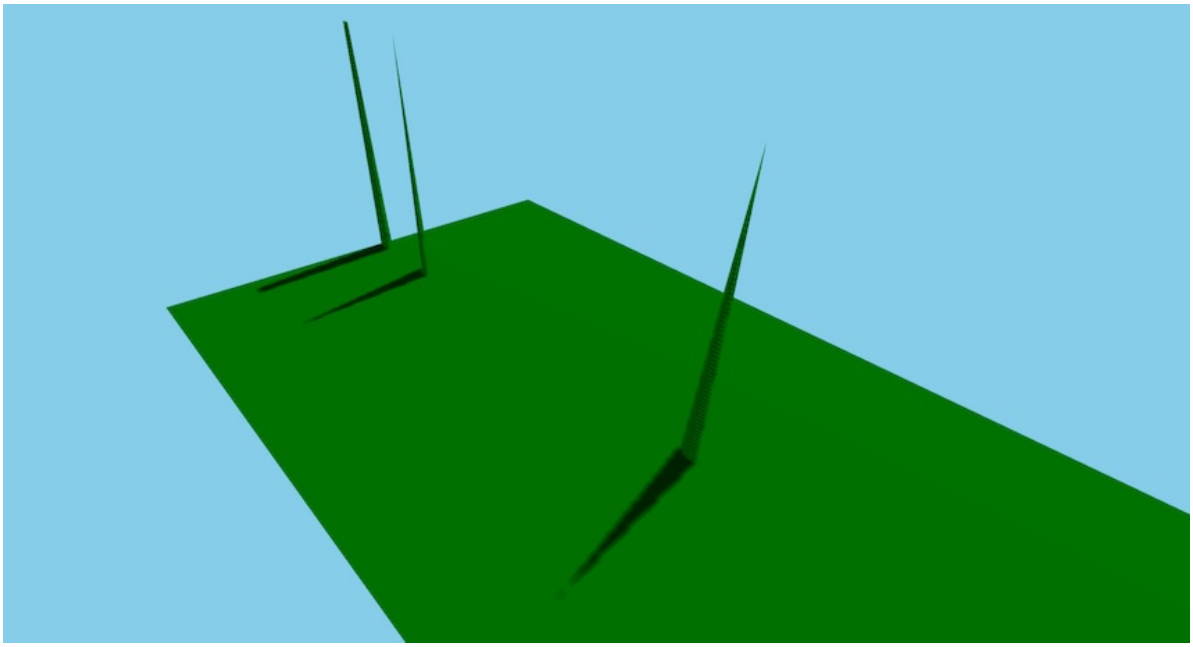


Whoa! That's pretty cool right? Now try this out.

```
    var shape = new THREE.PlaneGeometry(10, 20, faces, faces);
```

```
    shape.vertices[50].z = 5;
» shape.vertices[50 + 100].z = 5;
» shape.vertices[50 + 10*100].z = 5;
» shape.vertices[50 + 50*100].z = 5;
```

It should wind up looking something like the following:



	Let's Play!
	What happens when you set one of the `z` properties to a negative number? What happens if you add a line that also changes `shape.vertices[50].x`? What happens when you try other numbers inside the square brackets? Go crazy playing around with these lines—we'll delete them for something better in just a bit, so don't worry about breaking anything.

Changing meshes like this is how more complex shapes are usually built for computer games. Oftentimes designers build complex meshes in separate applications that let them do it with a few clicks of a mouse. These complex shapes are then loaded into games. We'll stick to coding right here in 3DE—

there's plenty that we can do. But first, let's take a closer look at this code.

What's Going on Here?

If you just want to code, skip ahead to *Digging a River*. But come back here—this section has some important stuff!

This code looks very much like the code we've written for planes and for physics-enabled meshes. But there are a few important differences.

Back in Chapter 1, *Project: Creating Simple Shapes*, we created a plane with X and Y sizes of `100` using `new THREE.PlaneGeometry(100, 100)`. Here, we make a plane with an X size of `10` and a Y size of `20`.

```
var faces = 99;
var shape = new THREE.PlaneGeometry(10, 20, faces, faces);
```

We also set the number of X and Y "faces" to 99 each. What we're doing here is telling the 3D code to build this simple rectangular plane from 99 smaller squares in both the X and Y directions. That is, instead of a single giant rectangle, this plane is made of 99 rectangles in both directions. If you do the math, 99 by 99 means that we have 9,801 total squares. That would be quite silly for a flat plane, but we'll make good use of all of these rectangles—or at least their corners.

These faces are important in 3D programming, so it's worth a quick look. We've seen faces before—they were the "chunks" from back in Chapter 1, *Project: Creating Simple Shapes*. When we program with them, they're rectangles, but computers don't like rectangles. Computers like triangles. So they take every face—every chunk—and split it into two triangles.

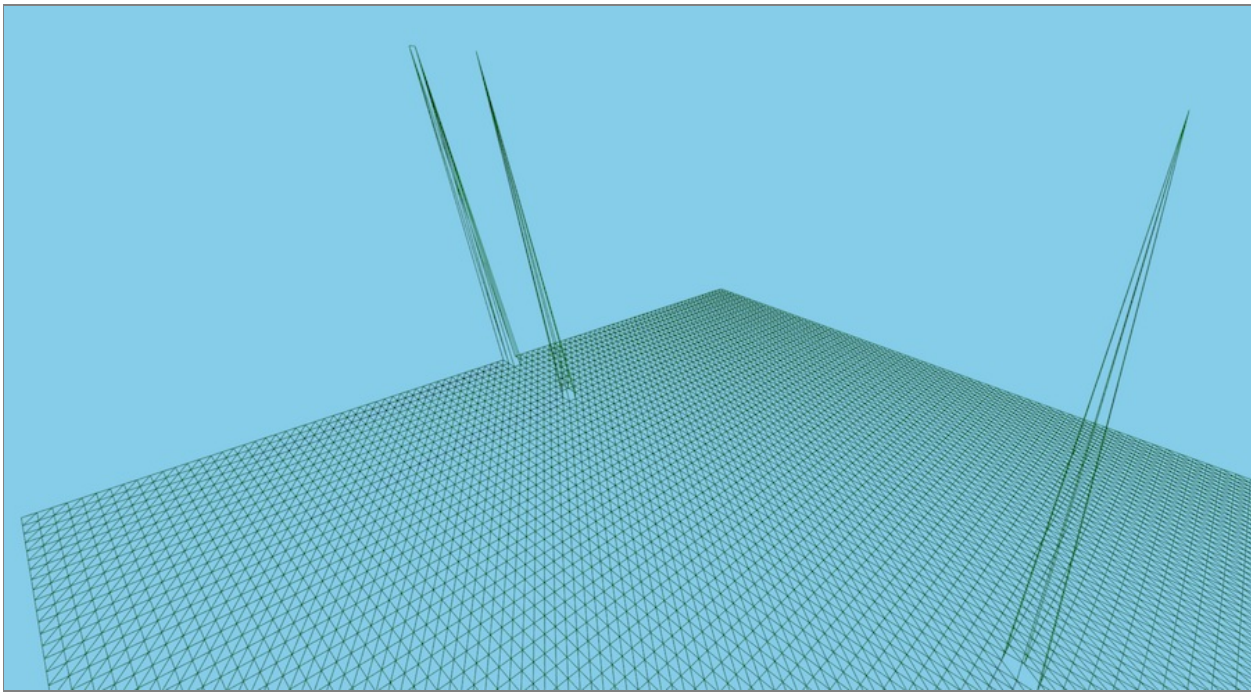
To see this, we can enable "wireframing" in the material.

```
var _cover = new THREE.MeshPhongMaterial({
  color: 'green',
  shininess: 0,
» wireframe: true
});
```

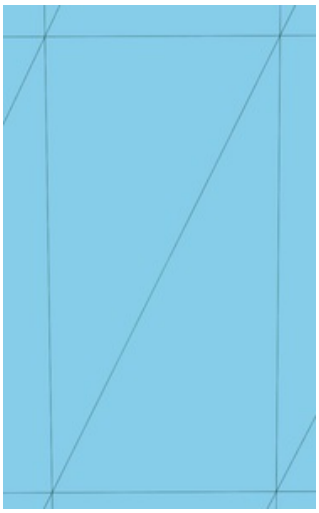
```
var cover = Physijs.createMaterial(_cover, 0.8, 0.1);
```

You can keep this all on one line—in the book, it's five shorter lines instead of one long one just so it fits! Be sure to add the comma after the 0 for `shininess` and the `wireframe` setting.

Wireframing hides the material, but still shows the faces that are used to build the shape. It should look something like this:

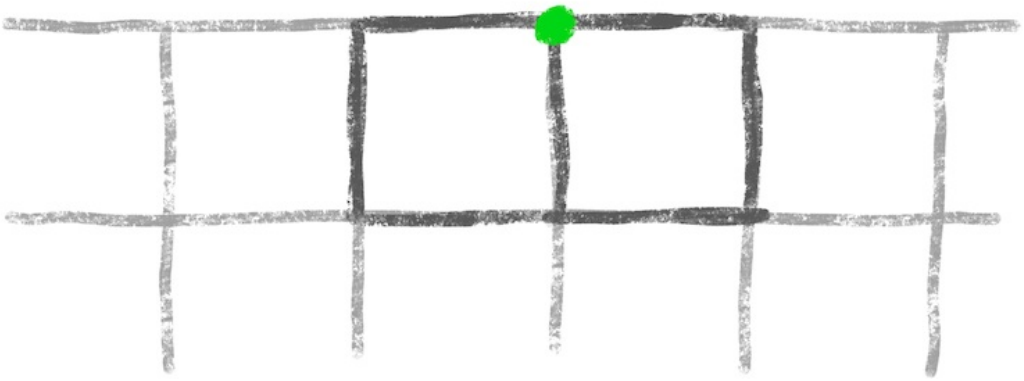


If you zoom in using your touchpad or scroll wheel on your mouse, you'll see that each face—each of the thousands of rectangles—are split into triangles.



For this game, it doesn't matter that computers use triangles. But all 3D programmers know that computers prefer triangles. And, since you are a 3D programmer, you needed to know that. So now you know!

Faces and triangles are important, but their corners are even more important. Our ground has thousands of rectangular faces all lying right next to each other. These rectangles share edges. They share corners. And we pulled four of those shared corners up by 5 (recall that two corners are right next to each other, while the other two are on rows 10 and 50).



These shared corners are called *vertices* in 3D programming. When we pulled `vertices[50]` up, the two faces that share that corner came with it. The faces are warped so that they remain connected to the rest of the plane.

You can remove the wireframe setting at this point. Let's look at the rest of this code. Notice that we create a physics material in two steps.

```
var _cover = new THREE.MeshPhongMaterial({color: 'green', shininess: 0});
var cover = Physijs.createMaterial(_cover, 0.8, 0.1);
```

First, we create a regular Phong material. This is grass, so we give it 0 shininess. We assign that to a temporary `_cover` variable—the underscore at the beginning is our way of telling ourselves that this variable will only be used briefly. In fact, it's only used on the next line where we create a physics-enabled material.

We don't always have to do it this way—we just used regular Phong materials in all of our other physics-enabled projects. We do it here so we can set those two

numbers, which are the material's friction and bounciness. Both of these values are a number between `0.0` and `1.0`. We set a high friction of `0.8`, meaning that it would be hard for something to slide across this ground material. We set a low bounciness of `0.1`, meaning that things won't bounce off of it.

The other new thing in the `addGround()` function is a different kind of physics-enabled mesh called a *height field mesh*.

```
var mesh = new Physijs.HeightfieldMesh(shape, cover, 0);
mesh.rotation.set(-0.475 * Math.PI, 0, 0);
mesh.receiveShadow = true;
mesh.castShadow = true;
```

We use this kind of mesh when working with warped or distorted shapes. Since we'll be warping the ground in the next section, this is the right mesh for us.

Also note that we don't quite rotate the ground flat. If we set the rotation around the X axis to `-0.5 * Math.PI`, then the ground would be perfectly flat. By setting it to `-0.475 * Math.PI`, we leave the ground at a slight angle. When we add a slippery raft to slippery water, the raft will slide down—kind of like the river is pushing it!

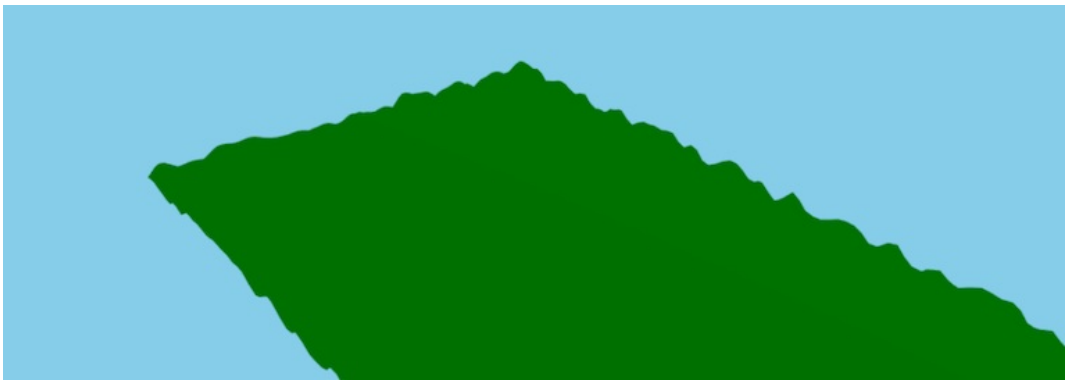
Rough Terrain

Ground is rarely perfectly flat. In graphics, we call rough terrain *noisy*, which is why we added that noise code collection in the setup.

Still inside the `addGround()` function, delete the lines that pulled up the vertices. Then, replace them with a loop that works through each vertex in the plane.

```
    var faces = 99;
    var shape = new THREE.PlaneGeometry(10, 20, faces, faces);

» var numVertices = shape.vertices.length;
» var noiseMaker = new SimplexNoise();
» for (var i=0; i<numVertices; i++) {
»   var vertex = shape.vertices[i];
»   var noise = 0.25 * noiseMaker.noise(vertex.x, vertex.y);
»   vertex.z = noise;
» }
```

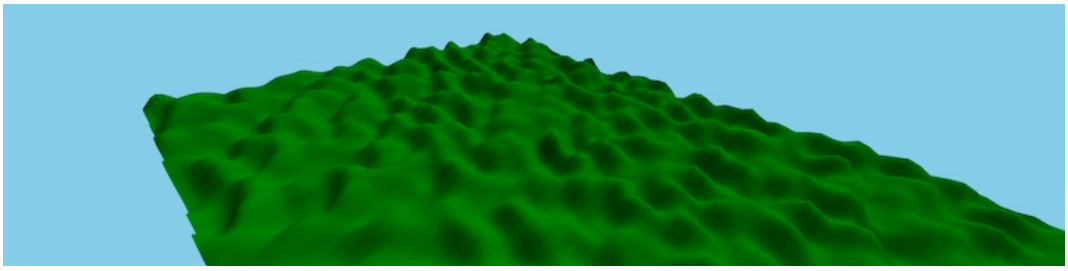
We don’t pull each vertex in the Z direction by the same amount. Instead, we pull each by the *noise*, which is calculated by code from the `noise.js` code collection. The result should be something like this.



The edges have bumps, but the middle is pretty boring. To make it more interesting, add the following two lines after the noise loop and above the `_cover` and `cover` lines:

```
    shape.computeFaceNormals();
    shape.computeVertexNormals();
```

This should look like:



That is much cooler!

It turns out that we need to call these two methods any time we push or pull vertices. 3D code keeps track of a *lot* of information so it can render scenes quickly and realistically. It can't keep track of out-of-the-ordinary changes, but it gives us a way to let the 3D code know when we've done something like pushing or pulling vertices.

The *normals* that are being recomputed are the direction in which faces and their corners are pointing. Normals help with lighting, shading, and shadows. We don't have to worry much about how normals work, but we do need to tell the renderer that we've changed them by telling the shape to `computeFaceNormals` and `computeVertexNormals`. So, after any change to vertices, we have to tell the 3D code to *recompute* where the vertices and faces are.

We can warp shapes and make them look pretty good. This is a river rafting game though. If we pull points down one at a time, it's going to take a long time to make a river in this plane. Let's look next at how we can make a river a little quicker.

Digging a River

When we pulled up `vertices[50]`, we pulled up the vertex at the top-middle of the plane. When we added `100`, `10*100`, and `50*100` to 50, we pulled up vertices in a straight line.

```
var faces = 99;
var shape = new THREE.PlaneGeometry(10, 20, faces, faces);
shape.vertices[50].z = 5;
shape.vertices[50 + 100].z = 5;
shape.vertices[50 + 10*100].z = 5;
shape.vertices[50 + 50*100].z = 5;
```

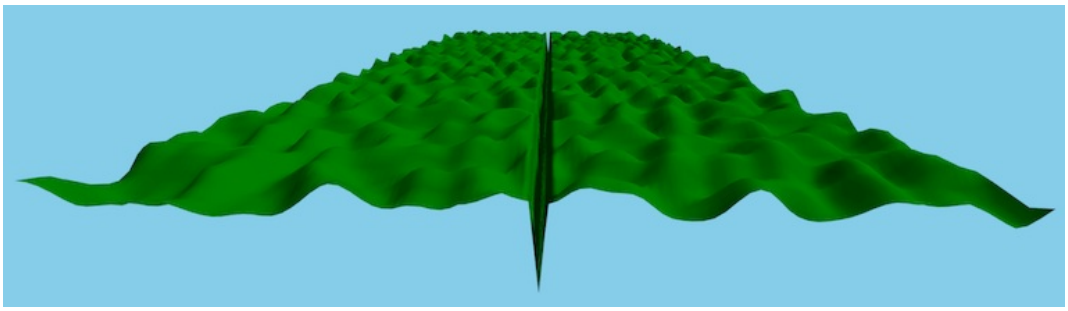
There are 99 rectangles across the top of the plane. That means there are 100 vertices—the 99 top-left corners plus the top-right corner on the end. So every time we move 100, we move to the next row.

We can make use of this in a for loop. Instead of starting the loop at 0, we can start at 50. And, instead of increasing by 1 each time through the loop, we can increase by 100.

Still in the `addGround()` function, add the following code below the noise loop and above the compute methods:

```
for (var j=50; j<numVertices; j+=100) {
  shape.vertices[j].z = -1;
}
```

Programmers are a strange bunch. The first loop in a function usually uses the `i` variable. The second one uses `j` (and the third uses `k`). Since we used `i` to add noise to the ground, we use `j` to find the vertices in the middle of the ground plane. This should pull down each one of those—every 100th vertex—by `-1`. If you click and drag the scene, the ground should now look something like:

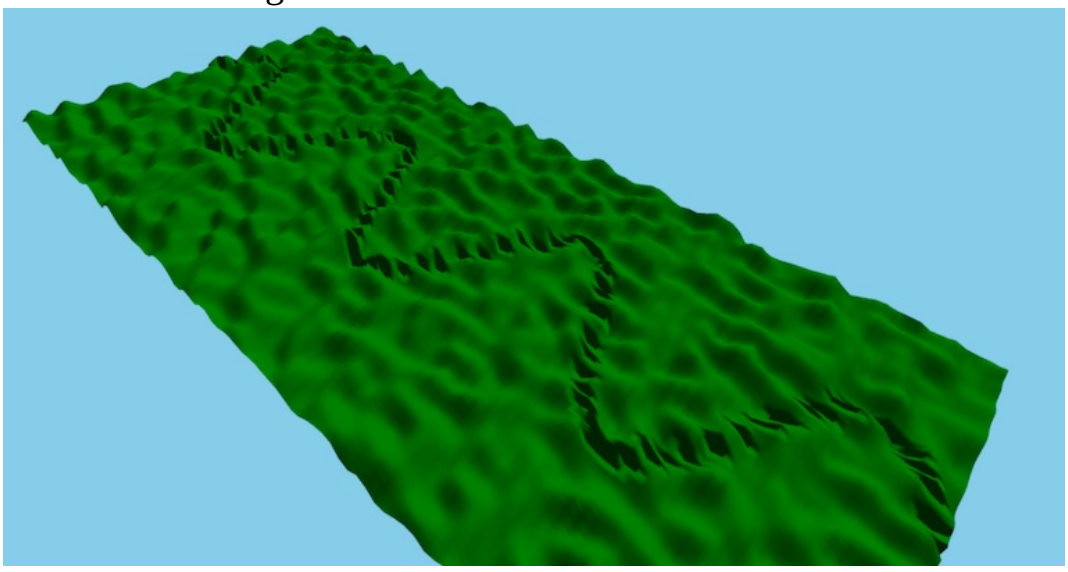


We need to make the river curve back and forth through the ground. We're 3D programmers, so "back and forth" means sines and cosines to us! Let's use a sine function to add a curve. Instead of pulling the river center down at `j`, we'll pull down the river center at `j` plus the result of the sine function.

To do this, change the for loop as follows:

```
  for (var j=50; j<numVertices; j+=100) {
»   var curve = 20 * Math.sin(7*Math.PI * j/numVertices);
»   var riverCenter = j + Math.floor(curve);
»
»   shape.vertices[riverCenter].z = -1;
  }
```

This should change the river to look like this:



The `7*Math.PI` inside the sine function says that the river will wind back and forth a little more than three times. We multiply the `Math.sin()` result by `20` to say how

far from the center the river will wind. The resulting number is the curve of the river at this point in the loop.

