The Pragmatic Programmers

3D Game Programming for Kids

Create Interactive Worlds With JavaScript

Chris Strom

Edited by Fahmida Y. Rashid

3D Game Programming for Kids

Create Interactive Worlds with JavaScript

Chris Strom

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

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*.

Printed in the United States of America.
ISBN-13: 978-1-937785-44-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—February 6, 2013

Introduction

Welcome to the world of programming!

I won't lie, it can be a frustrating world sometimes (it makes me cry at least once a week). But it is 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 eager hands is a great way to get started programming. It is chock full of clear and understandable explanations along the way. Best of all, we get to make some pretty cool games.

This is going to be a blast.

How I (the author) Learned to Program

When I was a kid, I copied computer program games out of books. This was a long time ago, so I bought books with nothing but programs, and typed them into computers.

When I first started doing it, I had no idea what I was doing. Eventually, I started to recognize certain things that were done over and over and almost understood them.

I started to change things—little things at first—to see what happened. Then I started making bigger changes. Eventually, I got pretty good at it. And after a long time, I could write my own programs.

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

What You Need for this Book

Not all Web browsers can generate the cool 3D gaming objects that we will build in this book. To get the most out of the book, you should install Google

Chrome[1] http://www.google.com/chrome. Other browsers will work, but some of the exercises in this book rely on features available only in Google Chrome. One browser that will definitely not work with the exercises is Internet Explorer.

For most of the exercises in the book, any old computer with Google Chrome installed will be sufficient. Later exercises that make use of interesting lighting, shadows and 3D materials, will need a computer capable of performing WebGL. You can find out if your computer supports WebGL by visiting the Get WebGL Site[2] http://get.webgl.org/. Don't worry too much about WebGL, there is still a ton of programming you will be able to do if your computer can't handle it.

What is JavaScript?

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

In this book, we will use the JavaScript programming language. We program in JavaScript because it is the language of the Web. It is the only programming language all Web browsers understand without needing any additional software. If you can program in JavaScript, not only can you make the kinds of games that we will learn in this book, but you can program just about every website there is.

We are not going to become experts in JavaScript.

We will learn just enough JavaScript to be able to program the games in this book. It turns out that is quite a lot of JavaScript—enough to be ready to learn the rest without too much difficulty.

How to Read this Book

There are two different kinds of chapters in this book: project chapters and learning chapters. The project chapters all start with "Project" like Chapter 1, *Project: Creating Simple Shapes*, on page ?.

If you want to learn programming the way that I did, just read the project chapters and follow along with all the projects in them. You'll create pretty cool game players and worlds to play in. You'll make spaceships. You'll make all sorts of cool stuff.

If you have questions about *why* the games are written the way they are, then read the learning chapters. We won't go over *everything* about programming,

1. https://www.google.com/chrome/
2. http://get.webgl.org/

but it should be enough to understand why we do what we do. These are the chapters that I wish I had when I was a kid.

Let's Get Started!

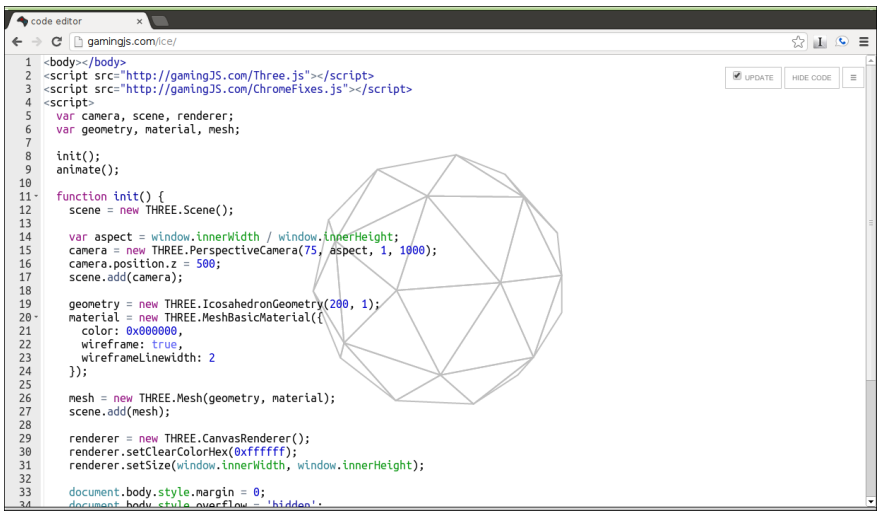
Enough introduction, let's jump right into programming!

There will be plenty of time for learning later in this book. For now, let's get started with programming!

1.1 Programming with the ICE Code Editor

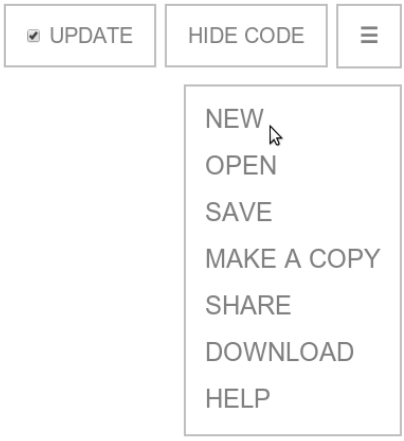
In this book, we will be using the ICE Code Editor to do our programming. The ICE Code Editor runs right inside your browser. It lets us type in our programming code and see the results right away.

To get started, open the ICE Code Editor at http://gamingJS.com/ice using Google's Chrome Web browser. It should look something like this:

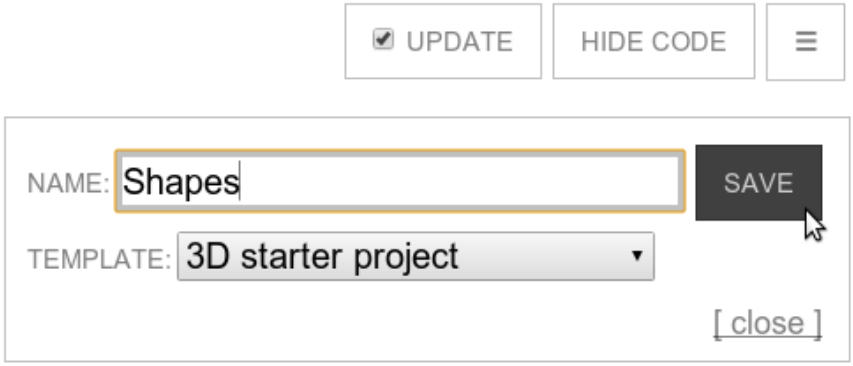


That spinning, multi-sided thing is just a sample of some of the stuff that we will be working on in this book. In this chapter, we will create a new project named Shapes.

To create a new project in the ICE Code Editor, we click on the menu button (the white button with three horizontal lines) on the upper right hand corner of the screen and select New from the dropdown.



Type the name of the project, Shapes, in the text field and click the Save button. Leave the template set as 3D Starter Project.

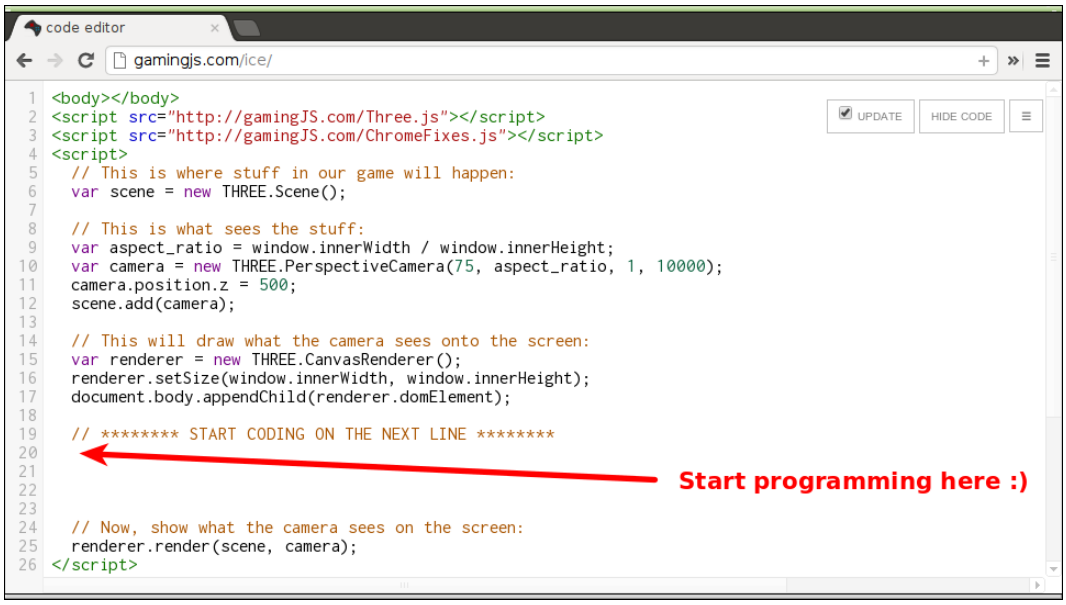


Remember, none of the projects in this book will work if you are using ICE Code Editor on Internet Explorer. While some of the exercises will work with Mozilla Firefox, it is easier to just stick with a single browser (Google Chrome) for all our projects.