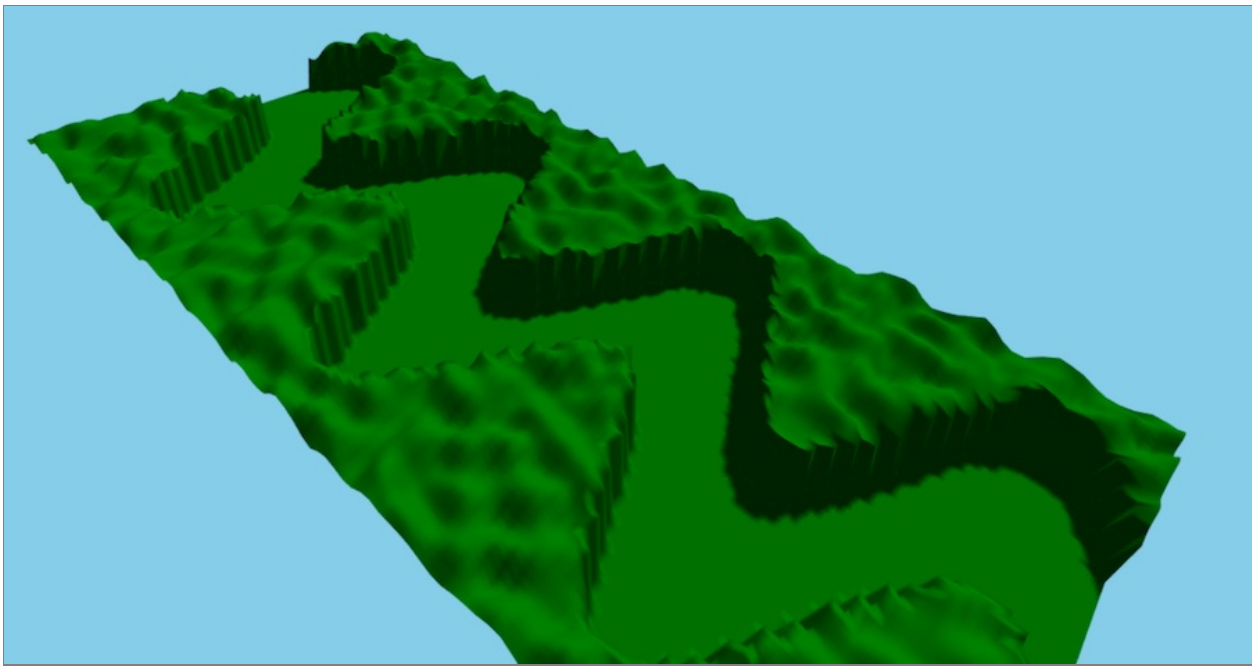
The `vertices` list only works with whole numbers like `50` or `5050`. The value of `curve` will always have a decimal like `50.1443`. So we need a function that removes everything after the decimal point—which is exactly what `Math.floor()` does. If `j` is `1050`, then `curve` might be `14.893`. That would make `riverCenter` be `1050 + Math.floor(14.893)`, or `1064`. So we pull down vertex number 1064. Working through the entire loop like this, we get our nice sine-made, curvy river.

	Let's Play!
	Play with the numbers. Change 20 to 50. Change 7 to 5. What does `20*Math.PI` look like? `2*Math.PI`? These numbers control how big the curves get and how many curves there are. What seems best to you for your game? I think `7*Math.PI` looks best, but it's up to you!

OK, we have a windy river, but it's certainly not big enough for a raft to travel down. Let's make it wider. Inside the loop, for each row, we want to pull down 10 vertices on either side of the center. So, inside the loop, we add *another* loop that starts from `-20` and ends at `20`.

```
  for (var j=50; j<numVertices; j+=100) {
    var curve = 20 * Math.sin(7*Math.PI * j/numVertices);
    var riverCenter = j + Math.floor(curve);

»   for (var k=-20; k<20; k++) {
»     shape.vertices[riverCenter + k].z = -1;
»   }
  }
```

With that, we should have a wide, curvy river.



That’s pretty cool, right?

We have two more things to do with the river. First, we’ll want a way to remember where the center points of the river are.

```
  function addGround() {
    var faces = 99;
    var shape = new THREE.PlaneGeometry(10, 20, faces, faces);

»   var riverPoints = [];
    var numVertices = shape.vertices.length;
    var noiseMaker = new SimplexNoise();
    for (var i=0; i<numVertices; i++) {
      var vertex = shape.vertices[i];
      var noise = 0.25 * noiseMaker.noise(vertex.x, vertex.y);
      vertex.z = noise;
    }

    for (var j=50; j<numVertices; j+=100) {
      var curve = 20 * Math.sin(7*Math.PI * j/numVertices);
      var riverCenter = j + Math.floor(curve);
»     riverPoints.push(shape.vertices[riverCenter]);

      for (var k=-20; k<20; k++) {
        shape.vertices[riverCenter + k].z = -1;
      }
```

```
    }

    shape.computeFaceNormals();
    shape.computeVertexNormals();

    var _cover = new THREE.MeshPhongMaterial({color: 'green', shininess: 0});
    var cover = Physijs.createMaterial(_cover, 0.8, 0.1);

    var mesh = new Physijs.HeightfieldMesh(shape, cover, 0);
    mesh.rotation.set(-0.475 * Math.PI, 0, 0);
    mesh.receiveShadow = true;
    mesh.castShadow = true;
»   mesh.riverPoints = riverPoints;

    scene.add(mesh);
    return mesh;
  }
```

Before we loop through each row, we'll make a list to hold these river center points. Then for each row, we'll push the center vertex onto that list. Finally, after we create the mesh, we'll add that list to the mesh under the `riverPoints` property. This will come in handy later.

Now we just need to add water. At the top of our code, we add a call to the `addWater()` function.

```
  var ground = addGround();
» var water = addWater();
```

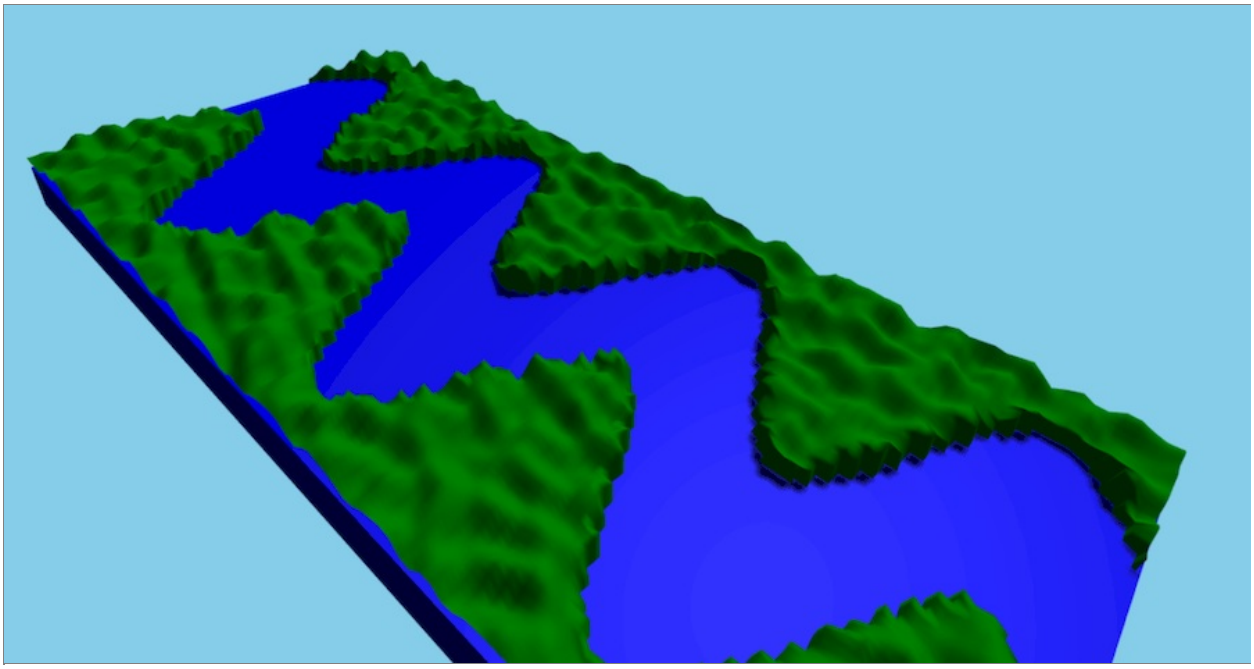
That will break our code, leaving our scene blank, because the `addWater()` function doesn't exist. We'll add that below the last curly brace in `addGround()` and above the `animate()` function.

```
  function addWater() {
    var shape = new THREE.CubeGeometry(10, 20, 1);
    var _cover = new THREE.MeshPhongMaterial({color: 'blue'});
    var cover = Physijs.createMaterial(_cover, 0, 0.6);

    var mesh = new Physijs.ConvexMesh(shape, cover, 0);
    mesh.rotation.set(-0.475 * Math.PI, 0, 0);
    mesh.position.y = -0.8;
    mesh.receiveShadow = true;
```

```
    scene.add(mesh);

    return mesh;
  }
```

There’s nothing new in there. We create a physics cover for the water just like we did with the ground, giving it 0 friction (very slippery) and `0.6` bounciness.

With that, we have water in our river!



That was a lot of work. And it was very different from what we’ve seen. The rest of the game should be a little more familiar. Next, we add a way to keep score.

Scoreboard

We'll want to keep score in this game so, in the code outline at the top of our code, add a call to the `addScoreboard()` function.

```
  var ground = addGround();
  var water = addWater();
» var scoreboard = addScoreboard();
```

But there's no `addScoreboard()` function yet, so this temporarily breaks things. To quickly get everything back, add the function after the last curly brace in the `addRiver()` function.

```
function addScoreboard() {
  var scoreboard = new Scoreboard();
  scoreboard.score(0);
  scoreboard.timer();
  scoreboard.help(
    'left / right arrow keys to turn. ' +
    'space bar to move forward. ' +
    'R to restart.'
  );
  return scoreboard;
}
```

That creates the same kind of scoreboard that we've been using throughout the book. It keeps score, counts the time, and gives a little help.

Now that we have a river and a scoreboard, let's add the raft to the game.

Build a Raft for Racing

A donut shape will work very nicely as a river raft. Add the `addRaft` call to the code outline at the top of our code.

```
  var ground = addGround();
  var water = addWater();
  var scoreboard = addScoreboard();
» var raft = addRaft();
```

Again, this breaks things because there's no `addRiver()` function yet. Let's add it after the last curly brace in the `addScoreboard()` function.

```
function addRaft() {
  var shape = new THREE.TorusGeometry(0.1, 0.05, 8, 20);
  var _cover = new THREE.MeshPhongMaterial({visible: false});
  var cover = Physijs.createMaterial(_cover, 0.4, 0.6);
  var mesh = new Physijs.ConvexMesh(shape, cover, 0.25);
  mesh.rotation.x = -Math.PI/2;

  cover = new THREE.MeshPhongMaterial({color: 'orange'});
  var tube = new THREE.Mesh(shape, cover);
  tube.position.z = -0.08;
  tube.castShadow = true;
  mesh.add(tube);
  mesh.tube = tube;

  shape = new THREE.SphereGeometry(0.02);
  cover = new THREE.MeshBasicMaterial({color: 'white'});
  var rudder = new THREE.Mesh(shape, cover);
  rudder.position.set(0.15, 0, 0);
  tube.add(rudder);

  scene.add(mesh);
  mesh.setAngularFactor(new THREE.Vector3(0, 0, 0));
  return mesh;
}
```

That's a large function, but it's all stuff we've seen before. We create the physics-enabled mesh for the raft, making it a little slippery (`0.4`) and a little bouncy (`0.6`). We make the mesh a little heavy (`0.25`). Finally, we rotate it flat.

The mesh itself will be invisible. We'll use it as a marker like we did with the avatar back in Chapter 8, *Project: Turning Our Avatar*. To see the raft, we create an orange tube, which gets added to the mesh. We also add a "rudder" so we know which direction the raft is facing. After adding all of this to the scene, we set the angular factor of the mesh so it can't spin. We'll spin the tube inside the mesh, but the mesh itself should always face the same direction.

The raft has been added to the scene at this point, but it's not at the start of the river. We'll change that in the next part of our code.

Resetting the Game

The ground, water, scoreboard, and raft are the most important pieces in our game. So after each is added, let's reset the game to a good starting point.

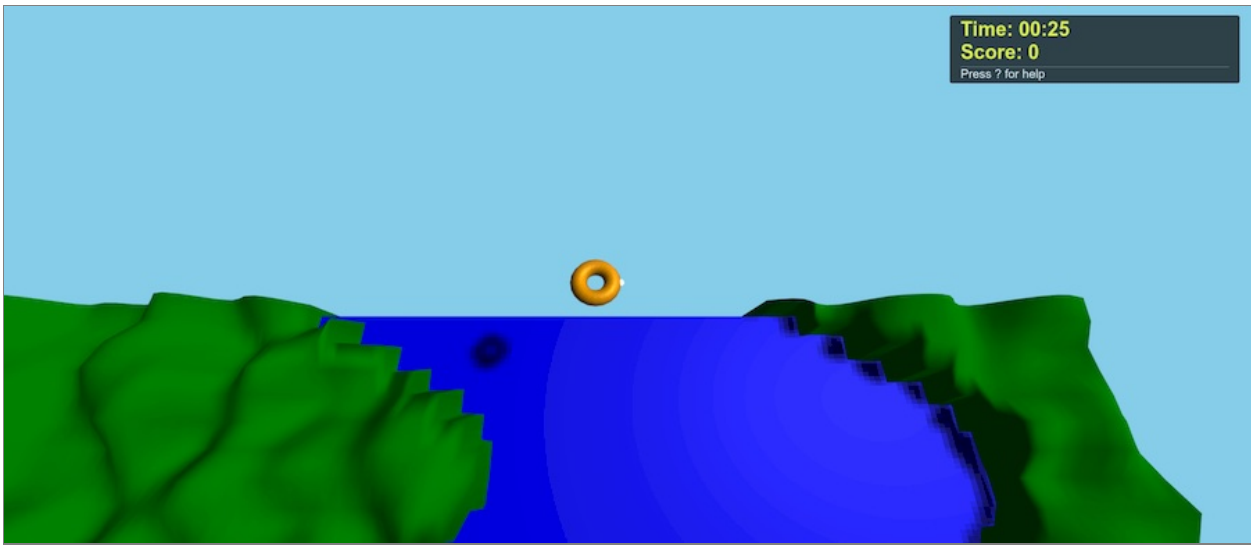
```
var ground = addGround();
var water = addWater();
var scoreboard = addScoreboard();
var raft = addRaft();
» reset();
```

Below `addRaft`, we add the `reset()` function.

```
function reset() {
  camera.position.set(0,-1,2);
  camera.lookAt(new THREE.Vector3(0, 0, 0));
  raft.add(camera);

  scoreboard.message('');
  scoreboard.resetTimer();
  scoreboard.score(0);

  raft.__dirtyPosition = true;
  raft.position.set(0.75, 2, -9.6);
  raft.setLinearVelocity(new THREE.Vector3(0, 0, 0));
}
```

Don't forget that `__dirtyPosition` starts with two underscore characters!

With that, the game should look like this when it starts:



We move the camera very close to the scene, then add it to the raft. We reset the score and the timer in the scoreboard. Finally, we move the raft to the starting line: a little left because of the curvy river, a little up because of the tilt that we added to the river, and back because that is where the edge of the river is.

The code in this function has to work for both starting the game and restarting the game. The `setLinearVelocity` call sets the speed of the raft to 0 every time it's called. Without that, a player who restarted the game midway through a race would restart at the starting line already at full speed.

At this point you should have a raft moving down the river, with the camera watching it the whole way. Of course, this is pretty useless without controls, so let's add some.

Keyboard Controls

We're ready to add keyboard controls to the scene, but first we need to remove the orbit controls. If we leave the orbit controls in, arrow key presses would move the raft *and* the camera, which would be very confusing. So either delete or comment out the orbit controls above the `START CODING` line.

```
// new THREE.OrbitControls(camera, renderer.domElement);
```

Now, let's add our game controls. Add them at the very bottom of your code, just above the final `</script>` tag.

```
document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;
  if (code == 'ArrowLeft') rotateRaft(1);
  if (code == 'ArrowRight') rotateRaft(-1);
  if (code == 'ArrowDown') pushRaft();
  if (code == 'Space') pushRaft();
  if (code == 'KeyR') reset();
}

function rotateRaft(direction) {
  raft.tube.rotation.z = raft.tube.rotation.z + direction * Math.PI/10;
}

function pushRaft() {
  var angle = raft.tube.rotation.z;
  var force = new THREE.Vector3(Math.cos(angle), 0, -Math.sin(angle));

  raft.applyCentralForce(force);
}
```

We've seen keyboard listeners a lot by this point, so `document.addEventListener` should already be familiar.

The `rotateRaft()` function doesn't have to worry about breaking physics because we turn the non-physics enabled tube. Remember we made the invisible mesh physics enabled and then placed the non-physics tube inside the mesh.

Finally, the `pushRaft` function uses `applyCentralForce` to push the raft, just like we pushed the game balls in Chapter 17, *Project: Ready, Steady, Launch*.

With that, we have the basic pieces of a pretty cool game! The left and right arrow keys will turn the raft, and the `space bar` will push the raft forward in the direction it's facing.

We can do a *lot* more with this game. We'll start with a finish line and then move on to some optional ways to add scoring to the game.

The Finish Line

Eventually the raft reaches the finish line. And then it falls off the edge of the river and keeps right on going. And going. Instead, let's pause the game so players can take a moment to admire their score before trying again. We need to make changes in four places: in our code outline and in the `reset`, `animate`, and `gameStep` functions.

Let's start with the code outline. Before the call to the `reset` function, we need to add a line for the `gameOver` variable.

```
» var gameOver;
  var ground = addGround();
  var water = addWater();
  var scoreboard = addScoreboard();
  var raft = addRaft();
  reset();
```

Other functions will use that variable to decide whether they need to animate or update the game. JavaScript is pretty uptight about when variables are declared. The rule of thumb is that variables need to be declared before they're used. The `gameOver` variable will be used in `reset`, `animate`, and `gameStep`, so we declare it before any of them are called.

We set `gameOver` for the first time at the end of the `reset` function. Whenever the game is started, which is what `reset` does, the game isn't over. So we set `gameOver` to `false`.

```
  function reset() {
    camera.position.set(0,-1,2);
    camera.lookAt(new THREE.Vector3(0, 0, 0));
    raft.add(camera);

    scoreboard.message('');
    scoreboard.resetTimer();
    scoreboard.score(0);

    raft.__dirtyPosition = true;
```

```
    raft.position.set(0.75, 2, -9.6);
    raft.setLinearVelocity(new THREE.Vector3(0, 0, 0));

»   gameOver = false;
»   animate();
»   scene.onSimulationResume();
»   gameStep();
  }
```

After restarting the game, we start the animation and start the game logic by calling `animate()` and `gameStep()`. We also call the `onSimulationResume()` method on the scene, which resets the physics code anytime it resumes after being paused.

Next we tell the `animate` function that it doesn't have to render the scene when the game is `gameOver`. That is, if `gameOver` is set to `true`, then we exit the `animate` function before updating the camera or rendering the scene.

```
  function animate() {
»   if (gameOver) return;
    requestAnimationFrame(animate);
    renderer.render(scene, camera);
  }
» // animate();
```

And since we started `animate()` inside the `reset()` function, we no longer need to call it here. So comment it out or delete it.

We do something similar in the `gameStep` function. If the game is over, then we exit immediately from the function without completing any of the usual steps. If the game has not ended yet, we then check to see whether it should.

```
  function gameStep() {
»   if (gameOver) return;
»   checkForGameOver();
    scene.simulate();
    // Update physics 60 times a second so that motion is smooth
    setTimeout(gameStep, 1000/60);
  }
» // gameStep();
```

Again, since we started `gameStep()` inside of the `reset()` function, we comment out

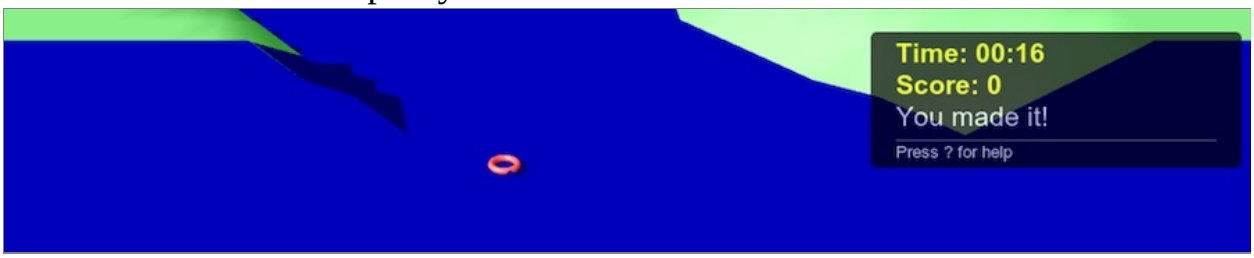
or delete the call to `gameStep()` after the function definition.

The `checkForGameOver` function is new. It can go right after the `gameStep` function.

```
function checkForGameOver() {
  if (raft.position.z > 9.8) {
    gameOver = true;
    scoreboard.stopTimer();
    scoreboard.message("You made it!");
  }

  if (scoreboard.getTime() > 60) {
    gameOver = true;
    scoreboard.stopTimer();
    scoreboard.message("Time's up. Too slow :(");
  }
}
```

We check to see whether the raft's Z position is greater than 9.8, which is very close to the edge of the river at 10.0. If the raft has reached 9.8, we set `gameOver` to `true` and update the scoreboard. We also add a check for slowpokes here.

With that, the game should pause at the end of the river and display the time it took the player to complete the race and a message, "You made it!" You might even be able to make it pretty fast:



Bonus: Keeping Score

This is already a pretty impressive game. It looks cool and it's challenging. So this is a fine stopping point. But if you'd like to add a bit more, here are some bonus features to add.

Time-Based Scoring

Let's calculate the score like this: you get more points the quicker you finish, and you get bonus points if you can finish really quick. We can do this by adding another check in `checkForGameOver()`.

```
  function checkForGameOver() {
    if (raft.position.z > 9.8) {
      gameOver = true;
      scoreboard.stopTimer();
      scoreboard.message("You made it!");
    }

    if (scoreboard.getTime() > 60) {
      gameOver = true;
      scoreboard.stopTimer();
      scoreboard.message("Time's up. Too slow :(");
    }

»   if (gameOver) {
»     var score = Math.floor(61-scoreboard.getTime());
»     scoreboard.addPoints(score);
»
»     if (scoreboard.getTime() < 40) scoreboard.addPoints(100);
»     if (scoreboard.getTime() < 30) scoreboard.addPoints(200);
»     if (scoreboard.getTime() < 20) scoreboard.addPoints(500);
»   }
  }
```

We check whether the game is over. If either raft made it to 9.8 or time expired, then `gameOver` is set to `true`. When that happens, it's time to tally up the score.