Coding with the ICE Code Editor

We will be using the ICE Code Editor throughout this book. You only need web access the first time that you connect to http://gamingJS.com/ice/. After the first visit, ICE is stored in your browser so you can keep working even if you are not connected to the Internet!

When ICE opens a new 3D project, there is already a lot of code in the file. We will look closely at that code later, but for now, let's begin our programming adventure on line 20. Look for the line that says, "START CODING ON THE NEXT LINE":




```
<body></body>
<script src="http://gamingJS.com/Three.js"></script>
<script src="http://gamingJS.com/ChromeFixes.js"></script>
<script>
  // This is where stuff in our game will happen:
  var scene = new THREE.Scene();

  // This is what sees the stuff:
  var aspect_ratio = window.innerWidth / window.innerHeight;
  var camera = new THREE.PerspectiveCamera(75, aspect_ratio, 1, 10000);
  camera.position.z = 500;
  scene.add(camera);

  // This will draw what the camera sees onto the screen:
  var renderer = new THREE.CanvasRenderer();
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);

  // ******** START CODING ON THE NEXT LINE ********


  // Now, show what the camera sees on the screen:
  renderer.render(scene, camera);
</script>
```

On line 20, type the following:

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

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

```
  var camera = new THREE.PerspectiveCamera(75, aspect_ratio,
  camera.position.z = 500;
  scene.add(camera);

  // This will draw what the camera sees onto the screen:
  var renderer = new THREE.CanvasRenderer();
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);

  // ******** START CODING ON THE NEXT LINE ********
  var shape = new THREE.SphereGeometry(100);
  var wrapper = new THREE.MeshNormalMaterial();
  var ball = new THREE.Mesh(shape, wrapper);
  scene.add(ball);
```

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

Don't worry too much about the structure of the code just yet; we will get familiar with it in [xxx](#chapter.javascript_basics) For now, let's take a closer look at the 3D programming that we just did.

3D things are built from two parts: the shape and a wrapper that covers the shape. These two things, the shape and its wrapper, are given a special name in 3D programming: *Mesh*.

A mesh is a fancy word for 3D things. Meshes need shapes (sometimes called *geometry*) and something to cover them (sometimes called *material*).

In this chapter we are going to look at different shapes. We will deal with different covers for our shapes until [xxx](#chapter.lights_and_materials).

Your work is automatically saved

Your work is automatically saved, so you do not have to do it yourself. If you want to save the code yourself anyway, click the white menu button in ICE and select the Save option from the dropdown. That's it!

1.2 Making Shapes with JavaScript

So far, we have seen only one kind of shape: the sphere. Shapes can be simple like cubes, pyramids, cones, and spheres. Shapes can also be more complex things like faces or cars. In this book, we are going to stick with simple shapes. When we build things like trees, we will combine simple shapes, such as spheres and cylinders, to make them.

Creating Spheres

Balls are always called spheres in geometry and in 3D programming. There are two ways to control the shape of a sphere in JavaScript.

Size: SphereGeometry(100)

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

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

❶ This just points to where you should change the sphere's size.

This should make it much bigger:

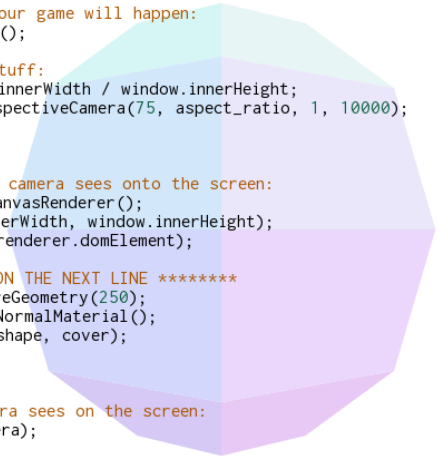
```
<script>
  // This is where stuff in our game will happen:
  var scene = new THREE.Scene();

  // This is what sees the stuff:
  var aspect_ratio = window.innerWidth / window.innerHeight;
  var camera = new THREE.PerspectiveCamera(75, aspect_ratio, 1, 10000);
  camera.position.z = 500;
  scene.add(camera);

  // This will draw what the camera sees onto the screen:
  var renderer = new THREE.CanvasRenderer();
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);

  // ******** START CODING ON THE NEXT LINE ********
  var shape = new THREE.SphereGeometry(250);
  var cover = new THREE.MeshNormalMaterial();
  var ball = new THREE.Mesh(shape, cover);
  scene.add(ball);


  // Now, show what the camera sees on the screen:
  renderer.render(scene, camera);
</script>
```