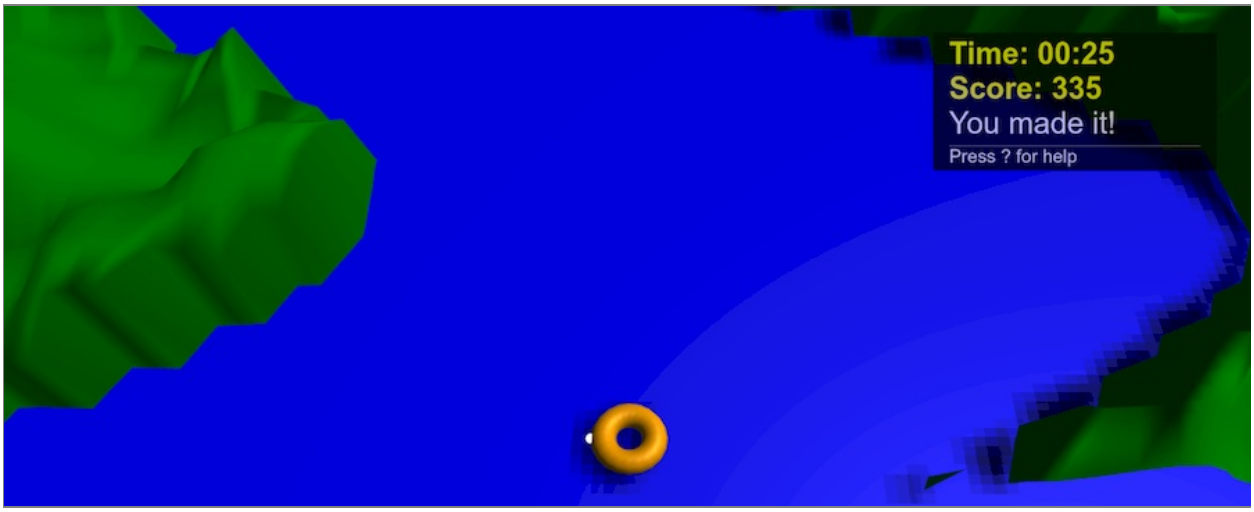
First, we subtract the time from 61. We "floor" that because we only want whole numbers. For example, if I finish in 29.9 seconds, then the score should be

`Math.floor(61 - 29.9)`, which is the same as `Math.floor(31.1)`, which is 31 points.

Next, we add bonus points:

- If the player finishes in less than 40 seconds, an additional 100 points are awarded.
- If the player finishes in less than 30 seconds, then both the 100 points and 200 more are awarded.
- If the player finishes in less than 20 seconds, then the 100 and the 200 points are awarded, along with an additional 500 points, for a possible total of 800 extra points to be won.

Can you do it? See the figure.



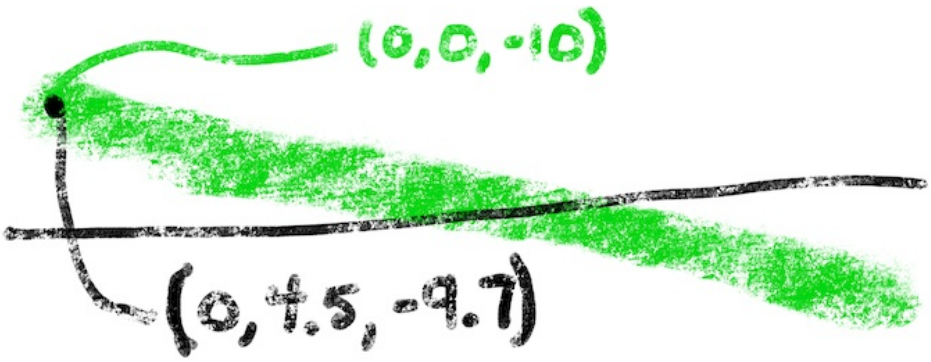
Power-Up Points

Scoring bonus points is much harder to add to this game than you might guess. Adding power-up fruit to the river or ground won't work. Instead, we have to add power-up fruit to the scene, but that's not easy because the river is tilted.

If we add the power-up fruit to the river, collision detection stops working for the fruit mesh. As far as the physics code is concerned, when we add the fruit to the river, the fruit is no longer a separate mesh. It's now part of the river—and

that's a problem. Listening for events on the fruit won't work since there won't be any events. We could listen for events on the river, but it would be really hard to tell the difference between colliding with the river and the fruit part of the river.

We need to add the fruit directly to the scene. That way, it stays as a separate mesh with its own collision events. But what coordinates do we use? Remember that the river is tilted. Looking from the side, we can see that the ground thinks that the top middle coordinate is at (0, 0, -10), but the scene thinks that coordinate is at (0, 4.5, -9.7).



We know where the center of the river is along the plane thanks to the `riverPoints` property that we added to the `ground`. But we need to convert those from ground coordinates to scene coordinates.

The math to do that is not too hard—it's just more sines and cosines, but we don't have to do that math ourselves. Instead, we'll have our 3D code translate from "local" river coordinates to "world" scene coordinates.

We'll want to add items to the game whenever it gets reset, so much of our work needs to take place in and after the `reset` function. Inside the definition of `reset`, add a call to `resetPowerUps`:

```
  function reset() {
»   resetPowerUps();


    camera.position.set(0,-1,2);
    camera.lookAt(new THREE.Vector3(0, 0, 0));
```

```
  raft.add(camera);

  scoreboard.message('');
  scoreboard.resetTimer();
  scoreboard.score(0);

  raft.__dirtyPosition = true;
  raft.position.set(0.75, 2, -9.6);
  raft.setLinearVelocity(new THREE.Vector3(0, 0, 0));

  gameOver = false;
  animate();
  scene.onSimulationResume();
  gameStep();
}
```

This will break the scene since we haven't defined `resetPowerUps()` yet. Let's do that next. Add it just below the closing curly brace for the `reset()` function.

```
function resetPowerUps() {
  var random20 = 20 + Math.floor(10*Math.random());
  var p20 = ground.riverPoints[random20];
  addPowerUp(p20);

  var random70 = 70 + Math.floor(10*Math.random());
  var p70 = ground.riverPoints[random70];
  addPowerUp(p70);
}
```

This will also break because `resetPowerUps()` is calling another new function, `addPowerUp()`, twice. We'll add that in a moment, but first, let's make sure we understand what's going on inside `resetPowerUps()`.

In there, we create two random numbers and use them to get two points along the river. There are 100 total river points. We want to put some fruit around the 20th and 70th river points. The location should be different in each game to keep the game challenging. The `riverPoints` property is a list, so we find a point with a whole number—just like we did with the `vertices` property when digging the river. We again use `Math.floor()` to get a random whole number. In this way, `p20` and `p70` will be random river points near river point 20 (near the beginning) and

point 70 (near the end).

We pass those random river points to the new `addPowerUp()` function. Add it just below `resetPowerUps()`.

```
function addPowerUp(riverPoint) {
  ground.updateMatrixWorld();
  var x = riverPoint.x + 4 * (Math.random() - 0.5);
  var y = riverPoint.y;
  var z = -0.5;
  var p = new THREE.Vector3(x, y, z);
  ground.localToWorld(p);

  var shape = new THREE.SphereGeometry(0.25, 25, 18);
  var cover = new THREE.MeshNormalMaterial();
  var mesh = new Physijs.SphereMesh(shape, cover, 0);
  mesh.position.copy(p);
  mesh.powerUp = true;
  scene.add(mesh);

  mesh.addEventListener('collision', function() {
    for (var i=0; i<scene.children.length; i++) {
      var obj = scene.children[i];
      if (obj == mesh) scene.remove(obj);
    }
    scoreboard.addPoints(200);
    scoreboard.message('Yum!');
    setTimeout(function() {scoreboard.clearMessage();}, 5*1000);
  });

  return mesh;
}
```

We've seen most of this before—except for the very beginning of the function. That's definitely new, but it does just what we talked about earlier. It converts an X-Y-Z location from a point along the ground into a scene coordinate. In 3D programming terms, we convert from local to world coordinates.

Before doing that, calling `updateMatrixWorld` is a good idea. This ensures that the various coordinate values are all up to date and accurate. After this, we copy the X and Y coordinates from the river point. We add a random number to the X

coordinate to move the fruit to the side of the river, making things more challenging. We use -0.5 for the Z coordinate to put the fruit in the water.

Once we have `p`, we ask `ground` to convert it from local ground coordinates to world scene coordinates with the `localToWorld()` method. After that, we're back in familiar territory.

We create a sphere shape and a cover, and use them to build a physics-enabled mesh. This mesh won't move, so it gets a mass of `0`. Then we set the position of the mesh to the same as the `p` coordinates using the `copy()` method, copying `p`'s coordinates to the mesh's position. Since those coordinates are scene coordinates, we can add the mesh directly to the scene.

Two other things to note are the `powerUp` property and the collision detection. The `powerUp` property will be used to remove power-up items from the scene when the game resets. We also add a collision event handler. When the raft collides with a power-up fruit, it will remove the fruit from the scene and add points to the scoreboard.

We use functions without names here. We used unnamed functions for creating methods in Chapter 16, *Learning about JavaScript Objects*. Using them directly inside event handlers like this can get messy, but it works. When a collision is detected, the code inside the unnamed `function()` is run—removing the object from the scene and adding points to the score. We also set a "Yum!" message on the scoreboard, clearing it five seconds later—with another unnamed function.

One last thing that we need to do is clean up power-up fruit when the game resets. We added the `powerUp` property to the power-up fruit for this reason. Whenever the `reset()` function calls the `resetPowerUps()` function that we just added, we'll look through the scene's "children" for objects with this property.

Add a call to `removeOldPowerUps()` at the beginning of `resetPowerUps()`.

```
  function resetPowerUps() {
»   removeOldPowerUps();
```

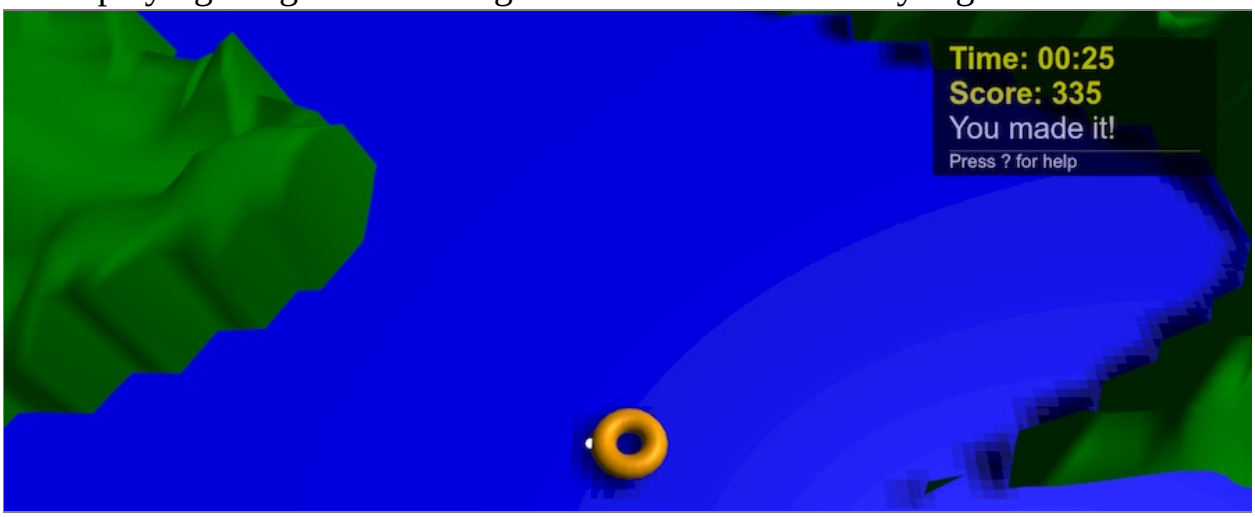
```
  var random20 = 20 + Math.floor(10*Math.random());
  var p20 = ground.riverPoints[random20];
  addPowerUp(p20);

  var random70 = 70 + Math.floor(10*Math.random());
  var p70 = ground.riverPoints[random70];
  addPowerUp(p70);
}
```

Then add the `removeOldPowerUps()` function below the last curly brace of `addPowerUp()`.

```
function removeOldPowerUps() {
  var last = scene.children.length - 1;
  for (var i=last; i>=0; i--) {
    var obj = scene.children[i];
    if (obj.powerUp) scene.remove(obj);
  }
}
```

This is the same thing we did in Chapter 14, *Project: The Purple Fruit Monster Game*, starting with the last *child* object in the scene and moving to the first so that we don't skip objects.

With that, we have two pieces of fruit that can help you score points like crazy while playing the game. You might even be able to beat my high score:



The Code So Far

If you'd like to double-check the code in this chapter, turn to *Code: River Rafter*.

What's Next

That was a lot of code. But it was worth it. We put many of our skills to use with physics, lights, and materials. We also saw glimpses of what else is possible in 3D programming by pulling vertices of shapes and converting local coordinates to *world* coordinates.

Like our other games, do not consider this one final. There's still plenty you can add. Maybe you can incorporate obstacles that take away points? Add some jumps? Make this a two-player game as we did in Chapter 18, *Project: Two-Player Games*? Make the course longer? You might try adding camera controls so you can see from the viewpoint of a raft passenger instead of viewing everything from above. Maybe the game would look better with texture images? Could you make sounds when the raft powers up or finishes? Or maybe you already have ideas for how to improve this game that I can't even begin to think of.

In our final chapter, we'll switch gears to putting our projects out on the web.

When you're done with this chapter, you will

- *Have a better idea of the parts that make up a website*
- *Understand what we need to build our own websites*
- *Know how to put a project on a site to share with others*

Chapter 20

Getting Code on the Web

JavaScript is the language of the internet. Every web or mobile site that you visit uses JavaScript in one way or another. So in our last chapter, let’s take a brief look at how internet sites work and how they rely on JavaScript.

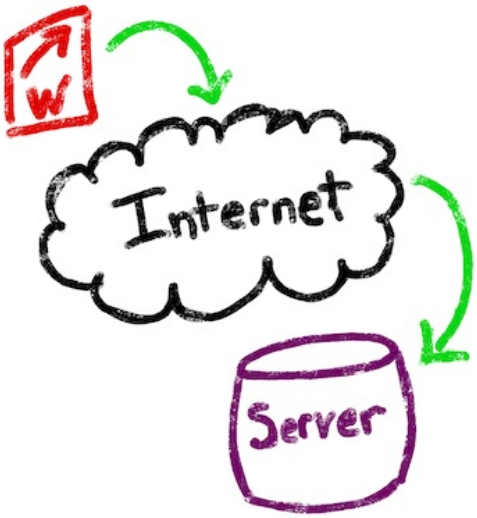
We won’t go into tons of detail here—just enough to get started making your own JavaScript-powered pages. The easiest way to do that will be to put one of our own projects on a website. That will require a few changes to the code that we’ve been writing, but compared to everything else we’ve covered in this book, this is going to be easy.

First, a quick overview of how web and mobile browsers work.

The Mighty, Mighty Browser

Behold the mighty browser:

When we tell a browser that we want a website or a page on a website, it sends a request through the internet to a machine that we call a *web server* as shown in the figure.



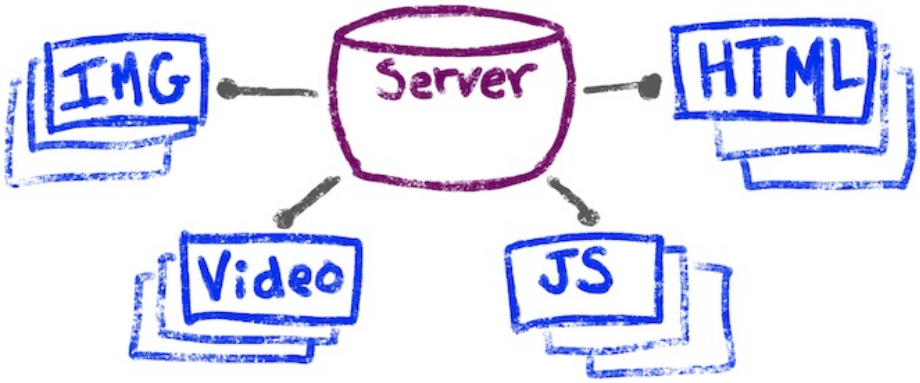
To reach the correct server, our browser has to look up the network address of the web server on the internet. The browser uses the Domain Name Service (DNS) to perform that lookup. When we visit http://www.code3Dgames.com, DNS replies with a network address of `151.101.1.195`. That network address—sometimes called an *Internet Protocol* or *IP* address—is all that a browser needs

to send a web request across the internet.

	Web Servers Must Be Publicly Available on the Internet
	Remember that the machine holding a website must be publicly available on the internet. The machines that you use at home are almost never publicly available. Even if someone else on the internet knows your machine's network address, they would still not be able to reach it because it's not publicly available—it's hidden behind your network. Unfortunately, this means you usually need to pay a little money to get your cool web games available on the internet. You need to pay a web hosting company to host your games. And you'll also have to pay for the DNS that maps a site name (like www.code3Dgames.com) to a network address (like 151.101.1.195). There are free options though, as we'll see in a bit.

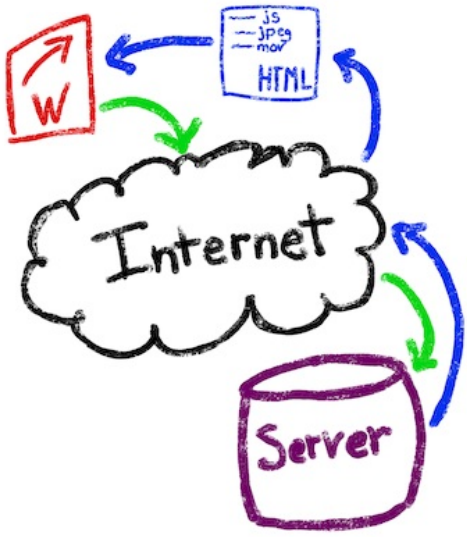
A web request asks one particular web server for one particular piece of information that it has. That information might be a web page, it might be an image, it might be a movie, and it might be JavaScript. But it's only for one thing.

When the browser's request reaches the web server, the server checks to see whether it has the requested item. Web servers can have all sorts of information stored on them as shown in the figure.



The first request that a browser sends to a server is usually for a web page—a file written in HTML, which we talked about in Chapter 9, *What's All That Other Code?*.

If the server has the web page the user's looking for, then it sends it back to the browser.



This is the kind of weird part. The web page is usually pretty small and uninteresting. When we think of web pages, we usually think of something that

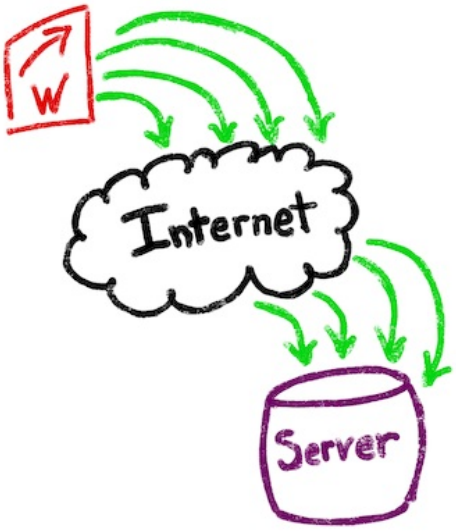
looks pretty and does lots of amazing things. But by themselves, web pages don't look pretty and don't do anything. Web pages look pretty much like the HTML that we saw in Chapter 9, *What's All That Other Code?*.

```
<body>
  <h1>Hello!</h1>
  <p>
    You can make <b>bold</b> words,
    <i>italic</i> words,
    even <u>underlined</u> words.
  </p>
  <p>
    You can link to
    <a href="http://code3Dgames.com">other pages</a>.
    You can also add images from web servers:
    <img src="/images/purple_fruit_monster.png">
  </p>
</body>
```

For anything fun to happen, the web page needs to tell the browser to make lots and lots of other requests. This is done with HTML tags and JavaScript.

Some of the tags in HTML are for paragraphs or formatting words. Other tags, like **<img>**, load things like images or styles that make cool stuff happen. Then there are the **<script>** tags that can load JavaScript code collections or let us write JavaScript directly on the web page.

And so, as soon as the browser gets the web page that it asked for, it has to ask for tons and tons more things that are described in those tags.



Since all of those things take time, we have to be careful about when we run our JavaScript. We don't want to try starting our JavaScript code before the code collections and images have finished loading.

The easiest way to wait for everything to load is what we've done throughout this book: put our JavaScript below everything else in the web page.

```
<body></body>
<script>
  // This is where we have been coding - after the <body> tags
</script>
```

The `<script>` tag is at the very bottom of the document, meaning that the browser will display the web page stuff (text, images, style information) before it runs our code. Everything else should be ready by the time the browser runs that code.

As you work on more and more JavaScript sites, you'll come across other solutions to this. Which solution you use doesn't matter too much. Just keep in mind that a lot more has to load on a web page than just the web page—and your code needs to handle that. Otherwise really strange bugs might creep into your pages and applications.

Free Websites

Earlier we noted that only publicly available web servers can serve up web pages, images, JavaScript, and so on. Normally this will cost you some money. But there are ways to make your web pages and JavaScript games publicly available for free. One of the easiest is Blogger.[8]

Many free sites will strip out your JavaScript or make it hard to get your JavaScript on their site. If you're looking for a free site to use, be sure that it supports adding your own JavaScript. Sites should support putting JavaScript inside `<script>` tags—like we've been doing throughout the book. Sites should also let us load code collections using `<script>` tags with `src` attributes. Happily, Blogger lets us do both.

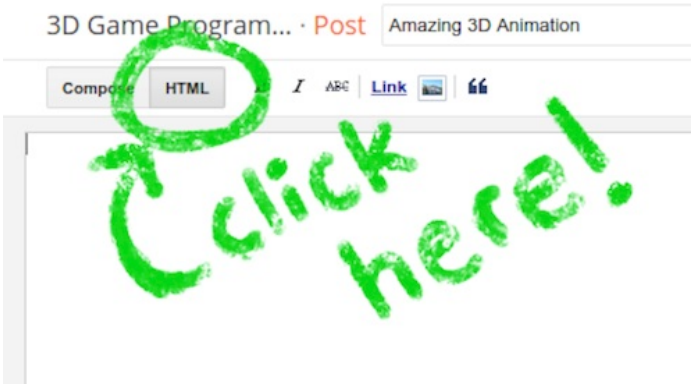
If you're sharing simple games with friends and family, a free site is the best option. It's easy to set up and use. You only need to pay for a site for more complex games—the kinds that need user accounts and have to save scores or other game information.

To get an idea of how to copy some of our projects onto a real website, we'll take a look at putting one of our 3D animations on Blogger.

Putting Your Code on Another Site

Before you work on this section, you'll need to create a Blogger account (or an account on a similar service). The instructions below should work for most sites, but all posting services have their own quirks that you may have to debug on your own. Also note that the Blogger controls can change over time and may not look or work exactly as shown here.

Posting our code to Blogger is pretty easy. Start a post like normal, but be sure to click the HTML button in the Post toolbar:



In the text area below the HTML button, add some HTML. The following creates a short paragraph, followed by a spot for your 3D game, followed by another short paragraph:

```
<p>I made this!</p>
<div id="3d-code-2018-12-31">
</div>
<p>
  It's in the first chapter of
  <a href="http://code3Dgames.com/">3D Game Programming for Kids</a>,
  second edition.
</p>
```

You can change the words inside the `<p>` tags to whatever you like. The `<div>` tag that we use here is an empty divider. We'll use that to hold our game.

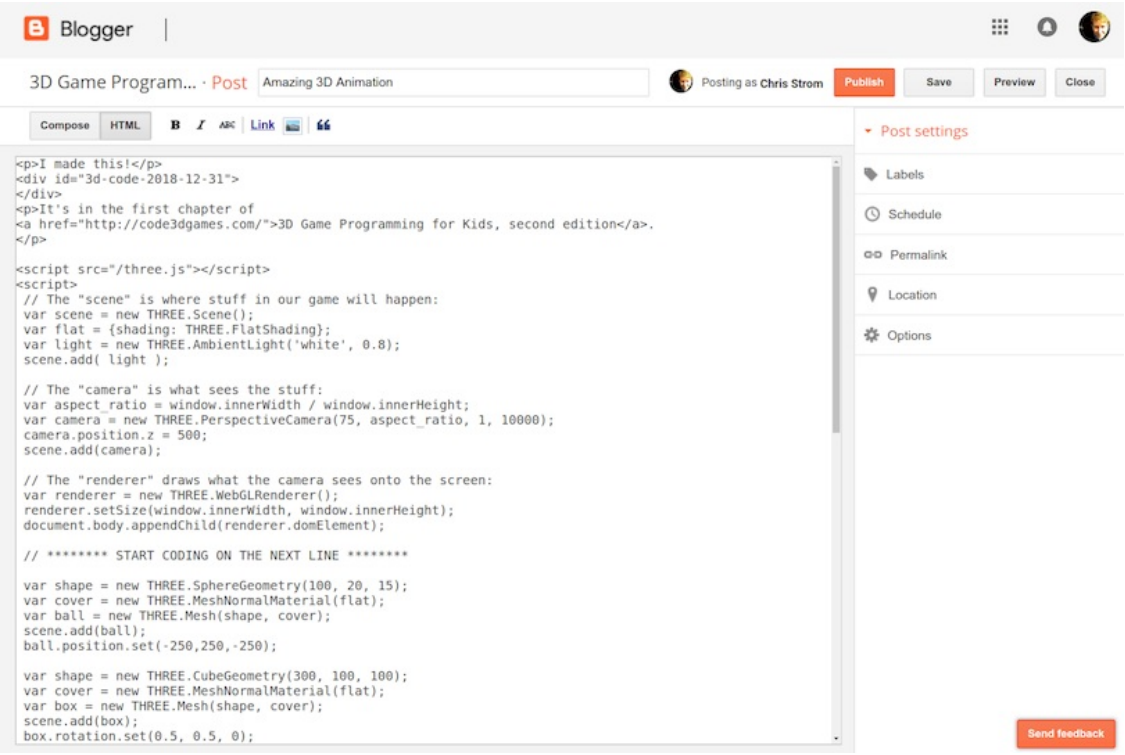
It's important that the `id` attribute for the `<div>` be unique—that there are no other

tags with the same `id` anywhere on the page. One way to do that is combining the purpose (`3d-code`) and the date you post the code (`2018-12-31`, for example).

Next, copy your code from 3DE and paste it into the Blogger post. For your first post, it's best to keep it simple, so we'll use the code from all the way back in Chapter 1, *Project: Creating Simple Shapes*.

When copying code from 3DE, *be sure to skip the first line* that contains `<body>` `<body>`. Only copy from the first `<script>` tag to the end.

Paste this into the Blogger post below the HTML that we added earlier as shown in the figure.



Before we click the Publish button, we need to make a few changes.

First, we have to tell Blogger where to find the code collections that we've been using. Throughout the book, our script tags have pointed to `src` code collections that start with a slash.

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

This tells the mighty web browser to find the code collection on the current web server. The web server for all of our code has been `www.code3Dgames.com`, since 3DE lives at `https://www.code3Dgames.com/3de`. But now that we're making a page on Blogger, we have to tell the Blogger pages to look for the code collections on `www.code3Dgames.com` instead of Blogger. To do that, just add `https://www.code3Dgames.com` at the start of the `src` values of the `<script>` tags.

```
<script src="https://code3Dgames.com/three.js"></script>
<script src="https://code3Dgames.com/controls/OrbitControls.js"></script>
```

Note that we also added the orbit controls here—mostly because they're fun!

Next, we need to set the aspect ratio of the scene. In Chapter 9, *What's All That Other Code?*, we saw that aspect ratio describes how wide and tall a scene is. On a web page, 4/3 is a good aspect ratio, so we change that as follows:

```
» var aspectRatio = 4/3;
  var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
  camera.position.z = 500;
  scene.add(camera);
```

From Chapter 9, *What's All That Other Code?*, we know that we usually add the renderer to the whole page. For our post, we want to attach it to the `<div>` that we added earlier. This requires two changes to the renderer code.

```
  // The "renderer" draws what the camera sees onto the screen:
  var renderer = new THREE.WebGLRenderer({antialias: true});
» // renderer.setSize(window.innerWidth, window.innerHeight);
» // document.body.appendChild(renderer.domElement);
» var container = document.getElementById('3d-code-2018-12-31');
» container.appendChild(renderer.domElement);
```

First, we no longer want to set the size of the renderer to be the same as the whole window. We'll set that in a bit, but for now, either delete the `renderer.setSize()` line or comment it out.

The other renderer change is to add it to the `<div>` tag. We'll want that `<div>` element to "contain" the animation, so we name the variable `container`. We "get" the `<div>` in JavaScript using the ID we set in the HTML. If you used `3d-code-`

2018-12-31 for the id of the tag earlier, then get it using document.getElementById('3d-code-2018-12-31'). Then we add the renderer to the container using the same appendChild() method we talked about in Chapter 9, *What's All That Other Code?*.

We want the renderer to fit inside the container. We'll use a function to do that. Add this function just below the line that appends the renderer to the container.

```
function resizeRenderer(){
  var width = container.clientWidth * 0.96;
  var height = width/aspectRatio;
  renderer.setSize(width, height);
}
resizeRenderer();
window.addEventListener('resize', resizeRenderer, false);
```

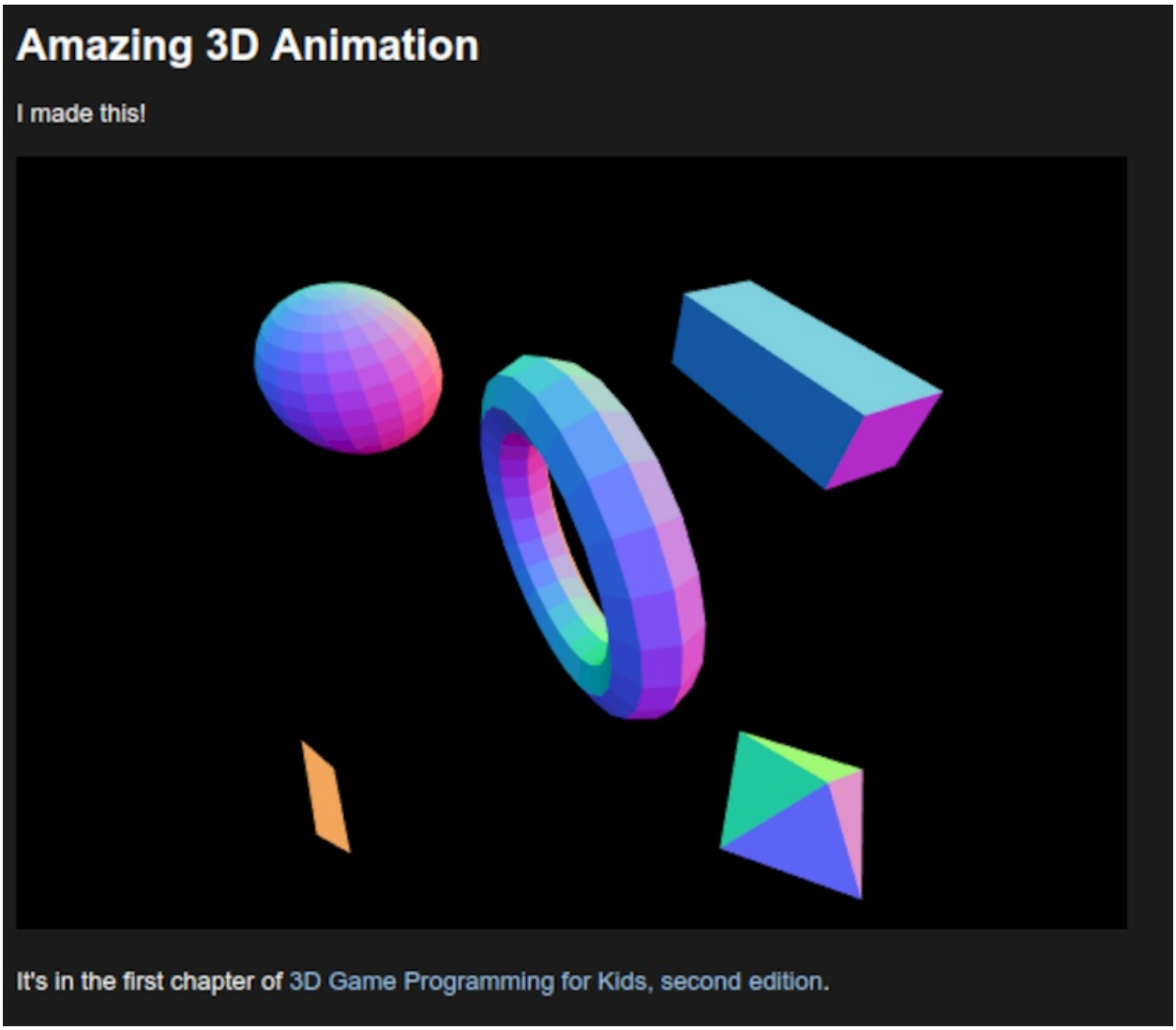
The resizeRenderer() function gets the width of the container from its clientWidth property. We shrink the width just a tiny bit because that helps with some browsers. We get the correct height by dividing the width by the aspect ratio. Last, we set the renderer size to those width and height values.

We call the resizeRenderer() function right away to get the size correct when the page first loads. Then we do something a little interesting: we add an event listener to the window. This listens for the "resize" event, and calls the resizeRenderer() function. That way, every time a user looking at our page resizes their browser, the renderer will update itself with the correct size.

The last bit of setup we'll do is adding orbit controls. We want our friends to be able to move the camera around the scene to get a better look at our amazing creation. We already added a <script> tag to pull in the code collection for the orbit controls, so we can create the controls now, just above the START CODING line:

```
new THREE.OrbitControls(camera, renderer.domElement);
```

With that, you should be able to publish your post and see your handiwork:



Be sure to share your work on the book forum![9]

The Code So Far

To see the complete code for the post created for this chapter turn to *Code: Getting Code on the Web*.[10]

What’s Next

You’re on your own now! I’ve taught you all I can, which means it’s time for you to let your creativity loose. None of the games in this book are meant to be finished products. Each and every game can be made better, but only if you add code and enhance gameplay. You’ve learned a ton just by making it to the end of the book. Now things get interesting—you find out what you can do by yourself. That’s the most exciting adventure I can imagine for you.

Good luck! And if you need some help, don’t forget to ask in the book forum. I can’t wait to hear about the amazing things you create!

Footnotes

[8] http://blogger.com

[9] https://talk.code3Dgames.com/

[10] http://code3Dgames.blogspot.com/2018/02/amazing-3d-animation.html

Appendix 1

Project Code

This appendix contains completed versions of all of the projects created in this book. Your code may not be exactly the same as the code that follows—that's OK. The code is included in case you run into problems and want to be able to compare your code to a working version.

Code: Creating Simple Shapes

This is the final version of the shapes code from Chapter 1, *Project: Creating Simple Shapes*:

```
<body></body>
<script src="/three.js"></script>
<script>
 // The "scene" is where stuff in our game will happen:
 var scene = new THREE.Scene();
 var flat = {flatShading: true};
 var light = new THREE.AmbientLight('white', 0.8);
 scene.add( light );

 // The "camera" is what sees the stuff:
 var aspectRatio = window.innerWidth / window.innerHeight;
 var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
 camera.position.z = 500;
 scene.add(camera);

 // The "renderer" draws what the camera sees onto the screen:
 var renderer = new THREE.WebGLRenderer();
 renderer.setSize(window.innerWidth, window.innerHeight);
 document.body.appendChild(renderer.domElement);

 // ******** START CODING ON THE NEXT LINE ********

 var shape = new THREE.SphereGeometry(100, 20, 15);
 var cover = new THREE.MeshNormalMaterial(flat);
 var ball = new THREE.Mesh(shape, cover);
 scene.add(ball);
 ball.position.set(-250,250,-250);

 var shape = new THREE.CubeGeometry(300, 100, 100);
 var cover = new THREE.MeshNormalMaterial(flat);
 var box = new THREE.Mesh(shape, cover);
 scene.add(box);
 box.rotation.set(0.5, 0.5, 0);
 box.position.set(250, 250, -250);

 var shape = new THREE.CylinderGeometry(1, 100, 100, 4);
 var cover = new THREE.MeshNormalMaterial(flat);
```

```
 var tube = new THREE.Mesh(shape, cover);
 scene.add(tube);
 tube.rotation.set(0.5, 0, 0);
 tube.position.set(250, -250, -250);

 var shape = new THREE.PlaneGeometry(100, 100);
 var cover = new THREE.MeshNormalMaterial(flat);
 var ground = new THREE.Mesh(shape, cover);
 scene.add(ground);
 ground.rotation.set(0.5, 0, 0);
 ground.position.set(-250, -250, -250);

 var shape = new THREE.TorusGeometry(100, 25, 8, 25);
 var cover = new THREE.MeshNormalMaterial(flat);
 var donut = new THREE.Mesh(shape, cover);
 scene.add(donut);

 var clock = new THREE.Clock();

 function animate() {
   requestAnimationFrame(animate);
   var t = clock.getElapsedTime();

   ball.rotation.set(t, 2*t, 0);
   box.rotation.set(t, 2*t, 0);
   tube.rotation.set(t, 2*t, 0);
   ground.rotation.set(t, 2*t, 0);
   donut.rotation.set(t, 2*t, 0);

   renderer.render(scene, camera);
 }

 animate();

 // Now, show what the camera sees on the screen:
 renderer.render(scene, camera);
</script>
```

Code: Playing with the Console and Finding What's Broken

There was no working code from Chapter 2, *Debugging: Fixing Code When Things Go Wrong*. We wrote some broken code in 3DE and explored the JavaScript console.

Code: Making an Avatar

This is the final version of the avatar code from Chapter 3, *Project: Making an Avatar*:

```
<body></body>
<script src="/three.js"></script>
<script>
  // The "scene" is where stuff in our game will happen:
  var scene = new THREE.Scene();
  var flat = {flatShading: true};
  var light = new THREE.AmbientLight('white', 0.8);
  scene.add(light);

  // The "camera" is what sees the stuff:
  var aspectRatio = window.innerWidth / window.innerHeight;
  var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
  camera.position.z = 500;
  scene.add(camera);

  // The "renderer" draws what the camera sees onto the screen:
  var renderer = new THREE.WebGLRenderer({antialias: true});
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);

  // ******** START CODING ON THE NEXT LINE ********

  var body = new THREE.SphereGeometry(100);
  var cover = new THREE.MeshNormalMaterial();
  var avatar = new THREE.Mesh(body, cover);
  scene.add(avatar);

  var hand = new THREE.SphereGeometry(50);

  var rightHand = new THREE.Mesh(hand, cover);
  rightHand.position.set(-150, 0, 0);
  avatar.add(rightHand);

  var leftHand = new THREE.Mesh(hand, cover);
  leftHand.position.set(150, 0, 0);
  avatar.add(leftHand);
```

```
  var foot = new THREE.SphereGeometry(50);

  var rightFoot = new THREE.Mesh(foot, cover);
  rightFoot.position.set(-75, -125, 0);
  avatar.add(rightFoot);

  var leftFoot = new THREE.Mesh(foot, cover);
  leftFoot.position.set(75, -125, 0);
  avatar.add(leftFoot);

  // Now, animate what the camera sees on the screen:
  var isCartwheeling = false;
  var isFlipping = false;
  function animate() {
    requestAnimationFrame(animate);
    if (isCartwheeling) {
      avatar.rotation.z = avatar.rotation.z + 0.05;
    }
    if (isFlipping) {
      avatar.rotation.x = avatar.rotation.x + 0.05;
    }
    renderer.render(scene, camera);
  }
  animate();
</script>
```

Code: Moving Avatars

This is the moving-avatar code from Chapter 4, *Project: Moving Avatars*:

```
<body></body>
<script src="/three.js"></script>
<script>
 // The "scene" is where stuff in our game will happen:
 var scene = new THREE.Scene();
 var flat = {flatShading: true};
 var light = new THREE.AmbientLight('white', 0.8);
 scene.add(light);

 // The "camera" is what sees the stuff:
 var aspectRatio = window.innerWidth / window.innerHeight;
 var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
 camera.position.z = 500;
 // scene.add(camera);

 // The "renderer" draws what the camera sees onto the screen:
 var renderer = new THREE.WebGLRenderer({antialias: true});
 renderer.setSize(window.innerWidth, window.innerHeight);
 document.body.appendChild(renderer.domElement);

 // ******** START CODING ON THE NEXT LINE ********

 var marker = new THREE.Object3D();
 scene.add(marker);

 var body = new THREE.SphereGeometry(100);
 var cover = new THREE.MeshNormalMaterial();
 var avatar = new THREE.Mesh(body, cover);
 marker.add(avatar);

 var hand = new THREE.SphereGeometry(50);

 var rightHand = new THREE.Mesh(hand, cover);
 rightHand.position.set(-150, 0, 0);
 avatar.add(rightHand);

 var leftHand = new THREE.Mesh(hand, cover);
 leftHand.position.set(150, 0, 0);
```

```
avatar.add(leftHand);

var foot = new THREE.SphereGeometry(50);

var rightFoot = new THREE.Mesh(foot, cover);
rightFoot.position.set(-75, -125, 0);
avatar.add(rightFoot);

var leftFoot = new THREE.Mesh(foot, cover);
leftFoot.position.set(75, -125, 0);
avatar.add(leftFoot);

marker.add(camera);

function makeTreeAt(x, z) {
  var trunk = new THREE.Mesh(
    new THREE.CylinderGeometry(50, 50, 200),
    new THREE.MeshBasicMaterial({color: 'sienna'})
  );

  var top = new THREE.Mesh(
    new THREE.SphereGeometry(150),
    new THREE.MeshBasicMaterial({color: 'forestgreen'})
  );
  top.position.y = 175;
  trunk.add(top);

  trunk.position.set(x, -75, z);
  scene.add(trunk);
}

// Trees
makeTreeAt( 500,  0);
makeTreeAt(-500,  0);
makeTreeAt( 750, -1000);
makeTreeAt(-750, -1000);

// Now, animate what the camera sees on the screen:
var isCartwheeling = false;
var isFlipping = false;
function animate() {
  requestAnimationFrame(animate);

  if (isCartwheeling) {
    avatar.rotation.z = avatar.rotation.z + 0.05;
```

```
    }
    if (isFlipping) {
      avatar.rotation.x = avatar.rotation.x + 0.05;
    }
    renderer.render(scene, camera);
  }
  animate();

  document.addEventListener('keydown', sendKeyDown);
  function sendKeyDown(event) {
    var code = event.code;
    if (code == 'ArrowLeft') marker.position.x = marker.position.x - 5;
    if (code == 'ArrowRight') marker.position.x = marker.position.x + 5;
    if (code == 'ArrowUp') marker.position.z = marker.position.z - 5;
    if (code == 'ArrowDown') marker.position.z = marker.position.z + 5;

    if (code == 'KeyC') isCartwheeling = !isCartwheeling;
    if (code == 'KeyF') isFlipping = !isFlipping;
  }
</script>
```

Code: Functions: Use and Use Again

We intentionally broke a lot of things as we explored functions in Chapter 5, *Functions: Use and Use Again*. A copy of the code that works follows:

```
<body></body>
<script src="/three.js"></script>
<script src="/controls/FlyControls.js"></script>
<script>
  // The "scene" is where stuff in our game will happen:
  var scene = new THREE.Scene();
  var flat = {flatShading: true};
  var light = new THREE.AmbientLight('white', 0.8);
  scene.add(light);

  // The "camera" is what sees the stuff:
  var aspectRatio = window.innerWidth / window.innerHeight;
  var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
  camera.position.z = 500;
  scene.add(camera);

  // The "renderer" draws what the camera sees onto the screen:
  var renderer = new THREE.WebGLRenderer({antialias: true});
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);

  // ******** START CODING ON THE NEXT LINE ********

  var shape = new THREE.SphereGeometry(50);
  var cover = new THREE.MeshBasicMaterial({color: 'blue'});
  var planet = new THREE.Mesh(shape, cover);
  planet.position.set(-300, 0, 0);
  scene.add(planet);

  var shape = new THREE.SphereGeometry(50);
  var cover = new THREE.MeshBasicMaterial({color: 'yellow'});
  var planet = new THREE.Mesh(shape, cover);
  planet.position.set(200, 0, 250);
  scene.add(planet);

  function makePlanet() {
    var size = r(50);
```

```
  var x = r(1000) - 500;
  var y = r(1000) - 500;
  var z = r(1000) - 1000;
  var surface = rColor();

  var shape = new THREE.SphereGeometry(size);
  var cover = new THREE.MeshBasicMaterial({color: surface});
  var planet = new THREE.Mesh(shape, cover);
  planet.position.set(x, y, z);
  scene.add(planet);
}

makePlanet();
makePlanet();

for (var i=0; i<100; i++) {
  makePlanet();
}

console.log(Math.random());

function r(max) {
  if (max) return max * Math.random();
  return Math.random();
}

var randomNum = r();
console.log(randomNum);

randomNum = r(100);
console.log(randomNum);

console.log(r(100));
console.log(r(100));

function rColor() {
  return new THREE.Color(r(), r(), r());
}

var controls = new THREE.FlyControls(camera);
controls.movementSpeed = 100;
controls.rollSpeed = 0.5;
controls.dragToLook = true;
controls.autoForward = false;
```

```
  var clock = new THREE.Clock();

  function animate() {
    var delta = clock.getDelta();
    controls.update(delta);

    renderer.render(scene, camera);
    requestAnimationFrame(animate);
  }
  animate();
</script>
```