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

Chunky: SphereGeometry(100, 20, 15)

If you click on the Hide Code button in ICE, you may notice that our sphere isn't _really_ a smooth ball:

You can easily hide or show the code

If you click the white Hide Code button in the upper right corner of the ICE Editor window, then you will see just the game area and the objects in the game. This is how you will play games in later chapters. To get your code back, click the white Show Code button within the ICE Editor.

Computers cannot 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 will get the right number of *chunks* so that it is close enough.

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

```
1 var shape = new THREE.SphereGeometry(100, 20, 15);
  var cover = new THREE.MeshNormalMaterial();
```

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

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

This should make a sphere that is much smoother:

Don't Change the Chunkiness Unless You Have To

The number of chunks that 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. As we will see in later chapters, it is usually easier for a computer to make things look smooth by choosing a different wrapper for the shape.

Play around with the numbers a bit more. You are already learning quite a bit here and playing with the numbers is a great way to keep learning!

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

```
var shape = new THREE.SphereGeometry(100);
var cover = new THREE.MeshNormalMaterial();
var ball = new THREE.Mesh(shape, cover);
scene.add(ball);
ball.position.set(-250,250,-250);
```
1

❶ The three numbers move the ball to the left, up and back. Don't worry too much about what the numbers do right now—we will talk about position when we start building game characters in Chapter 3, *Project: Making a Player*, on page ?.

Making Boxes with Cube

The next shape that we are going to make is a cube, which is another name for a box. There are three different ways to change a cube's shape: the width, the height and the depth.

Size: CubeGeometry(300, 100, 20)

To create a box, we are going to write more JavaScript below everything that we used to create our ball. Type the following:

```
var shape = new THREE.CubeGeometry(100, 100, 100);
var cover = new THREE.MeshNormalMaterial();
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:

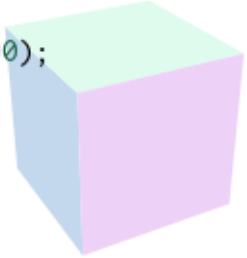
```
var shape = new THREE.CubeGeometry(100, 100, 100);
var cover = new THREE.MeshNormalMaterial();
var box = new THREE.Mesh(shape, cover);
scene.add(box);
box.rotation.set(0.5, 0.5, 0);
```

1

❶ These three numbers turn the box down, counterclockwise, and left-right. Play with the numbers if you like. To make a full turn in any direction, you would need a number around 7.3 (we'll talk about that number later).

This should get the cube rotated so that 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);
```

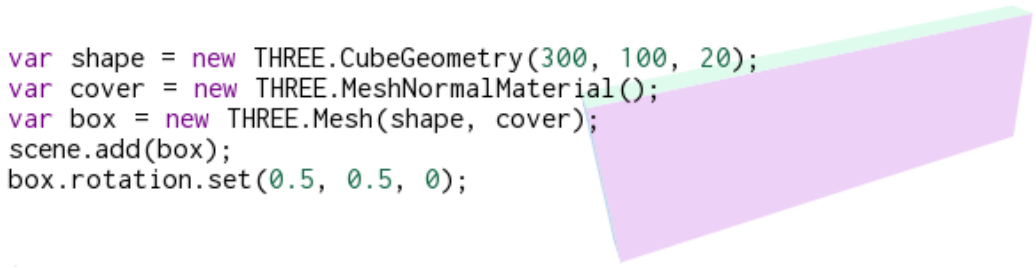


Each side of a cube does not have to be the same size. Our box so far is 100 wide (from left to right), 100 tall (up and down) and 100 deep (front to back). Let's change it so that it is 300 wide, 100 tall and only 20 deep:

```
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);
```

This should show something like:



Play around with the numbers to get a good feel for what they can do.