Code: Moving Hands and Feet

This is the code from Chapter 6, *Project: Moving Hands and Feet*:

```
<body></body>
<script src="/three.js"></script>
<script>
 // The "scene" is where stuff in our game will happen:
 var scene = new THREE.Scene();
 var flat = {flatShading: true};
 var light = new THREE.AmbientLight('white', 0.8);
 scene.add(light);

 // The "camera" is what sees the stuff:
 var aspectRatio = window.innerWidth / window.innerHeight;
 var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
 camera.position.z = 500;
 // scene.add(camera);

 // The "renderer" draws what the camera sees onto the screen:
 var renderer = new THREE.WebGLRenderer({antialias: true});
 renderer.setSize(window.innerWidth, window.innerHeight);
 document.body.appendChild(renderer.domElement);

 // ******** START CODING ON THE NEXT LINE ********

 var marker = new THREE.Object3D();
 scene.add(marker);

 //var cover = new THREE.MeshNormalMaterial({flatShading: true});
 var body = new THREE.SphereGeometry(100);
 var cover = new THREE.MeshNormalMaterial();
 var avatar = new THREE.Mesh(body, cover);
 marker.add(avatar);

 var hand = new THREE.SphereGeometry(50);

 var rightHand = new THREE.Mesh(hand, cover);
 rightHand.position.set(-150, 0, 0);
 avatar.add(rightHand);

 var leftHand = new THREE.Mesh(hand, cover);
```

```
leftHand.position.set(150, 0, 0);
avatar.add(leftHand);

var foot = new THREE.SphereGeometry(50);

var rightFoot = new THREE.Mesh(foot, cover);
rightFoot.position.set(-75, -125, 0);
avatar.add(rightFoot);

var leftFoot = new THREE.Mesh(foot, cover);
leftFoot.position.set(75, -125, 0);
avatar.add(leftFoot);

marker.add(camera);

function makeTreeAt(x, z) {
  var trunk = new THREE.Mesh(
    new THREE.CylinderGeometry(50, 50, 200),
    new THREE.MeshBasicMaterial({color: 'sienna'})
  );

  var top = new THREE.Mesh(
    new THREE.SphereGeometry(150),
    new THREE.MeshBasicMaterial({color: 'forestgreen'})
  );
  top.position.y = 175;
  trunk.add(top);

  trunk.position.set(x, -75, z);
  scene.add(trunk);
}

makeTreeAt( 500,  0);
makeTreeAt(-500,  0);
makeTreeAt( 750, -1000);
makeTreeAt(-750, -1000);

// Now, animate what the camera sees on the screen:

var clock = new THREE.Clock();
var isCartwheeling = false;
var isFlipping = false;
var isMovingRight = false;
var isMovingLeft = false;
var isMovingForward = false
```

```
var isMovingBack = false;


function animate() {
  requestAnimationFrame(animate);
  walk();
  acrobatics();
  renderer.render(scene, camera);
}
animate();

function walk() {
  if (!isWalking()) return;

  var speed = 10;
  var size = 100;
  var time = clock.getElapsedTime();
  var position = Math.sin(speed * time) * size;
  rightHand.position.z =  position;
  leftHand.position.z  = -position;
  rightFoot.position.z = -position;
  leftFoot.position.z  =  position;
}

function isWalking() {
  if (isMovingRight) return true;
  if (isMovingLeft) return true;
  if (isMovingForward) return true;
  if (isMovingBack) return true;
  return false;
}

function acrobatics() {
  if (isCartwheeling) {
    avatar.rotation.z = avatar.rotation.z + 0.05;
  }
  if (isFlipping) {
    avatar.rotation.x = avatar.rotation.x + 0.05;
  }
}

document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;
  if (code == 'ArrowLeft') {
```

```
      marker.position.x = marker.position.x - 5;
      isMovingLeft = true;

    }
    if (code == 'ArrowRight') {
      marker.position.x = marker.position.x + 5;
      isMovingRight = true;
    }
    if (code == 'ArrowUp') {
      marker.position.z = marker.position.z - 5;
      isMovingForward = true;
    }
    if (code == 'ArrowDown') {
      marker.position.z = marker.position.z + 5;
      isMovingBack = true;
    }

    if (code == 'KeyC') isCartwheeling = !isCartwheeling;
    if (code == 'KeyF') isFlipping = !isFlipping;
  }

  document.addEventListener('keyup', sendKeyUp);
  function sendKeyUp(event) {
    var code = event.code;
    if (code == 'ArrowLeft') isMovingLeft = false;
    if (code == 'ArrowRight') isMovingRight = false;
    if (code == 'ArrowUp') isMovingForward = false;
    if (code == 'ArrowDown') isMovingBack = false;
  }
</script>
```

Code: A Closer Look at JavaScript Fundamentals

There was no project code in Chapter 7, *A Closer Look at JavaScript Fundamentals*.

Code: Turning Our Avatar

This is the code from Chapter 8, *Project: Turning Our Avatar*:

```
<body></body>
<script src="/three.js"></script>
<script src="/tween.js"></script>
<script>
 // The "scene" is where stuff in our game will happen:
 var scene = new THREE.Scene();
 var flat = {flatShading: true};
 var light = new THREE.AmbientLight('white', 0.8);
 scene.add(light);

 // The "camera" is what sees the stuff:
 var aspectRatio = window.innerWidth / window.innerHeight;
 var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
 camera.position.z = 500;
 // scene.add(camera);

 // The "renderer" draws what the camera sees onto the screen:
 var renderer = new THREE.WebGLRenderer({antialias: true});
 renderer.setSize(window.innerWidth, window.innerHeight);
 document.body.appendChild(renderer.domElement);

 // ******** START CODING ON THE NEXT LINE ********

 var marker = new THREE.Object3D();
 scene.add(marker);

 //var cover = new THREE.MeshNormalMaterial(flat);
 var body = new THREE.SphereGeometry(100);
 var cover = new THREE.MeshNormalMaterial();
 var avatar = new THREE.Mesh(body, cover);
 marker.add(avatar);

 var hand = new THREE.SphereGeometry(50);

 var rightHand = new THREE.Mesh(hand, cover);
 rightHand.position.set(-150, 0, 0);
 avatar.add(rightHand);
```

```
var leftHand = new THREE.Mesh(hand, cover);
leftHand.position.set(150, 0, 0);
avatar.add(leftHand);

var foot = new THREE.SphereGeometry(50);

var rightFoot = new THREE.Mesh(foot, cover);
rightFoot.position.set(-75, -125, 0);
avatar.add(rightFoot);

var leftFoot = new THREE.Mesh(foot, cover);
leftFoot.position.set(75, -125, 0);
avatar.add(leftFoot);

marker.add(camera);

function makeTreeAt(x, z) {
  var trunk = new THREE.Mesh(
    new THREE.CylinderGeometry(50, 50, 200),
    new THREE.MeshBasicMaterial({color: 'sienna'})
  );

  var top = new THREE.Mesh(
    new THREE.SphereGeometry(150),
    new THREE.MeshBasicMaterial({color: 'forestgreen'})
  );
  top.position.y = 175;
  trunk.add(top);

  trunk.position.set(x, -75, z);
  scene.add(trunk);
}

makeTreeAt( 500,  0);
makeTreeAt(-500,  0);
makeTreeAt( 750, -1000);
makeTreeAt(-750, -1000);

// Now, animate what the camera sees on the screen:

var clock = new THREE.Clock();
var isCartwheeling = false;
var isFlipping = false;
var isMovingRight = false;
var isMovingLeft = false;
```

```
var isMovingForward = false;

var isMovingBack = false;
var direction;
var lastDirection;

 function animate() {
   requestAnimationFrame(animate);
   TWEEN.update();
   turn();
   walk();
   acrobatics();
   renderer.render(scene, camera);
 }
 animate();

function turn() {
  if (isMovingRight) direction = Math.PI/2;
  if (isMovingLeft) direction = -Math.PI/2;
  if (isMovingForward) direction = Math.PI;
  if (isMovingBack) direction = 0;
  if (!isWalking()) direction = 0;

  if (direction == lastDirection) return;
  lastDirection = direction;

  var tween = new TWEEN.Tween(avatar.rotation);
  tween.to({y: direction}, 500);
  tween.start();
}

function walk() {
  if (!isWalking()) return;

  var speed = 10;
  var size = 100;
  var time = clock.getElapsedTime();
  var position = Math.sin(speed * time) * size;
  rightHand.position.z =  position;
  leftHand.position.z  = -position;
  rightFoot.position.z = -position;
  leftFoot.position.z  =  position;
}

function isWalking() {
```

```
  if (isMovingRight) return true;
  if (isMovingLeft) return true;
  if (isMovingForward) return true;
  if (isMovingBack) return true;
  return false;
}

function acrobatics() {
  if (isCartwheeling) {
    avatar.rotation.z = avatar.rotation.z + 0.05;
  }
  if (isFlipping) {
    avatar.rotation.x = avatar.rotation.x + 0.05;
  }
}

document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;
  if (code == 'ArrowLeft') {
    marker.position.x = marker.position.x - 5;
    isMovingLeft = true;
  }
  if (code == 'ArrowRight') {
    marker.position.x = marker.position.x + 5;
    isMovingRight = true;
  }
  if (code == 'ArrowUp') {
    marker.position.z = marker.position.z - 5;
    isMovingForward = true;
  }
  if (code == 'ArrowDown') {
    marker.position.z = marker.position.z + 5;
    isMovingBack = true;
  }

  if (code == 'KeyC') isCartwheeling = !isCartwheeling;
  if (code == 'KeyF') isFlipping = !isFlipping;
}

document.addEventListener('keyup', sendKeyUp);
function sendKeyUp(event) {
  var code = event.code;
  if (code == 'ArrowLeft') isMovingLeft = false;
  if (code == 'ArrowRight') isMovingRight = false;
```

```
    if (code == 'ArrowUp') isMovingForward = false;
    if (code == 'ArrowDown') isMovingBack = false;
  }
</script>
```

Code: What's All That Other Code?

There was no new code in Chapter 9, *What's All That Other Code?*. We only explored the code that is automatically created when we start new projects.

Code: Collisions

This is the avatar code after we added collisions in Chapter 10, *Project: Collisions*:

```
<body></body>
<script src="/three.js"></script>
<script src="/tween.js"></script>
<script>
 // The "scene" is where stuff in our game will happen:
 var scene = new THREE.Scene();
 var flat = {flatShading: true};
 var light = new THREE.AmbientLight('white', 0.8);
 scene.add(light);

 // The "camera" is what sees the stuff:
 var aspectRatio = window.innerWidth / window.innerHeight;
 var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
 camera.position.z = 500;
 // scene.add(camera);

 // The "renderer" draws what the camera sees onto the screen:
 var renderer = new THREE.WebGLRenderer({antialias: true});
 renderer.setSize(window.innerWidth, window.innerHeight);
 document.body.appendChild(renderer.domElement);

 // ******** START CODING ON THE NEXT LINE ********

 var marker = new THREE.Object3D();
 scene.add(marker);

 //var cover = new THREE.MeshNormalMaterial(flat);
 var body = new THREE.SphereGeometry(100);
 var cover = new THREE.MeshNormalMaterial();
 var avatar = new THREE.Mesh(body, cover);
 marker.add(avatar);

 var hand = new THREE.SphereGeometry(50);

 var rightHand = new THREE.Mesh(hand, cover);
 rightHand.position.set(-150, 0, 0);
 avatar.add(rightHand);
```

```
 var leftHand = new THREE.Mesh(hand, cover);
 leftHand.position.set(150, 0, 0);
 avatar.add(leftHand);

 var foot = new THREE.SphereGeometry(50);

 var rightFoot = new THREE.Mesh(foot, cover);
 rightFoot.position.set(-75, -125, 0);
 avatar.add(rightFoot);

 var leftFoot = new THREE.Mesh(foot, cover);
 leftFoot.position.set(75, -125, 0);
 avatar.add(leftFoot);

 marker.add(camera);

var notAllowed = [];

function makeTreeAt(x, z) {
  var trunk = new THREE.Mesh(
    new THREE.CylinderGeometry(50, 50, 200),
    new THREE.MeshBasicMaterial({color: 'sienna'})
  );

  var top = new THREE.Mesh(
    new THREE.SphereGeometry(150),
    new THREE.MeshBasicMaterial({color: 'forestgreen'})
  );
  top.position.y = 175;
  trunk.add(top);

  var boundary = new THREE.Mesh(
    new THREE.CircleGeometry(300),
    new THREE.MeshNormalMaterial()
  );
  boundary.position.y = -100;
  boundary.rotation.x = -Math.PI/2;
  trunk.add(boundary);

  notAllowed.push(boundary);

  trunk.position.set(x, -75, z);
  scene.add(trunk);
}
```

```
makeTreeAt( 500,  0);
makeTreeAt(-500,  0);
makeTreeAt( 750, -1000);
makeTreeAt(-750, -1000);

// Now, animate what the camera sees on the screen:

var clock = new THREE.Clock();
var isCartwheeling = false;
var isFlipping = false;
var isMovingRight = false;
var isMovingLeft = false;
var isMovingForward = false;
var isMovingBack = false;
var direction;
var lastDirection;

function animate() {
  requestAnimationFrame(animate);
  TWEEN.update();
  turn();
  walk();
  acrobatics();
  renderer.render(scene, camera);
}
animate();

 function turn() {
   if (isMovingRight) direction = Math.PI/2;
   if (isMovingLeft) direction = -Math.PI/2;
   if (isMovingForward) direction = Math.PI;
   if (isMovingBack) direction = 0;
   if (!isWalking()) direction = 0;

   if (direction == lastDirection) return;
   lastDirection = direction;

   var tween = new TWEEN.Tween(avatar.rotation);
   tween.to({y: direction}, 500);
   tween.start();
 }

 function walk() {
   if (!isWalking()) return;
```

```
  var speed = 10;
  var size = 100;
  var time = clock.getElapsedTime();
  var position = Math.sin(speed * time) * size;
  rightHand.position.z =  position;
  leftHand.position.z  = -position;
  rightFoot.position.z = -position;
  leftFoot.position.z  =  position;
}

function isWalking() {
  if (isMovingRight) return true;
  if (isMovingLeft) return true;
  if (isMovingForward) return true;
  if (isMovingBack) return true;
  return false;
}

function acrobatics() {
  if (isCartwheeling) {
    avatar.rotation.z = avatar.rotation.z + 0.05;
  }
  if (isFlipping) {
    avatar.rotation.x = avatar.rotation.x + 0.05;
  }
}

function isColliding() {
  var vector = new THREE.Vector3(0, -1, 0);
  var raycaster = new THREE.Raycaster(marker.position, vector);

  var intersects = raycaster.intersectObjects(notAllowed);
  if (intersects.length > 0) return true;

  return false;
}

document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;
  if (code == 'ArrowLeft') {
    marker.position.x = marker.position.x - 5;
    isMovingLeft = true;
  }
```

```
  if (code == 'ArrowRight') {
    marker.position.x = marker.position.x + 5;
    isMovingRight = true;
  }
  if (code == 'ArrowUp') {
    marker.position.z = marker.position.z - 5;
    isMovingForward = true;
  }
  if (code == 'ArrowDown') {
    marker.position.z = marker.position.z + 5;
    isMovingBack = true;
  }

  if (code == 'KeyC') isCartwheeling = !isCartwheeling;
  if (code == 'KeyF') isFlipping = !isFlipping;

  if (isColliding()) {
    if (isMovingLeft)    marker.position.x = marker.position.x + 5;
    if (isMovingRight)   marker.position.x = marker.position.x - 5;
    if (isMovingForward) marker.position.z = marker.position.z + 5;
    if (isMovingBack)    marker.position.z = marker.position.z - 5;
  }
 }

 document.addEventListener('keyup', sendKeyUp);
 function sendKeyUp(event) {
   var code = event.code;
   if (code == 'ArrowLeft') isMovingLeft = false;
   if (code == 'ArrowRight') isMovingRight = false;
   if (code == 'ArrowUp') isMovingForward = false;
   if (code == 'ArrowDown') isMovingBack = false;
 }
</script>
```

Code: Fruit Hunt

This is the avatar code after we added it to the fruit-hunt game in Chapter 11, *Project: Fruit Hunt*. This code uses `WebGLRenderer` to make the trees a little prettier, but the `CanvasRenderer` should work nearly as well.

```
<body></body>
<script src="/three.js"></script>
<script src="/tween.js"></script>
<script src="/scoreboard.js"></script>
<script src="/sounds.js"></script>
<script>
 // The "scene" is where stuff in our game will happen:
 var scene = new THREE.Scene();
 var flat = {flatShading: true};
 var light = new THREE.AmbientLight('white', 0.8);
 scene.add(light);

 // The "camera" is what sees the stuff:
 var aspectRatio = window.innerWidth / window.innerHeight;
 var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
 camera.position.z = 500;
 // scene.add(camera);

 // The "renderer" draws what the camera sees onto the screen:
 var renderer = new THREE.WebGLRenderer({antialias: true});
 renderer.setSize(window.innerWidth, window.innerHeight);
 document.body.appendChild(renderer.domElement);

 // ******** START CODING ON THE NEXT LINE ********

 var marker = new THREE.Object3D();
 scene.add(marker);

 //var cover = new THREE.MeshNormalMaterial(flat);
 var body = new THREE.SphereGeometry(100);
 var cover = new THREE.MeshNormalMaterial();
 var avatar = new THREE.Mesh(body, cover);
 marker.add(avatar);

 var hand = new THREE.SphereGeometry(50);
```

```
var rightHand = new THREE.Mesh(hand, cover);
rightHand.position.set(-150, 0, 0);
avatar.add(rightHand);

var leftHand = new THREE.Mesh(hand, cover);
leftHand.position.set(150, 0, 0);
avatar.add(leftHand);

var foot = new THREE.SphereGeometry(50);

var rightFoot = new THREE.Mesh(foot, cover);
rightFoot.position.set(-75, -125, 0);
avatar.add(rightFoot);

var leftFoot = new THREE.Mesh(foot, cover);
leftFoot.position.set(75, -125, 0);
avatar.add(leftFoot);

marker.add(camera);

var scoreboard = new Scoreboard();
scoreboard.countdown(45);
scoreboard.score();
scoreboard.help(
  'Arrow keys to move. ' +
  'Space bar to jump for fruit. ' +
  'Watch for shaking trees with fruit. ' +
  'Get near the tree and jump before the fruit is gone!'
);
scoreboard.onTimeExpired(timeExpired);
function timeExpired() {
  scoreboard.message("Game Over!");
}

var notAllowed = [];
var treeTops = [];

function makeTreeAt(x, z) {
  var trunk = new THREE.Mesh(
    new THREE.CylinderGeometry(50, 50, 200),
    new THREE.MeshBasicMaterial({color: 'sienna'})
  );

  var top = new THREE.Mesh(
    new THREE.SphereGeometry(150),
```

```
    new THREE.MeshBasicMaterial({color: 'forestgreen'})
  );
  top.position.y = 175;
  trunk.add(top);

  var boundary = new THREE.Mesh(
    new THREE.CircleGeometry(300),
    new THREE.MeshNormalMaterial()
  );
  boundary.position.y = -100;
  boundary.rotation.x = -Math.PI/2;
  trunk.add(boundary);

  notAllowed.push(boundary);
  treeTops.push(top);

  trunk.position.set(x, -75, z);
  scene.add(trunk);
}

makeTreeAt( 500,  0);
makeTreeAt(-500,  0);
makeTreeAt( 750, -1000);
makeTreeAt(-750, -1000);

var treasureTreeNumber;
function updateTreasureTreeNumber() {
  var rand = Math.random() * treeTops.length;
  treasureTreeNumber = Math.floor(rand);
}

function shakeTreasureTree() {
  updateTreasureTreeNumber();

  var tween = new TWEEN.Tween({shake: 0});
  tween.to({shake: 20 * 2 * Math.PI}, 8*1000);
  tween.onUpdate(shakeTreeUpdate);
  tween.onComplete(shakeTreeComplete);
  tween.start();
}

function shakeTreeUpdate(update) {
  var top = treeTops[treasureTreeNumber];
  top.position.x = 50 * Math.sin(update.shake);
```

```
}

function shakeTreeComplete() {
  var top = treeTops[treasureTreeNumber];
  top.position.x = 0;
  setTimeout(shakeTreasureTree, 2*1000);
}

shakeTreasureTree();

// Now, animate what the camera sees on the screen:

var clock = new THREE.Clock();
var isCartwheeling = false;
var isFlipping = false;
var isMovingRight = false;
var isMovingLeft = false;
var isMovingForward = false;
var isMovingBack = false;
var direction;
var lastDirection;

function animate() {
  requestAnimationFrame(animate);
  TWEEN.update();
  turn();
  walk();
  acrobatics();
  renderer.render(scene, camera);
}
animate();

function turn() {
  if (isMovingRight) direction = Math.PI/2;
  if (isMovingLeft) direction = -Math.PI/2;
  if (isMovingForward) direction = Math.PI;
  if (isMovingBack) direction = 0;
  if (!isWalking()) direction = 0;

  if (direction == lastDirection) return;
  lastDirection = direction;

  var tween = new TWEEN.Tween(avatar.rotation);
  tween.to({y: direction}, 500);
  tween.start();
```

```
}

function walk() {
  if (!isWalking()) return;

  var speed = 10;
  var size = 100;
  var time = clock.getElapsedTime();
  var position = Math.sin(speed * time) * size;
  rightHand.position.z =  position;
  leftHand.position.z  = -position;
  rightFoot.position.z = -position;
  leftFoot.position.z  =  position;
}

function isWalking() {
  if (isMovingRight) return true;
  if (isMovingLeft) return true;
  if (isMovingForward) return true;
  if (isMovingBack) return true;
  return false;
}

function acrobatics() {
  if (isCartwheeling) {
    avatar.rotation.z = avatar.rotation.z + 0.05;
  }
  if (isFlipping) {
    avatar.rotation.x = avatar.rotation.x + 0.05;
  }
}

function jump() {
  if (avatar.position.y > 0) return;
  checkForTreasure();
  animateJump();
}

function checkForTreasure() {
  var top = treeTops[treasureTreeNumber];
  var tree = top.parent;
  var p1 = tree.position;
  var p2 = marker.position;
  var xDiff = p1.x - p2.x;
```

```
  var zDiff = p1.z - p2.z;

  var distance = Math.sqrt(xDiff*xDiff + zDiff*zDiff);
  if (distance < 500) scorePoints();
}

function scorePoints() {
  if (scoreboard.getTimeRemaining() == 0) return;
  scoreboard.addPoints(10);
  Sounds.bubble.play();
  animateFruit();
}

function animateJump() {
  var tween = new TWEEN.Tween({jump: 0});
  tween.to({jump: Math.PI}, 400);
  tween.onUpdate(animateJumpUpdate);
  tween.onComplete(animateJumpComplete);
  tween.start();
}

function animateJumpUpdate(update) {
  avatar.position.y = 100 * Math.sin(update.jump);
}

function animateJumpComplete() {
  avatar.position.y = 0;
}

var fruit;
function animateFruit() {
  if (fruit) return;

  fruit = new THREE.Mesh(
    new THREE.CylinderGeometry(25, 25, 5, 25),
    new THREE.MeshBasicMaterial({color: 'gold'})
  );
  marker.add(fruit);

  var tween = new TWEEN.Tween({height: 200, spin: 0});
  tween.to({height: 350, spin: 2 * Math.PI}, 500);
  tween.onUpdate(animateFruitUpdate);
  tween.onComplete(animateFruitComplete);
  tween.start();
}
```

```
function animateFruitUpdate(update) {
  fruit.position.y = update.height;
  fruit.rotation.x = update.spin;
}

function animateFruitComplete() {
  marker.remove(fruit);
  fruit = undefined;
}

function isColliding() {
  var vector = new THREE.Vector3(0, -1, 0);
  var raycaster = new THREE.Raycaster(marker.position, vector);

  var intersects = raycaster.intersectObjects(notAllowed);
  if (intersects.length > 0) return true;

  return false;
}

document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;

  if (code == 'ArrowLeft') {
    marker.position.x = marker.position.x - 5;
    isMovingLeft = true;
  }
  if (code == 'ArrowRight') {
    marker.position.x = marker.position.x + 5;
    isMovingRight = true;
  }
  if (code == 'ArrowUp') {
    marker.position.z = marker.position.z - 5;
    isMovingForward = true;
  }
  if (code == 'ArrowDown') {
    marker.position.z = marker.position.z + 5;
    isMovingBack = true;
  }

  if (code == 'KeyC') isCartwheeling = !isCartwheeling;
  if (code == 'KeyF') isFlipping = !isFlipping;
```

```
  if (code == 'Space') jump();

  if (isColliding()) {
    if (isMovingLeft)    marker.position.x = marker.position.x + 5;
    if (isMovingRight)   marker.position.x = marker.position.x - 5;
    if (isMovingForward) marker.position.z = marker.position.z + 5;
    if (isMovingBack)    marker.position.z = marker.position.z - 5;
  }
 }

 document.addEventListener('keyup', sendKeyUp);
 function sendKeyUp(event) {
  var code = event.code;
  if (code == 'ArrowLeft') isMovingLeft = false;
  if (code == 'ArrowRight') isMovingRight = false;
  if (code == 'ArrowUp') isMovingForward = false;
  if (code == 'ArrowDown') isMovingBack = false;
 }
</script>
```

Code: Working with Lights and Materials

This is the final version of the code that we used to explore lights and materials in Chapter 12, *Working with Lights and Materials*:

```
<body></body>
<script src="/three.js"></script>
<script src="/controls/OrbitControls.js"></script>
<script>
  // The "scene" is where stuff in our game will happen:
  var scene = new THREE.Scene();
  var flat = {flatShading: true};
  var light = new THREE.AmbientLight('white', 0.1);
  scene.add(light);

  // The "camera" is what sees the stuff:
  var aspectRatio = window.innerWidth / window.innerHeight;
  var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
  camera.position.z = 500;
  camera.position.y = 500;
  camera.lookAt(new THREE.Vector3(0,0,0));
  scene.add(camera);

  // The "renderer" draws what the camera sees onto the screen:
  var renderer = new THREE.WebGLRenderer({antialias: true});
  renderer.shadowMap.enabled = true;
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);

  // ******** START CODING ON THE NEXT LINE ********

  var shape = new THREE.TorusGeometry(50, 20, 8, 20);
  var cover = new THREE.MeshPhongMaterial({color: 'red'});
  cover.specular.setRGB(0.9, 0.9, 0.9);
  var donut = new THREE.Mesh(shape, cover);
  donut.position.set(0, 150, 0);
  donut.castShadow = true;
  scene.add(donut);

  var texture = new THREE.TextureLoader().load("/textures/hardwood.png");
  var shape = new THREE.PlaneGeometry(1000, 1000, 10, 10);
  var cover = new THREE.MeshPhongMaterial();
```

```
cover.map = texture;
var ground = new THREE.Mesh(shape, cover);
ground.rotation.x = -Math.PI/2;
ground.receiveShadow = true;
scene.add(ground);

var point = new THREE.PointLight('white', 0.4);
point.position.set(0, 300, -100);
point.castShadow = true;
// scene.add(point);

var shape = new THREE.SphereGeometry(10);
var cover = new THREE.MeshPhongMaterial({emissive: 'white'});
var phonyLight = new THREE.Mesh(shape, cover);
point.add(phonyLight);

var spot = new THREE.SpotLight('white', 0.7);
spot.position.set(200, 300, 0);
spot.castShadow = true;
spot.shadow.camera.far = 750;
spot.angle = Math.PI/4;
spot.penumbra = 0.1;
scene.add(spot);

var shape = new THREE.CylinderGeometry(4, 10, 20);
var cover = new THREE.MeshPhongMaterial({emissive: 'white'});
var phonyLight = new THREE.Mesh(shape, cover);
phonyLight.position.y = 10;
phonyLight.rotation.z = -Math.PI/8;
spot.add(phonyLight);

var sunlight = new THREE.DirectionalLight('white', 0.4);
sunlight.position.set(200, 300, 0);
sunlight.castShadow = true;
// scene.add(sunlight);
var d = 500;
sunlight.shadow.camera.left = -d;
sunlight.shadow.camera.right = d;
sunlight.shadow.camera.top = d;
sunlight.shadow.camera.bottom = -d;

controls = new THREE.OrbitControls( camera, renderer.domElement );

// Start Animation
```

```
  var clock = new THREE.Clock();

  function animate() {
    requestAnimationFrame(animate);
    var t = clock.getElapsedTime();

    // Animation code goes here...
    donut.rotation.set(t, 2*t, 0);
    donut.position.z = 200 * Math.sin(t);

    renderer.render(scene, camera);
  }
  animate();
</script>
```

Code: Phases of the Moon

This is the final version of the moon-phases code from Chapter 13, *Project: Phases of the Moon*:

```
<body></body>
<script src="/three.js"></script>
<script src="/controls/FlyControls.js"></script>
<script>
  // The "scene" is where stuff in our game will happen:
  var scene = new THREE.Scene();
  var flat = {flatShading: true};
  var light = new THREE.AmbientLight('white', 0.1);
  scene.add(light);

  // The "camera" is what sees the stuff:
  var aspectRatio = window.innerWidth / window.innerHeight;
  var w = window.innerWidth / 2;
  var h = window.innerHeight / 2;
  var camera = new THREE.OrthographicCamera(-w, w, h, -h, 1, 10000);
  camera.position.y = 500;
  camera.rotation.x = -Math.PI/2;
  scene.add(camera);
  var aboveCam = camera;

  // The "renderer" draws what the camera sees onto the screen:
  var renderer = new THREE.WebGLRenderer({antialias: true});
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);

  // ******** START CODING ON THE NEXT LINE ********

  var cover = new THREE.MeshPhongMaterial({emissive: 'yellow'});
  var shape = new THREE.SphereGeometry(50, 32, 16);
  var sun = new THREE.Mesh(shape, cover);
  scene.add(sun);

  var sunlight = new THREE.PointLight('white', 1.7);
  sun.add(sunlight);

  var earthLocal = new THREE.Object3D();
  earthLocal.position.x = 300;
```

```
scene.add(earthLocal);

var texture = new THREE.TextureLoader().load("/textures/earth.png");
var cover = new THREE.MeshPhongMaterial({map: texture});
var shape = new THREE.SphereGeometry(20, 32, 16);
var earth = new THREE.Mesh(shape, cover);
earthLocal.add(earth);

var moonOrbit = new THREE.Object3D();
earthLocal.add(moonOrbit);

var texture = new THREE.TextureLoader().load("/textures/moon.png");
var cover = new THREE.MeshPhongMaterial({map: texture, specular: 'black'});
var shape = new THREE.SphereGeometry(15, 32, 16);
var moon = new THREE.Mesh(shape, cover);
moon.position.set(0, 0, 100);
moon.rotation.set(0, Math.PI/2, 0);
moonOrbit.add(moon);

var moonCam = new THREE.PerspectiveCamera(70, aspectRatio, 1, 10000);
moonCam.position.z = 25;
moonCam.rotation.y = Math.PI;
moonOrbit.add(moonCam);

camera = moonCam;

var shipCam = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
shipCam.position.set(0, 0, 500);
scene.add(shipCam);

var controls = new THREE.FlyControls(shipCam, renderer.domElement);
controls.movementSpeed = 42;
controls.rollSpeed = 0.15;
controls.dragToLook = true;
controls.autoForward = false;

var cover = new THREE.PointsMaterial({color: 'white', size: 15});
var shape = new THREE.Geometry();

var distance = 4000;
for (var i = 0; i < 500; i++) {
  var ra = 2 * Math.PI * Math.random();
  var dec = 2 * Math.PI * Math.random();

  var point = new THREE.Vector3();
```

```
  point.x = distance * Math.cos(dec) * Math.cos(ra);
  point.y = distance * Math.sin(dec);
  point.z = distance * Math.cos(dec) * Math.sin(ra);

  shape.vertices.push(point);
}

var stars = new THREE.Points(shape, cover);
scene.add(stars);


// Start Animation

var clock = new THREE.Clock();
function animate() {
  requestAnimationFrame(animate);

  // Animation code goes here...
  var delta = clock.getDelta();
  controls.update(delta);

  renderer.render(scene, camera);
}
animate();

var speed = 10;
var pause = false;
var days = 0;
var clock2 = new THREE.Clock();

function gameStep() {
  setTimeout(gameStep, 1000/30);

  if (pause) return;

  days = days + speed * clock2.getDelta();

  earth.rotation.y = days;

  var years = days / 365.25;
  earthLocal.position.x = 300 * Math.cos(years);
  earthLocal.position.z = -300 * Math.sin(years);
  moonOrbit.rotation.y = days / 29.5;
}
```

```
  gameStep();

  document.addEventListener("keydown", sendKeyDown);
  function sendKeyDown(event) {
    var code = event.code;

    if (code == 'Digit1') speed = 1;
    if (code == 'Digit2') speed = 10;
    if (code == 'Digit3') speed = 100;
    if (code == 'Digit4') speed = 1000;
    if (code == 'KeyP') pauseUnpause();
    if (code == 'KeyC') switchCamera();
    if (code == 'KeyF') fly();
  }

  function pauseUnpause() {
    pause = !pause;
    clock2.running = false;
  }

  function switchCamera() {
    if (camera == moonCam) camera = aboveCam;
    else camera = moonCam;
  }

  function fly() {
    camera = shipCam;
  }
</script>
```

Code: The Purple Fruit Monster Game

This is the final version of the game code from Chapter 14, *Project: The Purple Fruit Monster Game*:

```
<body></body>
<script src="/three.js"></script>
<script src="/physi.js"></script>
<script src="/scoreboard.js"></script>
<script>
  // Physics settings
  Physijs.scripts.ammo = '/ammo.js';
  Physijs.scripts.worker = '/physijs_worker.js';

  // The "scene" is where stuff in our game will happen:
  var scene = new Physijs.Scene();
  scene.setGravity(new THREE.Vector3( 0, -250, 0 ));
  var flat = {flatShading: true};
  var light = new THREE.AmbientLight('white', 0.8);
  scene.add(light);

  // The "camera" is what sees the stuff:
  var w = window.innerWidth / 2;
  var h = window.innerHeight / 2;
  var camera = new THREE.OrthographicCamera(-w, w, h, -h, 1, 10000);
  camera.position.z = 500;
  scene.add(camera);

  // The "renderer" draws what the camera sees onto the screen:
  var renderer = new THREE.WebGLRenderer({antialias: true});
  renderer.setSize(window.innerWidth, window.innerHeight);
  renderer.setClearColor('skyblue');
  document.body.appendChild(renderer.domElement);

  // ******** START CODING ON THE NEXT LINE ********

  var gameOver = false;

  var ground = addGround();
  var avatar = addAvatar();
  var scoreboard = addScoreboard();
```

```
reset();

function addGround() {
  var shape = new THREE.BoxGeometry(2*w, h, 10);
  var cover = new THREE.MeshBasicMaterial({color: 'lawngreen'});
  var ground = new Physijs.BoxMesh(shape, cover, 0);
  ground.position.y = -h/2;
  scene.add(ground);
  return ground;
}

function addAvatar() {
  var shape = new THREE.CubeGeometry(100, 100, 1);
  var cover = new THREE.MeshBasicMaterial({visible: false});
  var avatar = new Physijs.BoxMesh(shape, cover, 1);
  scene.add(avatar);

  var image = new THREE.TextureLoader().load("/images/monster.png");
  var material = new THREE.SpriteMaterial({map: image});
  var sprite = new THREE.Sprite(material);
  sprite.scale.set(100, 100, 1);
  avatar.add(sprite);

  avatar.setLinearFactor(new THREE.Vector3(1, 1, 0));
  avatar.setAngularFactor(new THREE.Vector3(0, 0, 0));

  return avatar;
}

function addScoreboard() {
  var scoreboard = new Scoreboard();
  scoreboard.score();
  scoreboard.help(
    "Use arrow keys to move and the space bar to jump. " +
    "Don't let the fruit get past you!!!"
  );
  return scoreboard;
}

function reset() {
  avatar.__dirtyPosition = true;
  avatar.position.set(-0.6*w, 200, 0);
  avatar.setLinearVelocity(new THREE.Vector3(0, 250, 0));

  scoreboard.score(0);
```

```
  scoreboard.message('');


  var last = scene.children.length - 1;
  for (var i=last; i>=0; i--) {
    var obj = scene.children[i];
    if (obj.isFruit) scene.remove(obj);
  }

  if (gameOver) {
    gameOver = false;
    animate();
  }
}

function launchFruit() {
  if (gameOver) return;

  var speed = 500 + (10 * Math.random() * scoreboard.getScore());
  var fruit = makeFruit();
  fruit.setLinearVelocity(new THREE.Vector3(-speed, 0, 0));
  fruit.setAngularVelocity(new THREE.Vector3(0, 0, 10));
}
launchFruit();
setInterval(launchFruit, 3*1000);

function makeFruit() {
  var shape = new THREE.SphereGeometry(40, 16, 24);
  var cover = new THREE.MeshBasicMaterial({visible: false});
  var fruit = new Physijs.SphereMesh(shape, cover);
  fruit.position.set(w, 40, 0);
  scene.add(fruit);

  var image = new THREE.TextureLoader().load("/images/fruit.png");
  cover = new THREE.MeshBasicMaterial({map: image, transparent: true});
  shape = new THREE.PlaneGeometry(80, 80);
  var picturePlane = new THREE.Mesh(shape, cover);
  fruit.add(picturePlane);

  fruit.setAngularFactor(new THREE.Vector3(0, 0, 1));
  fruit.setLinearFactor(new THREE.Vector3(1, 1, 0));
  fruit.isFruit = true;

  return fruit;
}
```

```
function checkMissedFruit() {

  var count=0;
  for (var i=0; i<scene.children.length; i++) {
    var obj = scene.children[i];
    if (obj.isFruit && obj.position.x < -w) count++;
  }
  if (count > 10) {
    gameOver = true;
    scoreboard.message(
      'Purple Fruit Monster missed too much fruit! ' +
      'Press R to try again.'
    );
  }
}

function gameStep() {
  scene.simulate();
  setTimeout(gameStep, 1000/30);
}
gameStep();

var clock = new THREE.Clock();
function animate() {
  if (gameOver) return;

  requestAnimationFrame(animate);
  var t = clock.getElapsedTime();

  // Animation code goes here...
  renderer.render(scene, camera);
}
animate();

document.addEventListener("keydown", sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;

  if (code == 'ArrowLeft') left();
  if (code == 'ArrowRight') right();
  if (code == 'ArrowUp') up();
  if (code == 'ArrowDown') down();
  if (code == 'Space') up();
  if (code == 'KeyR') reset();
```

```
  }

  function left()  { move(-100, 0); }
  function right() { move(100,  0); }
  function up()    { move(0, 250); }
  function down()  { move(0, -50); }

  function move(x, y) {
    if (x > 0) avatar.scale.x = 1;
    if (x < 0) avatar.scale.x = -1;

    var dir = new THREE.Vector3(x, y, 0);
    avatar.applyCentralImpulse(dir);
  }

  avatar.addEventListener('collision', sendCollision);
  function sendCollision(object) {
    if (gameOver) return;

    if (object.isFruit) {
      scoreboard.addPoints(10);
      avatar.setLinearVelocity(new THREE.Vector3(0, 250, 0));
      scene.remove(object);
    }
    if (object == ground) {
      gameOver = true;
      scoreboard.message(
        "Purple Fruit Monster crashed! " +
        "Press R to try again."
      );
    }
  }
</script>
```

Code: Tilt-a-Board

This is the final version of the game code from Chapter 15, *Project: Tilt-a-Board*, including the two bonus challenges:

```
<body></body>
<script src="/three.js"></script>
<script src="/physi.js"></script>
<script src="/spe.js"></script>
<script>
  // Physics settings
  Physijs.scripts.ammo = '/ammo.js';
  Physijs.scripts.worker = '/physijs_worker.js';

  // The "scene" is where stuff in our game will happen:
  var scene = new Physijs.Scene();
  scene.setGravity(new THREE.Vector3( 0, -100, 0 ));
  var flat = {flatShading: true};
  var light = new THREE.AmbientLight('white', 0.2);
  scene.add(light);

  // The "camera" is what sees the stuff:
  var aspectRatio = window.innerWidth / window.innerHeight;
  var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
  camera.position.z = 500;
  scene.add(camera);

  // The "renderer" draws what the camera sees onto the screen:
  var renderer = new THREE.WebGLRenderer({antialias: true});
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);

  camera.position.set(0, 100, 200);
  camera.lookAt(new THREE.Vector3(0, 0, 0));
  renderer.shadowMap.enabled = true;

  // ******** START CODING ON THE NEXT LINE ********

  var lights = addLights();
  var ball = addBall();
  var board = addBoard();
  var goal = addGoal();
```

```
reset();
addGoalLight();

function addLights() {
  var lights = new THREE.Object3D();

  var light1 = new THREE.PointLight('white', 0.4);
  light1.position.set(50, 50, -100);
  light1.castShadow = true;
  lights.add(light1);

  var light2 = new THREE.PointLight('white', 0.5);
  light2.position.set(-50, 50, 175);
  light2.castShadow = true;
  lights.add(light2);

  scene.add(lights);
  return lights;
}

function addBall() {
  var shape = new THREE.SphereGeometry(10, 25, 21);
  var cover = new THREE.MeshPhongMaterial({color: 'red'});
  cover.specular.setRGB(0.6, 0.6, 0.6);

  var ball = new Physijs.SphereMesh(shape, cover);
  ball.castShadow = true;

  scene.add(ball);
  return ball;
}

function addBoard() {
  var cover = new THREE.MeshPhongMaterial({color: 'gold'});
  cover.specular.setRGB(0.9, 0.9, 0.9);

  var shape = new THREE.CubeGeometry(50, 2, 200);
  var beam1 = new Physijs.BoxMesh(shape, cover, 0);
  beam1.position.set(-37, 0, 0);
  beam1.receiveShadow = true;

  var beam2 = new Physijs.BoxMesh(shape, cover, 0);
  beam2.position.set(75, 0, 0);
  beam2.receiveShadow = true;
```

```
  beam1.add(beam2);


  shape = new THREE.CubeGeometry(200, 2, 50);
  var beam3 = new Physijs.BoxMesh(shape, cover, 0);
  beam3.position.set(40, 0, -40);
  beam3.receiveShadow = true;
  beam1.add(beam3);

  var beam4 = new Physijs.BoxMesh(shape, cover, 0);
  beam4.position.set(40, 0, 40);
  beam4.receiveShadow = true;
  beam1.add(beam4);

  beam1.rotation.set(0.1, 0, 0);
  scene.add(beam1);
  return beam1;
}

function addGoal() {
  shape = new THREE.CubeGeometry(100, 2, 100);
  cover = new THREE.MeshNormalMaterial({wireframe: true});
  var goal = new Physijs.BoxMesh(shape, cover,  0);
  goal.position.y = -50;
  scene.add(goal);

  return goal;
}

function reset() {
  ball.__dirtyPosition = true;
  ball.__dirtyRotation = true;
  ball.position.set(-33, 200, -65);
  ball.setLinearVelocity(new THREE.Vector3(0, 0, 0));
  ball.setAngularVelocity(new THREE.Vector3(0, 0, 0));

  board.__dirtyRotation = true;
  board.rotation.set(0.1, 0, 0);
}

var fire, goalFire;
function addGoalLight(){
  var material = new THREE.TextureLoader().load('/textures/spe/star.png');
  fire = new SPE.Group({texture: {value: material}});
  goalFire = new SPE.Emitter({particleCount: 1000, maxAge: {value: 4}});
```

```
  fire.addEmitter(goalFire);


  scene.add(fire.mesh);

  goalFire.velocity.value = new THREE.Vector3(0, 75, 0);
  goalFire.velocity.spread = new THREE.Vector3(10, 7.5, 5);
  goalFire.acceleration.value = new THREE.Vector3(0, -15, 0);
  goalFire.position.spread = new THREE.Vector3(25, 0, 0);
  goalFire.size.value = 25;
  goalFire.size.spread = 10;
  goalFire.color.value = [new THREE.Color('white'), new THREE.Color('red')];
  goalFire.disable();
}

function win(flashCount) {
  if (!flashCount) flashCount = 0;

  goalFire.enable();

  flashCount++;
  if (flashCount > 10) {
    reset();
    goalFire.disable();
    return;
  }

  setTimeout(win, 500, flashCount);
}
goal.addEventListener('collision', win);

function addBackground() {
  var cover = new THREE.PointsMaterial({color: 'white', size: 2});
  var shape = new THREE.Geometry();

  var distance = 500;
  for (var i = 0; i < 2000; i++) {
    var ra = 2 * Math.PI * Math.random();
    var dec = 2 * Math.PI * Math.random();

    var point = new THREE.Vector3();
    point.x = distance * Math.cos(dec) * Math.cos(ra);
    point.y = distance * Math.sin(dec);
    point.z = distance * Math.cos(dec) * Math.sin(ra);
```

```
    shape.vertices.push(point);
  }

  var stars = new THREE.Points(shape, cover);
  scene.add(stars);
}

// Animate motion in the game
var clock = new THREE.Clock();
function animate() {
  requestAnimationFrame(animate);
  renderer.render(scene, camera);

  var dt = clock.getDelta();

  fire.tick(dt);
  lights.rotation.y = lights.rotation.y + dt/2;
}
animate();

// Run physics
function gameStep() {
  if (ball.position.y < -500) reset(ball);
  scene.simulate();

  // Update physics 60 times a second so that motion is smooth
  setTimeout(gameStep, 1000/60);
}
gameStep();

document.addEventListener("keydown", sendKeyDown);

function sendKeyDown(event){
  var code = event.code;
  if (code == 'ArrowLeft') left();
  if (code == 'ArrowRight') right();
  if (code == 'ArrowUp') up();
  if (code == 'ArrowDown') down();
}

function left()  { tilt('z',  0.02); }
function right() { tilt('z', -0.02); }
function up()    { tilt('x', -0.02); }
function down()  { tilt('x',  0.02); }
```

```
  function tilt(dir, amount) {
    board.__dirtyRotation = true;
    board.rotation[dir] = board.rotation[dir] + amount;
  }
</script>
```