Believe it or not, you already know a ton about JavaScript and 3D programming. There is still lots to learn, of course, but we can already make balls and boxes. We can already move them and turn them. And we only had to write 10 lines of JavaScript to do it all—nice!

Like the ball, let's move our box out of the way so that we can play with more shapes:

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

Using Cylinders for all Kinds of Shapes

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

Size: CylinderGeometry(20, 20, 100)

Below the box code, type in the following to create a cylinder:

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

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

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

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

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

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?

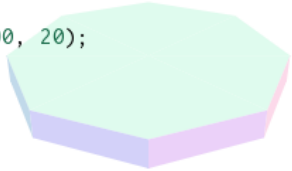
Try This Yourself

Play with those numbers and see what you can create!

What did you find?

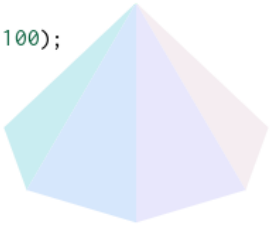
A flat cylinder is a disc:

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



And a cylinder with 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();
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.

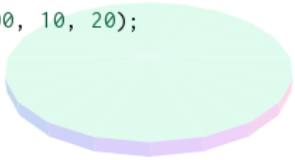
Pyramids: CylinderGeometry(1, 100, 100, 4)

Did you notice that cylinders also look chunky? 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:

```
var shape = new THREE.CylinderGeometry(100, 100, 10, 20);
var cover = new THREE.MeshNormalMaterial();
var tube = new THREE.Mesh(shape, cover);
scene.add(tube);
tube.rotation.set(0.5, 0, 0);
```

Then you should see something like:

```
var shape = new THREE.CylinderGeometry(100, 100, 10, 20);
var cover = new THREE.MeshNormalMaterial();
var tube = new THREE.Mesh(shape, cover);
scene.add(tube);
tube.rotation.set(0.5, 0, 0);
```



Just as with spheres, you should only use lots of chunks like that 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. See if you can figure it out.

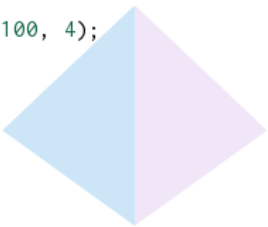
Try This Yourself

Play with different numbers and see what you can create!

Were you able to figure it out? Don't worry if you didn't. It's actually pretty sneaky the way that we will do it.

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 will get:

```
var shape = new THREE.CylinderGeometry(1, 100, 100, 4);
var cover = new THREE.MeshNormalMaterial();
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 are doing great so far. Let's move on to the last two shapes that we are going to learn about in this chapter.

When you are done with this chapter, you will:

- *Learn a super-powerful tool (functions) for programmers*
- *Know two reasons to use functions*
- *Understand how JavaScript functions work*

CHAPTER 5

Functions: Use and Use Again

We have come across functions more than once. Most recently we saw them in Chapter 3, *Project: Making a Player*, on page ?, where we used them to make a forest. If you were paying close attention, you may have noticed that we also used a function to build the keyboard event listener in the same chapter.

Although we have used functions already, we have not talked too much about them. You may already have a sense that they are pretty powerful, so let's take a closer look now.

We are not going to talk about every aspect of functions—they can get quite complicated. We will talk just enough to be able to understand the functions that we use throughout the book.

5.1 Getting Started

Create a new project in the ICE Code Editor. Use the Empty Project template and call it Functions.

After the opening <script> tag, enter the following JavaScript.

```
var log = document.createElement('div');
log.style.height = '75px';
log.style.width = '450px';
log.style.overflow = 'auto';
log.style.border = '1px solid #666';
log.style.backgroundColor = '#ccc';
log.style.padding = '8px';
log.style.position = 'absolute';
log.style.bottom = '10px';
log.style.right = '20px';
document.body.appendChild(log);

var message = document.createElement('div');
```

```
message.textContent = 'Hello, JavaScript functions!';
log.appendChild(message);

message = document.createElement('div');
message.textContent = 'My name is Chris.';
log.appendChild(message);

message = document.createElement('div');
message.textContent = 'I like popcorn.';
log.appendChild(message);
```

The first chunk of that code creates a place within the browser to log messages. The last three blocks of code write three different messages to that log. If you have everything typed in correctly, you should see the three messages printed at the bottom-right of the page.