Code: Learning about JavaScript Objects

The code from Chapter 16, *Learning about JavaScript Objects*, should look something like the following:

```
<body></body>
<script src="/three.js"></script>
<script>
  // Your code goes here...

  var bestMovie = {
    title: 'Star Wars',
    year: 1977,
  };

  var bestMovie = {
    title: 'Star Wars',
    year: 1977,
    stars: ['Mark Hamill', 'Harrison Ford', 'Carrie Fisher'],
  };

  var bestMovie = {
    title: 'Star Wars',
    year: 1977,
    stars: ['Mark Hamill', 'Harrison Ford', 'Carrie Fisher'],
    logMe: function() {
      console.log(this.title + ', starring: ' + this.stars);
    },
  };

  bestMovie.logMe();

  var bestMovie = {
    title: 'Star Wars',
    year: 1977,
    stars: ['Mark Hamill', 'Harrison Ford', 'Carrie Fisher'],
    logMe: function() {
      var me = this.about();
      console.log(me);
    },
    about: function() {
      return this.title + ', starring: ' + this.stars;
```

```
  },
};
bestMovie.logMe();

var greatMovie = Object.create(bestMovie);
greatMovie.logMe();
// => Star Wars, starring: Mark Hamill,Harrison Ford,Carrie Fisher

greatMovie.title = 'Toy Story';
greatMovie.year = 1995;
greatMovie.stars = ['Tom Hanks', 'Tim Allen'];

greatMovie.logMe();
// => Toy Story, starring: Tom Hanks,Tim Allen

bestMovie.logMe();
// => Star Wars, starring: Mark Hamill,Harrison Ford,Carrie Fisher

function Movie(title, stars) {
  this.title = title;
  this.stars = stars;
  this.year = (new Date()).getFullYear();
}
var kungFuMovie = new Movie('Kung Fu Panda', ['Jack Black', 'Angelina Jolie']);
console.log(kungFuMovie.title);
// => Kung Fu Panda
console.log(kungFuMovie.stars);
// => ['Jack Black', 'Angelina Jolie']
console.log(kungFuMovie.year);
// => 2018

Movie.prototype.logMe = function() {
  console.log(this.title + ', starring: ' + this.stars);
};
kungFuMovie.logMe();
// => Kung Fu Panda, starring: Jack Black,Angelina Jolie

setTimeout(kungFuMovie.logMe.bind(kungFuMovie), 500);
setTimeout(bestMovie.logMe.bind(bestMovie), 500);

// The Challenges :)

var bestMovie = {
  title: 'Star Wars',
  year: 1977,
```

```
    stars: ['Mark Hamill', 'Harrison Ford', 'Carrie Fisher'],
    logMe: function() {
      var me = this.about();
      console.log(me);
    },
    about: function() {
      return this.title + ', starring: ' + this.stars;
    },
    logFullTitle: function() {
      var title = this.fullTitle();
      console.log(title);
    },
    fullTitle: function() {
      return this.title + ' (' + this.year + ')';
    }
  };
  bestMovie.logMe();
  bestMovie.logFullTitle();

  function Movie(title, stars, date) {
    this.title = title;
    this.stars = stars;
    if (date) this.year = date;
    else this.year = (new Date()).getFullYear();
  }

  Movie.prototype.about = function() {
    return this.title + ', starring: ' + this.stars;
  };
  Movie.prototype.logMe = function() {
    var me = this.about();
    console.log(me);
  };
  var kungFuMovie = new Movie('Kung Fu Panda', ['Jack Black', 'Angelina Jolie'],
2008);
  console.log(kungFuMovie.year);
  kungFuMovie.logMe();
</script>
```

Code: Ready, Steady, Launch

This is the final version of the game code from Chapter 17, *Project: Ready, Steady, Launch*:

```
<body></body>
<script src="/three.js"></script>
<script src="/physi.js"></script>
<script src="/scoreboard.js"></script>

<script>
 // Physics settings
 Physijs.scripts.ammo = '/ammo.js';
 Physijs.scripts.worker = '/physijs_worker.js';

 // The "scene" is where stuff in our game will happen:
 var scene = new Physijs.Scene({ fixedTimeStep: 2 / 60 });
 scene.setGravity(new THREE.Vector3( 0, -100, 0 ));
 var flat = {flatShading: true};
 var light = new THREE.HemisphereLight('white', 'grey', 0.7);
 scene.add(light);

 // The "camera" is what sees the stuff:
 var width = window.innerWidth,
     height = window.innerHeight,
     aspectRatio = width / height;
 var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
 // var camera = new THREE.OrthographicCamera(
 //   -width/2, width/2, height/2, -height/2, 1, 10000
 // );
 camera.position.z = 500;
 camera.position.y = 200;
 camera.lookAt(new THREE.Vector3(0,0,0));
 scene.add(camera);

 // The "renderer" draws what the camera sees onto the screen:
 var renderer = new THREE.WebGLRenderer({antialias: true});
 renderer.setSize(window.innerWidth, window.innerHeight);
 document.body.appendChild(renderer.domElement);
 document.body.style.backgroundColor = '#ffffff';
 // ******** START CODING ON THE NEXT LINE ********
```

```
function Launcher() {
  this.angle = 0;
  this.power = 0;
  this.draw();
}
Launcher.prototype.draw = function() {
  var direction = new THREE.Vector3(0, 1, 0);
  var position = new THREE.Vector3(0, -100, 250);
  var length = 100;
  this.arrow = new THREE.ArrowHelper(
    direction,
    position,
    length,
    'yellow'
  );
  scene.add(this.arrow);
};
Launcher.prototype.vector = function() {
  return new THREE.Vector3(
    Math.sin(this.angle),
    Math.cos(this.angle),
    0
  );
};
Launcher.prototype.moveLeft = function(){
  this.angle = this.angle - Math.PI / 100;
  this.arrow.setDirection(this.vector());
};
Launcher.prototype.moveRight = function(){
  this.angle = this.angle + Math.PI / 100;
  this.arrow.setDirection(this.vector());
};
Launcher.prototype.powerUp = function(){
  if (this.power >= 100) return;
  this.power = this.power + 5;
  this.arrow.setLength(this.power);
};
Launcher.prototype.launch = function(){
  var shape = new THREE.SphereGeometry(10);
  var material = new THREE.MeshPhongMaterial({color: 'yellow'});
  var ball = new Physijs.SphereMesh(shape, material, 1);
  ball.name = 'Game Ball';
  ball.position.set(0,0,300);
  scene.add(ball);
```

```
  var speedVector = new THREE.Vector3(
    2.5 * this.power * this.vector().x,
    2.5 * this.power * this.vector().y,
   -80
  );
  ball.setLinearVelocity(speedVector);

  this.power = 0;
  this.arrow.setLength(100);
};

function Basket(size, points) {
  this.size = size;
  this.points = points;
  this.height = 100/Math.log10(size);

  var r = Math.random;
  this.color = new THREE.Color(r(), r(), r());

  this.draw();
}
Basket.prototype.draw = function() {
  var cover = new THREE.MeshPhongMaterial({
    color: this.color,
    shininess: 50,
    specular: 'white'
  });

  var shape = new THREE.CubeGeometry(this.size, 1, this.size);
  var goal = new Physijs.BoxMesh(shape, cover, 0);
  goal.position.y = this.height / 100;
  scene.add(goal);

  var halfSize = this.size/2;
  var halfHeight = this.height/2;

  shape = new THREE.CubeGeometry(this.size, this.height, 1);
  var side1 = new Physijs.BoxMesh(shape, cover, 0);
  side1.position.set(0, halfHeight, halfSize);
  scene.add(side1);

  var side2 = new Physijs.BoxMesh(shape, cover, 0);
  side2.position.set(0, halfHeight, -halfSize);
  scene.add(side2);
```

```
  shape = new THREE.CubeGeometry(1, this.height, this.size);

  var side3 = new Physijs.BoxMesh(shape, cover, 0);
  side3.position.set(halfSize, halfHeight, 0);
  scene.add(side3);

  var side4 = new Physijs.BoxMesh(shape, cover, 0);
  side4.position.set(-halfSize, halfHeight, 0);
  scene.add(side4);

  this.waitForScore(goal);
};
Basket.prototype.waitForScore = function(goal){
  goal.addEventListener('collision', this.score.bind(this));
};
Basket.prototype.score = function(ball){
  if (scoreboard.getTimeRemaining() == 0) return;
  scoreboard.addPoints(this.points);
  scene.remove(ball);
};

function Wind() {
  this.draw();
  this.change();
}
Wind.prototype.draw = function(){
  var dir = new THREE.Vector3(1, 0, 0);
  var start = new THREE.Vector3(0, 200, 250);
  this.arrow = new THREE.ArrowHelper(dir, start, 1, 'lightblue');
  scene.add(this.arrow);
};
Wind.prototype.change = function(){
  if (Math.random() < 0.5) this.direction = -1;
  else this.direction = 1;
  this.strength = 20*Math.random();

  this.arrow.setLength(5 * this.strength);
  this.arrow.setDirection(this.vector());

  setTimeout(this.change.bind(this), 10000);
};
Wind.prototype.vector = function(){
  var x = this.direction * this.strength;
  return new THREE.Vector3(x, 0, 0);
```

```
};

var launcher = new Launcher();


var scoreboard = new Scoreboard();
scoreboard.countdown(60);
scoreboard.score(0);
scoreboard.help(
  'Use right and left arrow keys to point the launcher. ' +
  'Press and hold the down arrow key to power up the launcher. ' +
  'Let go of the down arrow key to launch. ' +
  'Watch out for the wind!!!'
);
scoreboard.onTimeExpired(timeExpired);
function timeExpired() {
  scoreboard.message("Game Over!");
}

var goal1 = new Basket(200, 10);
var goal2 = new Basket(40, 100);

var wind = new Wind();

var light = new THREE.PointLight( 0xffffff, 1, 0 );
light.position.set(120, 150, -150);
scene.add(light);

function allBalls() {
  var balls = [];
  for (var i=0; i<scene.children.length; i++) {
    if (scene.children[i].name.startsWith('Game Ball')) {
      balls.push(scene.children[i]);
    }
  }
  return balls;
}

// Animate motion in the game
function animate() {
  if (scoreboard.getTimeRemaining() == 0) return;
  requestAnimationFrame(animate);
  renderer.render(scene, camera);
}
animate();
```

```
// Run physics
function gameStep() {
  if (scoreboard.getTimeRemaining() == 0) return;


  scene.simulate();

  var balls = allBalls();
  for (var i=0; i<balls.length; i++) {
    balls[i].applyCentralForce(wind.vector());
    if (balls[i].position.y < -100) scene.remove(balls[i]);
  }

  // Update physics 60 times a second so that motion is smooth
  setTimeout(gameStep, 1000/60);
}
gameStep();

function reset() {
  if (scoreboard.getTimeRemaining() > 0) return;
  scoreboard.score(0);
  scoreboard.countdown(60);

  var balls = allBalls();
  for (var i=0; i<balls.length; i++) {
    scene.remove(balls[i]);
  }

  animate();
  gameStep();
}

document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  var code = event.code;
  if (code == 'ArrowLeft') launcher.moveLeft();
  if (code == 'ArrowRight') launcher.moveRight();
  if (code == 'ArrowDown') launcher.powerUp();
  if (code == 'KeyR') reset();
}

document.addEventListener('keyup', sendKeyUp);
function sendKeyUp(event){
  var code = event.code;
```

```
    if (code == 'ArrowDown') launcher.launch();
  }
</script>
```

Code: Two-Player Ready, Steady, Launch

This is the final version of the game code from Chapter 18, *Project: Two-Player Games*:

```
<script src="/three.js"></script>
<script src="/physi.js"></script>
<script src="/scoreboard.js"></script>

<script>
 // Physics settings
 Physijs.scripts.ammo = '/ammo.js';
 Physijs.scripts.worker = '/physijs_worker.js';

 // The "scene" is where stuff in our game will happen:
 var scene = new Physijs.Scene({ fixedTimeStep: 2 / 60 });
 scene.setGravity(new THREE.Vector3( 0, -100, 0 ));
 var flat = {flatShading: true};
 var light = new THREE.HemisphereLight('white', 'grey', 0.7);
 scene.add(light);

 // The "camera" is what sees the stuff:
 var width = window.innerWidth,
     height = window.innerHeight,
     aspectRatio = width / height;
 var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
 // var camera = new THREE.OrthographicCamera(
 //   -width/2, width/2, height/2, -height/2, 1, 10000
 // );
 camera.position.z = 500;
 camera.position.y = 200;
 camera.lookAt(new THREE.Vector3(0,0,0));
 scene.add(camera);

 // The "renderer" draws what the camera sees onto the screen:
 var renderer = new THREE.WebGLRenderer({antialias: true});
 renderer.setSize(window.innerWidth, window.innerHeight);
 document.body.appendChild(renderer.domElement);
 document.body.style.backgroundColor = '#ffffff';

 // ******** START CODING ON THE NEXT LINE ********
```

```
function Launcher(location) {
  this.location = location;
  this.color = 'yellow';
  if (location == 'right') this.color = 'lightblue';
  this.angle = 0;
  this.power = 0;
  this.draw();
  this.keepScore();
}
Launcher.prototype.draw = function() {
  var direction = new THREE.Vector3(0, 1, 0);
  var x = 0;
  if (this.location == 'left') x = -100;
  if (this.location == 'right') x = 100;
  var position = new THREE.Vector3( x, -100, 250 );
  var length = 100;
  this.arrow = new THREE.ArrowHelper(
    direction,
    position,
    length,
    this.color
  );
  scene.add(this.arrow);
};
Launcher.prototype.vector = function() {
  return new THREE.Vector3(
    Math.sin(this.angle),
    Math.cos(this.angle),
    0
  );
};
Launcher.prototype.moveLeft = function(){
  this.angle = this.angle - Math.PI / 100;
  this.arrow.setDirection(this.vector());
};
Launcher.prototype.moveRight = function(){
  this.angle = this.angle + Math.PI / 100;
  this.arrow.setDirection(this.vector());
};
Launcher.prototype.powerUp = function(){
  if (this.power >= 100) return;
  this.power = this.power + 5;
  this.arrow.setLength(this.power);
};
Launcher.prototype.launch = function(){
```

```
  var shape = new THREE.SphereGeometry(10);

  var material = new THREE.MeshPhongMaterial({color: this.color});
  var ball = new Physijs.SphereMesh(shape, material, 1);
  ball.name = 'Game Ball';
  ball.scoreboard = this.scoreboard;
  var p = this.arrow.position;
  ball.position.set(p.x, p.y, p.z);
  scene.add(ball);

  var speedVector = new THREE.Vector3(
    2.5 * this.power * this.vector().x,
    2.5 * this.power * this.vector().y,
   -80
  );
  ball.setLinearVelocity(speedVector);

  this.power = 0;
  this.arrow.setLength(100);
};
Launcher.prototype.keepScore = function(){
  var scoreboard = new Scoreboard('top' + this.location);
  scoreboard.countdown(60);
  scoreboard.score(0);

  var moveKeys;
  if (this.location == 'left') moveKeys = 'A and D';
  if (this.location == 'right') moveKeys = 'J and L';

  var launchKeys;
  if (this.location == 'left') launchKey = 'S';
  if (this.location == 'right') launchKey = 'K';

  scoreboard.help(
    'Use the ' + moveKeys + ' keys to point the launcher. ' +
    'Press and hold the ' + launchKey + ' key to power up the launcher. ' +
    'Let go of the ' + launchKey + ' key to launch. ' +
    'Watch out for the wind!!!'
  );
  scoreboard.onTimeExpired(timeExpired);
  function timeExpired() {
    scoreboard.message("Game Over!");
  }
  this.scoreboard = scoreboard;
};
```

```
Launcher.prototype.reset = function(){
  var scoreboard = this.scoreboard;

  if (scoreboard.getTimeRemaining() > 0) return;
  scoreboard.score(0);
  scoreboard.countdown(60);
};

function Basket(size, points) {
  this.size = size;
  this.points = points;
  this.height = 100/Math.log10(size);

  var r = Math.random;
  this.color = new THREE.Color(r(), r(), r());

  this.draw();
}
Basket.prototype.draw = function() {
  var cover = new THREE.MeshPhongMaterial({
    color: this.color,
    shininess: 50,
    specular: 'white'
  });

  var shape = new THREE.CubeGeometry(this.size, 1, this.size);
  var goal = new Physijs.BoxMesh(shape, cover, 0);
  goal.position.y = this.height / 100;
  scene.add(goal);

  var halfSize = this.size/2;
  var halfHeight = this.height/2;

  shape = new THREE.CubeGeometry(this.size, this.height, 1);
  var side1 = new Physijs.BoxMesh(shape, cover, 0);
  side1.position.set(0, halfHeight, halfSize);
  scene.add(side1);

  var side2 = new Physijs.BoxMesh(shape, cover, 0);
  side2.position.set(0, halfHeight, -halfSize);
  scene.add(side2);

  shape = new THREE.CubeGeometry(1, this.height, this.size);
  var side3 = new Physijs.BoxMesh(shape, cover, 0);
  side3.position.set(halfSize, halfHeight, 0);
```

```
  scene.add(side3);

  var side4 = new Physijs.BoxMesh(shape, cover, 0);

  side4.position.set(-halfSize, halfHeight, 0);
  scene.add(side4);

  this.waitForScore(goal);
};
Basket.prototype.waitForScore = function(goal){
  goal.addEventListener('collision', this.score.bind(this));
};
Basket.prototype.score = function(ball){
  var scoreboard = ball.scoreboard;
  if (scoreboard.getTimeRemaining() == 0) return;
  scoreboard.addPoints(this.points);
  scene.remove(ball);
};

function Wind() {
  this.draw();
  this.change();
}
Wind.prototype.draw = function(){
  var dir = new THREE.Vector3(1, 0, 0);
  var start = new THREE.Vector3(0, 200, 250);
  this.arrow = new THREE.ArrowHelper(dir, start, 1, 'lightblue');
  scene.add(this.arrow);
};
Wind.prototype.change = function(){
  if (Math.random() < 0.5) this.direction = -1;
  else this.direction = 1;
  this.strength = 20*Math.random();

  this.arrow.setLength(5 * this.strength);
  this.arrow.setDirection(this.vector());

  setTimeout(this.change.bind(this), 10000);
};
Wind.prototype.vector = function(){
  var x = this.direction * this.strength;
  return new THREE.Vector3(x, 0, 0);
};

var launcher1 = new Launcher('left');
```

```
var launcher2 = new Launcher('right');
var scoreboard = launcher1.scoreboard;

var goal1 = new Basket(200, 10);

var goal2 = new Basket(40, 100);

var wind = new Wind();

var light = new THREE.PointLight( 0xffffff, 1, 0 );
light.position.set(120, 150, -150);
scene.add(light);

function allBalls() {
  var balls = [];
  for (var i=0; i<scene.children.length; i++) {
    if (scene.children[i].name.startsWith('Game Ball')) {
      balls.push(scene.children[i]);
    }
  }
  return balls;
}

// Animate motion in the game
function animate() {
  if (scoreboard.getTimeRemaining() == 0) return;
  requestAnimationFrame(animate);
  renderer.render(scene, camera);
}
animate();

// Run physics
function gameStep() {
  if (scoreboard.getTimeRemaining() == 0) return;

  scene.simulate();

  var balls = allBalls();
  for (var i=0; i<balls.length; i++) {
    balls[i].applyCentralForce(wind.vector());
    if (balls[i].position.y < -100) scene.remove(balls[i]);
  }

  // Update physics 60 times a second so that motion is smooth
  setTimeout(gameStep, 1000/60);
```

```
}
gameStep();

function reset() {
  if (scoreboard.getTimeRemaining() > 0) return;


  launcher1.reset();
  launcher2.reset();

  var balls = allBalls();
  for (var i=0; i<balls.length; i++) {
    scene.remove(balls[i]);
  }

  animate();
  gameStep();
}

var powerUp1;
var powerUp2;
function powerUpLauncher1(){ launcher1.powerUp(); }
function powerUpLauncher2(){ launcher2.powerUp(); }

document.addEventListener('keydown', sendKeyDown);
function sendKeyDown(event) {
  if (event.repeat) return;

  var code = event.code;
  if (code == 'KeyA') launcher1.moveLeft();
  if (code == 'KeyD') launcher1.moveRight();
  if (code == 'KeyS') {
    clearInterval(powerUp1);
    powerUp1 = setInterval(powerUpLauncher1, 20);
  }

  if (code == 'KeyJ') launcher2.moveLeft();
  if (code == 'KeyL') launcher2.moveRight();
  if (code == 'KeyK') {
    clearInterval(powerUp2);
    powerUp2 = setInterval(powerUpLauncher2, 20);
  }

  if (code == 'KeyR') reset();
}
```

```
 document.addEventListener('keyup', sendKeyUp);
 function sendKeyUp(event){
   var code = event.code;
   if (code == 'KeyS') {
     launcher1.launch();

     clearInterval(powerUp1);
   }
   if (code == 'KeyK') {
     launcher2.launch();
     clearInterval(powerUp2);
   }
 }
</script>
```