```
19 message = document.createElement('div');
20 message.textContent = 'My name is Chris.';
21 log.appendChild(message);
22
23 message = document.createElement('div');
24 message.textContent = 'I like popcorn.';
25 log.appendChild(message);
26 </script>
```

Hello, JavaScript functions!
My name is Chris.
I like popcorn.

Back in Chapter 3, *Project: Making a Player*, on page ?, we used a function to avoid having to repeat the same process for creating a tree four times. So you can probably guess the first thing that we are going to change here. Let's change the way that we log those three messages.

Start by deleting everything starting with the first `var message` line all the way through the last `log.appendChild` line. Where that code was, add the following.

```
logMessage('Hello, JavaScript functions!', log);
logMessage('My name is Chris.', log);
logMessage('I like popcorn.', log);

function logMessage(message, log) {
  var holder = document.createElement('div');
  holder.textContent = message;
  log.appendChild(holder);
}
```

When we write that code, a surprising thing happens—it gets easier to read. Even non-programmers could read those first three lines and figure out that they send a message to the log. Believe it or not, this is a *huge* win for programmers like us.

Readable code is easier to change later

One of the skills that separates great programmers from good programmers is the ability to change working code. And great programmers know that it is easier to make changes when the code is easy to read.

If we decide later that we want to add the time before each message, it is much easier to figure out where to make that change now. Obviously we need to change something inside the function. Before, it would have taken us some time to figure out that those three code blocks were writing log messages and how to change them.

This also brings up a very important rule.

Keep your code DRY—Don't Repeat Yourself

This book was published by the same people behind a very famous book called *The Pragmatic Programmer*. If you keep programming, you will read that book one day. In that book, there is a famous tip that good programmers keep their code DRY—that they follow the rule known as Don't Repeat Yourself, or DRY for short.

When we first wrote our code, we repeated three things:

1. Creating a holder for the message
2. Adding a text message to the holder
3. Adding the message holder to the log

It was easy to see that we were repeating ourselves since the code in each of the three chunks was identical except for the message. This is another opportunity for us to be lazy. If we add more than three messages, we only have to type one more line, not three.

And of course, if we have to change something about the log message, we only have to change one function, not three different blocks of code.

We are not quite done using functions here. If you look at all of the code, you will notice that it takes a long time to get to the important stuff.


```
1  <body></body>
2  <script>
3  var log = document.createElement('div');
4  log.style.height = '75px';
5  log.style.width = '450px';
6  log.style.overflow = 'auto';
7  log.style.border = '1px solid #666';
8  log.style.backgroundColor = '#ccc';
9  log.style.padding = '8px';
10 log.style.position = 'absolute';
11 log.style.bottom = '10px';
12 log.style.right = '20px';
13 document.body.appendChild(log);
14
15 logMessage('Hello, JavaScript functions!', log);
16 logMessage('My name is Chris.', log);
17 logMessage('I like popcorn.', log);
18
19 function logMessage(message, log) {
20   var holder = document.createElement('div');
21   holder.textContent = message;
22   log.appendChild(holder);
23 }
24 </script>
```

The important work—writing the messages—does not start until line 15. Before we write messages to the log we need a log, but all of that other stuff is just noise.

To fix that, let's move the noise into a function below the logMessage() lines.

```
function makeLog() {
  var holder = document.createElement('div');
  holder.style.height = '75px';
  holder.style.width = '450px';
  holder.style.overflow = 'auto';
  holder.style.border = '1px solid #666';
  holder.style.backgroundColor = '#ccc';
  holder.style.padding = '8px';
  holder.style.position = 'absolute';
  holder.style.bottom = '10px';
  holder.style.right = '20px';
  document.body.appendChild(holder);

  return holder;
}
```

Note that we have changed log to holder. Also, don't forget the last line that returns holder.

With that, we can create our log with this function. Our first four lines become the following:

```
var log = makeLog();
logMessage('Hello, JavaScript functions!', log);
logMessage('My name is Chris.', log);
logMessage('I like popcorn.', log);
```

That is some very easy-to-read code!

It turns out to be more difficult to write code like that than you would think. Really good programmers know that you don't use functions until you have a good reason for them. In other words, good programmers do exactly what we have done here: write working code first, then look for ways to make it better.

Always Start with Ugly Code

You are a very smart person. You have to be to have made it this far. So you must be thinking, "Oh, I can just write readable code first."

Believe me when I say that you can't. Programmers know this so well that we have a special name for trying it: *premature generalization*. That's just a fancy way to say it is a mistake to guess how functions are going to be used before writing *ugly* code first. Programmers have fancy names for mistakes that we make a lot.

5.2 Understanding Simple Functions

So far we have looked at reasons why we want to use functions. Let's see how functions actually work.

Remove the three logMessage() lines from the code. Write the following after the var log = makeLog line.