Code: River Rafter

This is the final version of the game code from Chapter 19, *Project: River Rafter*. It is very long. It includes a few extras to play around with, as well.

```
<body></body>
<script src="/three.js"></script>
<script src="/physi.js"></script>
<script src="/controls/OrbitControls.js"></script>
<script src="/scoreboard.js"></script>
<script src="/noise.js"></script>
<script>
 // Physics settings
 Physijs.scripts.ammo = '/ammo.js';
 Physijs.scripts.worker = '/physijs_worker.js';

 // The "scene" is where stuff in our game will happen:
 var scene = new Physijs.Scene();
 scene.setGravity(new THREE.Vector3( 0, -10, 0 ));
 var flat = {flatShading: true};
 var light = new THREE.AmbientLight('white', 0.2);
 scene.add(light);

 var sunlight = new THREE.DirectionalLight('white', 0.8);
 sunlight.position.set(4, 6, 0);
 sunlight.castShadow = true;
 scene.add(sunlight);
 var d = 10;
 sunlight.shadow.camera.left = -d;
 sunlight.shadow.camera.right = d;
 sunlight.shadow.camera.top = d;
 sunlight.shadow.camera.bottom = -d;

 // The "camera" is what sees the stuff:
 var aspectRatio = window.innerWidth / window.innerHeight;
 var camera = new THREE.PerspectiveCamera(75, aspectRatio, 0.1, 100);
 camera.position.set(-8, 8, 8);
 scene.add(camera);

// The "renderer" draws what the camera sees onto the screen:
var renderer = new THREE.WebGLRenderer({antialias: true});
renderer.setClearColor('skyblue');
```

```
renderer.shadowMap.enabled = true;
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

// new THREE.OrbitControls(camera, renderer.domElement);

// ******** START CODING ON THE NEXT LINE ********

var gameOver;
var ground = addGround();
var water = addWater();
var scoreboard = addScoreboard();
var raft = addRaft();
reset();

function addGround() {
  var faces = 99;
  var shape = new THREE.PlaneGeometry(10, 20, faces, faces);

  var riverPoints = [];
  var numVertices = shape.vertices.length;
  var noiseMaker = new SimplexNoise();
  for (var i=0; i<numVertices; i++) {
    var vertex = shape.vertices[i];
    var noise = 0.25 * noiseMaker.noise(vertex.x, vertex.y);
    vertex.z = noise;
  }

  for (var j=50; j<numVertices; j+=100) {
    var curve = 20 * Math.sin(7*Math.PI * j/numVertices);
    var riverCenter = j + Math.floor(curve);
    riverPoints.push(shape.vertices[riverCenter]);

    for (var k=-20; k<20; k++) {
      shape.vertices[riverCenter + k].z = -1;
    }
  }

  shape.computeFaceNormals();
  shape.computeVertexNormals();

  var _cover = new THREE.MeshPhongMaterial({color: 'green', shininess: 0});
  var cover = Physijs.createMaterial(_cover, 0.8, 0.1);

  var mesh = new Physijs.HeightfieldMesh(shape, cover, 0);
```

```
  mesh.rotation.set(-0.475 * Math.PI, 0, 0);

  mesh.receiveShadow = true;
  mesh.castShadow = true;
  mesh.riverPoints = riverPoints;

  scene.add(mesh);
  return mesh;
}

function addWater() {
  var shape = new THREE.CubeGeometry(10, 20, 1);
  var _cover = new THREE.MeshPhongMaterial({color: 'blue'});
  var cover = Physijs.createMaterial(_cover, 0, 0.6);

  var mesh = new Physijs.ConvexMesh(shape, cover, 0);
  mesh.rotation.set(-0.475 * Math.PI, 0, 0);
  mesh.position.y = -0.8;
  mesh.receiveShadow = true;
  scene.add(mesh);

  return mesh;
}

function addScoreboard() {
  var scoreboard = new Scoreboard();
  scoreboard.score(0);
  scoreboard.timer();
  scoreboard.help(
    'left / right arrow keys to turn. ' +
    'space bar to move forward. ' +
    'R to restart.'
  );
  return scoreboard;
}

function addRaft() {
  var shape = new THREE.TorusGeometry(0.1, 0.05, 8, 20);
  var _cover = new THREE.MeshPhongMaterial({visible: false});
  var cover = Physijs.createMaterial(_cover, 0.4, 0.6);
  var mesh = new Physijs.ConvexMesh(shape, cover, 0.25);
  mesh.rotation.x = -Math.PI/2;

  cover = new THREE.MeshPhongMaterial({color: 'orange'});
  var tube = new THREE.Mesh(shape, cover);
```

```
  tube.position.z = -0.08;
  tube.castShadow = true;

  mesh.add(tube);
  mesh.tube = tube;

  shape = new THREE.SphereGeometry(0.02);
  cover = new THREE.MeshBasicMaterial({color: 'white'});
  var rudder = new THREE.Mesh(shape, cover);
  rudder.position.set(0.15, 0, 0);
  tube.add(rudder);

  scene.add(mesh);
  mesh.setAngularFactor(new THREE.Vector3(0, 0, 0));
  return mesh;
}

function reset() {
  resetPowerUps();

  camera.position.set(0,-1,2);
  camera.lookAt(new THREE.Vector3(0, 0, 0));
  raft.add(camera);

  scoreboard.message('');
  scoreboard.resetTimer();
  scoreboard.score(0);

  raft.__dirtyPosition = true;
  raft.position.set(0.75, 2, -9.6);
  raft.setLinearVelocity(new THREE.Vector3(0, 0, 0));

  gameOver = false;
  animate();
  scene.onSimulationResume();
  gameStep();
}

function resetPowerUps() {
  removeOldPowerUps();

  var random20 = 20 + Math.floor(10*Math.random());
  var p20 = ground.riverPoints[random20];
  addPowerUp(p20);
```

```
  var random70 = 70 + Math.floor(10*Math.random());
  var p70 = ground.riverPoints[random70];
  addPowerUp(p70);
}

function addPowerUp(riverPoint) {
  ground.updateMatrixWorld();
  var x = riverPoint.x + 4 * (Math.random() - 0.5);
  var y = riverPoint.y;
  var z = -0.5;
  var p = new THREE.Vector3(x, y, z);
  ground.localToWorld(p);

  var shape = new THREE.SphereGeometry(0.25, 25, 18);
  var cover = new THREE.MeshNormalMaterial();
  var mesh = new Physijs.SphereMesh(shape, cover, 0);
  mesh.position.copy(p);
  mesh.powerUp = true;
  scene.add(mesh);

  mesh.addEventListener('collision', function() {
    for (var i=0; i<scene.children.length; i++) {
      var obj = scene.children[i];
      if (obj == mesh) scene.remove(obj);
    }
    scoreboard.addPoints(200);
    scoreboard.message('Yum!');
    setTimeout(function() {scoreboard.clearMessage();}, 5*1000);
  });

  return mesh;
}

function removeOldPowerUps() {
  var last = scene.children.length - 1;
  for (var i=last; i>=0; i--) {
    var obj = scene.children[i];
    if (obj.powerUp) scene.remove(obj);
  }
}

// Animate motion in the game
function animate() {
  if (gameOver) return;
  requestAnimationFrame(animate);
```

```
  renderer.render(scene, camera);
}
// animate();

// Run physics
function gameStep() {
  if (gameOver) return;
  checkForGameOver();
  scene.simulate();
  // Update physics 60 times a second so that motion is smooth
  setTimeout(gameStep, 1000/60);
}
// gameStep();

function checkForGameOver() {
  if (raft.position.z > 9.8) {
    gameOver = true;
    scoreboard.stopTimer();
    scoreboard.message("You made it!");
  }

  if (scoreboard.getTime() > 60) {
    gameOver = true;
    scoreboard.stopTimer();
    scoreboard.message("Time's up. Too slow :(");
  }

   if (gameOver) {
     var score = Math.floor(61-scoreboard.getTime());
     scoreboard.addPoints(score);

     if (scoreboard.getTime() < 40) scoreboard.addPoints(100);
     if (scoreboard.getTime() < 30) scoreboard.addPoints(200);
     if (scoreboard.getTime() < 20) scoreboard.addPoints(500);
   }
 }

 document.addEventListener('keydown', sendKeyDown);
 function sendKeyDown(event) {
   var code = event.code;
   if (code == 'ArrowLeft') rotateRaft(1);
   if (code == 'ArrowRight') rotateRaft(-1);
   if (code == 'ArrowDown') pushRaft();
   if (code == 'Space') pushRaft();
   if (code == 'KeyR') reset();
```

```
  }

  function rotateRaft(direction) {
    raft.tube.rotation.z = raft.tube.rotation.z + direction * Math.PI/10;
  }

  function pushRaft() {
    var angle = raft.tube.rotation.z;
    var force = new THREE.Vector3(Math.cos(angle), 0, -Math.sin(angle));

    raft.applyCentralForce(force);
  }
</script>
```

Code: Getting Code on the Web

This is the code used to create the Blogger post from Chapter 20, *Getting Code on the Web*. You can find the completed Blogger post at: http://code3Dgames.blogspot.com/2018/02/amazing-3d-animation.html.

```
<p>I made this!</p>
<div id="3d-code-2018-12-31">
</div>
<p>It's in the first chapter of
  <a href="http://code3Dgames.com/">
    3D Game Programming for Kids, second edition</a>.
</p>
<script src="https://code3Dgames.com/three.js"></script>
<script src="https://code3Dgames.com/controls/OrbitControls.js"></script>
<script>
 // The "scene" is where stuff in our game will happen:
 var scene = new THREE.Scene();
 var flat = {flatShading: true};
 var light = new THREE.AmbientLight('white', 0.8);
 scene.add( light );

 // The "camera" is what sees the stuff:
 var aspectRatio = 4/3;
 var camera = new THREE.PerspectiveCamera(75, aspectRatio, 1, 10000);
 camera.position.z = 500;
 scene.add(camera);

 // The "renderer" draws what the camera sees onto the screen:
 var renderer = new THREE.WebGLRenderer({antialias: true});
 var container = document.getElementById('3d-code-2018-12-31');
 container.appendChild(renderer.domElement);

 function resizeRenderer(){
   var width = container.clientWidth * 0.96;
   var height = width/aspectRatio;
   renderer.setSize(width, height);
 }
 resizeRenderer();
 window.addEventListener('resize', resizeRenderer, false);

 new THREE.OrbitControls(camera, renderer.domElement);
```

```
// ******** START CODING ON THE NEXT LINE ********

var shape = new THREE.SphereGeometry(100, 20, 15);
var cover = new THREE.MeshNormalMaterial(flat);
var ball = new THREE.Mesh(shape, cover);
scene.add(ball);
ball.position.set(-250,250,-250);

var shape = new THREE.CubeGeometry(300, 100, 100);
var cover = new THREE.MeshNormalMaterial(flat);
var box = new THREE.Mesh(shape, cover);
scene.add(box);
box.rotation.set(0.5, 0.5, 0);
box.position.set(250, 250, -250);

var shape = new THREE.CylinderGeometry(1, 100, 100, 4);
var cover = new THREE.MeshNormalMaterial(flat);
var tube = new THREE.Mesh(shape, cover);
scene.add(tube);
tube.rotation.set(0.5, 0, 0);
tube.position.set(250, -250, -250);

var shape = new THREE.PlaneGeometry(100, 100);
var cover = new THREE.MeshNormalMaterial(flat);
var ground = new THREE.Mesh(shape, cover);
scene.add(ground);
ground.rotation.set(0.5, 0, 0);
ground.position.set(-250, -250, -250);

var shape = new THREE.TorusGeometry(100, 25, 8, 25);
var cover = new THREE.MeshNormalMaterial(flat);
var donut = new THREE.Mesh(shape, cover);
scene.add(donut);

var clock = new THREE.Clock();

function animate() {
  requestAnimationFrame(animate);
  var t = clock.getElapsedTime();

  ball.rotation.set(t, 2*t, 0);
  box.rotation.set(t, 2*t, 0);
  tube.rotation.set(t, 2*t, 0);
  ground.rotation.set(t, 2*t, 0);
```

```
    donut.rotation.set(t, 2*t, 0);

    renderer.render(scene, camera);
  }

  animate();

  // Now, show what the camera sees on the screen:
  renderer.render(scene, camera);
</script>
```

Appendix 2

JavaScript Code Collections Used in This Book

This appendix contains a list of the JavaScript code collections used in this book and details for how you can find more information about each. All of the code libraries used in this book are free and open source, meaning that the people that wrote the code want you to use it however you like. The only rule is that you have to give people credit for their work—like we do here!

Three.js

Three.js is the main code collection used throughout this book.[11] The home page for the project includes lots of cool animations and samples, many of which you can try in the 3DE Code Editor.

We're using version 87 of Three.js. Detailed documentation for properties and methods not discussed in this book is available online.[12]

Physijs

The physics engine we use in this book is Physijs.[13] The Physijs web page includes brief samples and some introductory articles.

The Physijs project doesn't have as much documentation as the Three.js project, but some documentation exists for the project wiki.[14] We're using the version of Physijs that is compatible with Three.js r87. Since Physijs continues to grow, the wiki may refer to newer features than those supported by the version we're using.

Controls

The two controls we use in this book are the fly controls and orbit controls.

Fly Controls

We used `FlyControls.js` to fly around space in both Chapter 5, *Functions: Use and Use Again* and Chapter 13, *Project: Phases of the Moon*. We can create fly controls after the camera is defined with this:

```
var controls = new THREE.FlyControls(camera);
```

Then the following keys let you fly through the scene:

Motion	Direction	Keys
Move	Forward / Backward	W / S
Move	Left / Right	A / D
Move	Up / Down	R / F
Spin	Clockwise / Counterclockwise	Q / E
Spin	Left / Right	Left Arrow / Right Arrow
Spin	Up / Down	Up Arrow / Down Arrow

The following options are available for fly controls:

```
controls.movementSpeed = 100;
controls.rollSpeed = 0.5;
controls.dragToLook = true;
controls.autoForward = false;
```

How fast you move is controlled by `movementSpeed`. The spin speed is controlled by `rollSpeed`. Clicking and dragging the mouse in the scene will move the camera if `dragToLook` is set to `true`. If `autoForward` is `true`, then the camera automatically flies forward without pressing any keys.

Orbit Controls

The other kind of controls are the **OrbitControls.js**, which let us spin the camera around the center of the scene. We used these controls to get a better look at things as we built them in Chapter 12, *Working with Lights and Materials* and in Chapter 19, *Project: River Rafter*.

After the camera and renderer are defined, we can create orbit controls with the following:

```
var controls = new THREE.OrbitControls( camera, renderer.domElement );
```

These controls let us click and drag the scene. Scrolling with the mouse or touchpad will zoom in and out. The arrow keys move up, down, left, and right.

Noise

We used noise.js to create uneven terrain in Chapter 19, *Project: River Rafter*. The noise code collection is bonus code included in Three.js. You can use it on any shape you like. All we have to do is count the number of vertices in the shape, create a noise maker, then loop over the vertices adding noise to each. The example from the river rafter is a good place to start:

```
var numVertices = shape.vertices.length;
var noiseMaker = new SimplexNoise();
for (var i=0; i<numVertices; i++) {
  var vertex = shape.vertices[i];
  var noise = 0.25 * noiseMaker.noise(0.1 * vertex.x, 0.1 * vertex.y);
  vertex.z = noise;
}
```

You can play with the numbers on the line that sets `noise`. Try increasing and decreasing both the `0.25` number at the start and the `0.1` numbers inside the `noise()` method.

Scoreboard.js

The Scoreboard.js code collection is *simple* JavaScript code that provides the basics of scoring in games.[15] It supports very little configuration, but aims to be easy to use for programmers.

The Scoreboard.js code collection supports messages, help text, scoring, an elapsed timer, and a countdown timer. Each of these can be shown, hidden, reset, and updated.

Shader Particle Engine

The spe.js code collection is a particle engine, which creates nifty effects like star fields, fire, clouds, and more.[16] We used spe.js to create fire in Chapter 15, *Project: Tilt-a-Board* when someone scored a goal.

There are a *lot* of options for spe.js. If you're interested in playing more with spe.js, check out some of the examples from the home page and view the HTML source with Ctrl+U or Command+U. You should be able to copy that code into 3DE to play with it.

Sounds.js

The Sounds.js JavaScript code collection contains the bare minimum of sounds for use in games.[17]

To use the Sounds.js code collection, it must be sourced in a `<script>` tag:

```
<script src="http://code3Dgames.com/sounds.js"></script>
```

At the time of this writing, eleven sounds were available: bubble, buzz, click, donk, drip, guitar, knock, scratch, snick, spring, and swish. Each sound can be played with code similar to the following:

```
Sounds.bubble.play();
```

To make a sound repeat, replace the `play` method with `repeat`:

```
Sounds.bubble.repeat();
```

To stop the sound at a later time, call the `stop` method:

```
Sounds.bubble.stop();
```

If you want a sound to repeat for a fixed length of time, then start a repeating sound with a timeout to stop the sound:

```
Sounds.bubble.repeat();
setTimeout(function(){Sounds.bubble.stop();}, 5*1000);
```

The preceding code would start repeated bubble sounds. After five seconds, the timeout function runs, stopping the repeating bubble sounds.

Tween.js

In this book, when we wanted to change values (location, rotation, speed) over the course of time, we used the Tween code collection.[18]

Building a Tween involves several parts. A Tween needs the starting value or values, the ending values, the time that it takes to move from the start to the end values, and a function that's called as the Tween is running. A Tween also needs to be started and updated to work.

The Tween from Chapter 11, *Project: Fruit Hunt*, contains a really good example.

```
new TWEEN.
  Tween({
    height: 150,
    spin: 0
  }).
  to({
    height: 250,
    spin: 4
  }, 500).
  onUpdate(function () {
    fruit.position.y = this.height;
    fruit.rotation.z = this.spin;
  }).
  start();
```

This moves between two values: the height and the spin. Over the course of half a second (500 milliseconds), the height moves from 150 to 250. The spin moves from 0 to 4. Each time the Tween is updated, we change the position and rotation of the fruit being animated. The current values that are being Tweened are made available as a property of the special `this` object.

The last thing we do in the preceding example is to start the Tween.

Tweens also need something to tell them to update. In 3D programming, we normally do this in the `animate` function with a `TWEEN.update` call.

```
function animate() {
  requestAnimationFrame(animate);
  TWEEN.update();
  renderer.render(scene, camera);
}
```

In addition to `onUpdate` are the `onStart` and `onComplete` methods, which call a function when the Tween starts and finishes.

Footnotes

[11] http://threejs.org/

[12] http://code3Dgames.com/docs/threejs/

[13] http://chandlerprall.github.io/Physijs/

[14] https://github.com/chandlerprall/Physijs/wiki

[15] https://github.com/eee-c/scoreboard.js

[16] https://eee-c.github.io/ShaderParticleEngine/

[17] https://github.com/eee-c/Sounds.js

[18] https://github.com/tweenjs/tween.js

Bibliography

[Ada95] Douglas Adams. ***The Hitchhiker's Guide to the Galaxy***. Ballantine Books, New York, NY, 1995.

Thank you!
How did you enjoy this book? Please let us know. Take a moment to email us at support@pragprog.com with your feedback. Tell us your story and you could wir free ebooks. Please use the subject line "Book Feedback."

Ready for your next great Pragmatic Bookshelf book? Come on over to https://pragprog.com and use the coupon code BUYANOTHER2018 to save 30% your next ebook.

Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks ne water. If rash persists, see a doctor. Doesn't apply to *The Pragmatic Programmer* ebook because it's older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfact Increase dosage regularly.

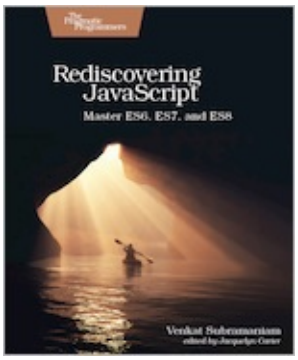
And thank you for your continued support,

Andy Hunt, Publisher

You May Be Interested In…

Select a cover for more information

Rediscovering JavaScript

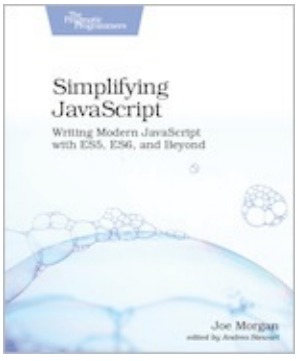


JavaScript is no longer to be feared or loathed—the world's most popular and ubiquitous language has evolved into a respectable language. Whether you're writing frontend applications or server-side code, the phenomenal features from ES6 and beyond—like the rest operator, generators, destructuring, object literals, arrow functions, modern classes, promises, async, and metaprogramming capabilities—will get you excited and eager to program with JavaScript. You've found the right book to get started quickly and dive deep into the essence of modern JavaScript. Learn practical tips to apply the elegant parts of the language and the gotchas to avoid.

Venkat Subramaniam

(286 pages) ISBN: 9781680505467 $45.95

Simplifying JavaScript



The best modern JavaScript is simple, readable, and predictable. Learn to write modern JavaScript not by memorizing a list of new syntax, but with practical examples of how syntax changes can make code more expressive. Starting from variable declarations that communicate intention clearly, see how modern principles can improve all parts of code. Incorporate ideas with curried functions, array methods, classes, and more to create code that does more with less while yielding fewer bugs.

Joe Morgan

(282 pages) ISBN: 9781680502886 $47.95

The Way of the Web Tester

This book is for everyone who needs to test the web. As a tester, you'll automate your tests. As a developer, you'll build more robust solutions. And as a team, you'll gain a vocabulary and a means to coordinate how to write and organize automated tests for the web. Follow the testing pyramid and level up your skills in user interface testing, integration testing, and unit testing. Your new skills will free you up to do other, more important things while letting the computer do the one thing it's really good at: quickly running thousands of repetitive tasks.

Jonathan Rasmusson

(256 pages) ISBN: 9781680501834 $29

Test-Driving JavaScript Applications

Debunk the myth that JavaScript is not easily testable. Whether you use Node.js, Express, MongoDB, jQuery, AngularJS, or directly manipulate the DOM, you can test-drive JavaScript. Learn the craft of writing meaningful, deterministic automated tests with Karma, Mocha, and Chai. Test asynchronous JavaScript, decouple and properly mock out dependencies, measure code coverage, and create lightweight modular designs of both server-side and client-side code. Your investment in writing tests will pay high dividends as you create code that's predictable and cost-effective to change.

Venkat Subramaniam

(362 pages) ISBN: 9781680501742 $38