```
logMessage(hello("President Obama"), log);
logMessage(hello("Mom"), log);
logMessage(hello("Your Name"), log);

function hello(name) {
  return 'Hello, ' + name + '! You look very pretty today :)';
}
```

The result of this hello() function would be to return the phrase: "Hello, President Obama! You look very pretty today :)". Logging these phrases should look something like:

```
Hello, President Obama! You look very pretty today
:)
Hello, Mom! You look very pretty today :)
Hello, Purple Fruit Monster! You look very pretty
```

There is a lot going on in the hello function to make that work, so let's break the function down into smaller pieces.

```
1 function hello(name) {
2   return 'Hello, ' + name + '! You look very pretty today :)';
}
```

The piece of a function are:

❶ The word function, which tells JavaScript that we are making a... function.

The name of the function, hello in this case.

Function *arguments*. In this case, we are accepting one argument (name) that we will use inside the function body. When we call the function with an argument—hello(Fred)—then we are telling the function that any time it uses the name argument, it is the same as using Fred.

The body of the function starts with an open curly brace, {, and ends with a closing curly brace }. You have probably never used curly braces when writing English. You'll use them a lot when writing JavaScript.

❷ The word return tells JavaScript what we want the result of the function to be. It can be anything: numbers, letters, words, dates, even more interesting things.

JavaScript lines, even those inside functions, should end with a semicolon.

Letters, words, and sentences are Strings

Things inside quotes 'Hello', are called *strings*. Even in other programming languages, letters, words and sentences are usually called strings.

Always be sure to close your quotes. If you forget, you will get very weird errors that are hard to fix.

Next, let's try to break it intentionally so that we get an idea of what to do when things go wrong.

5.3 When Things Go Wrong

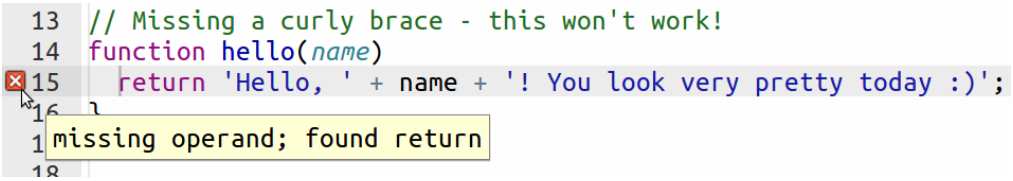
Let's put our hacker hats on and try to break some functions. While it's easy to do something wrong with JavaScript functions, it is not always easy to figure out what you did wrong. The most common mistakes that programmers make generate weird errors. Let's take a look so that you might be better prepared.

Unexpected Errors

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

```
// Missing a curly brace - this won't work!
function hello(name)
  return 'Hello, ' + name + '! You look very pretty today :)';
}
```

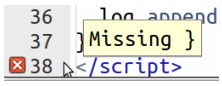
This is a compile-time error in JavaScript—one of the errors that JavaScript can detect when it trying to read, compile and run—that we met in [xxx](#section.console_debugging). Since it is a compile-time error, the ICE Code editor will tell us about the problem.



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

```
// Missing a curly brace - this won't work!
function hello(name) {
  return 'Hello, ' + name + '! You look very pretty today :)';
```

There are no errors in our hello function, but there is an error at the very bottom of our code.



This can be a tough error to fix. Often programmers will type many lines and possibly several functions before they realize that they have done something wrong. Then it just takes time to figure out where you meant to add a curly brace.

Challenge

Try the following broken code on your own. Where do the errors show up? *Hint:* as in Section 2.4, *Debugging in the Console*, on page ?, some of these may be run-time errors.

Don't type parentheses around the argument:

```
// Missing parentheses around the arguments - this won't work!
function hello name {
  return 'Hello, ' + name + '! You look very pretty today :)';
}
```

Don't type an argument:

```
function hello() {
  return 'Hello, ' + name + '! You look very pretty today :)';
}
```

Use a different variable name in the function body:

```
function hello(name) {
  return 'Hello, ' + person + '! You look very pretty today :)';
}
```

Call the function with the wrong name:

```
logMessage(helo("President Obama"), log);

function hello(name) {
  return 'Hello, ' + name + '! You look very pretty today :)';
}
```

Wow! There sure are a lot of ways to break functions. And believe me when I tell you that you will break functions like these and many other ways as you get to be a great programmer.

> **Tip 1** Great programmers break things all the time. Because they break things so much, they are really good at fixing things. This is what makes great programmers great.

Don't ever be upset at yourself if you break code. Broken code is a chance to learn. And don't forget to use the JavaScript Console like we learned in *Playing with the Console and Finding What's Broken* to help troubleshoot!

5.4 Stupid Function Tricks (or are they Awesome Tricks?)

Functions are so special in JavaScript that you can do all sorts of crazy things to them.

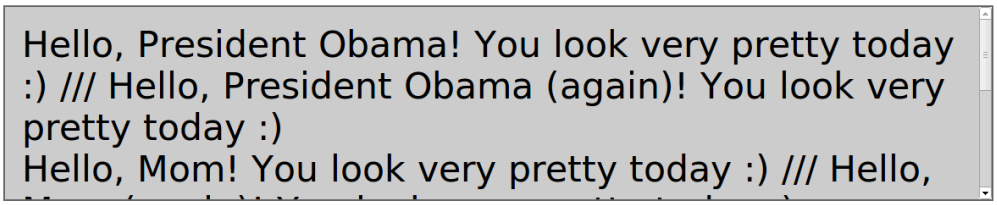
Recursion

Change the hello like this:

```
function hello(name) {
  var ret = 'Hello, ' + name + '! ' + 'You look very pretty today :)';
  if (!name.match(/again/)) {
    ret = ret + ' /// ' + hello(name + ' (again)');
  }
  return ret;
}
```

❶ Look closely here. Inside the body of the function hello, we are calling the function hello!

This will log the hello messages twice.



A function that calls itself like this is actually not crazy. It is so common that it has a special name: a recursive function.

Be careful with recursive functions! If there is nothing that stops the recursive function from calling itself over and over, you will lock your browser and have to go into edit-only mode to fix it (see Appendix 1, *When ICE is Broke*, on page ?).

In this case, we stop the recursion by only calling the hello function again if the name variable does not match the again. If name does not match again, then we call hello() with name + '(again)' so that the next call will include again.

Recursion can be a tough concept, but you have already seen it in the name of your code editor:

- What does the I in ICE Code Editor stand for?
- It stands for ICE Code Editor.
- What does the I in ICE Code Editor stand for?
- It stands for ICE Code Editor.
- What does the I in ICE Code Editor stand for?
-

People will keep asking that question until they get sick of asking. Computers don't get sick of asking, so you have to tell them when to get sick of it.

5.5 What's Next

Functions are a very powerful tool for JavaScript programmers. As you will see shortly, we are going to make a lot of use of them in the upcoming chapters. Let's get started in the next chapter as we teach our player avatar how to move its arms!

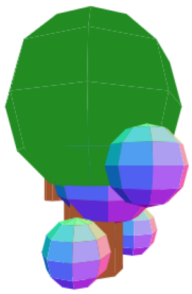
When you are done with this chapter, you will…

- *Be able to stop game elements from moving through each other*
- *Understand collisions, which are important in gaming*
- *Have game boundaries for our player*

CHAPTER 10

Project: Collisions

After Chapter 8, *Project: Turning Our Player*, on page ?, we have ourselves a pretty slick game player. It moves, it walks, it even turns. But you may have noticed something odd about our player. It can walk through trees.

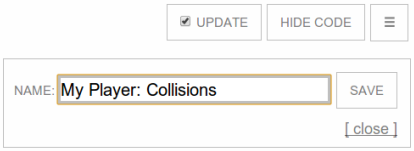


This will be another chapter where we use math, especially geometry concepts. And again, we should find them pretty easy.

In this chapter, we are going to use tools that are built into our Three.js 3D JavaScript library to prevent the player-in-a-tree effect. As we will see in other chapters, there are other ways to do the same thing. There are times when using an approach like what we cover in this chapter makes sense.

10.1 Getting Started

Let's start by making a new copy of our player. From the menu in the ICE Code Editor, select Make a Copy and enter My Player: Collisions as the new project name.



10.2 Rays and Intersections

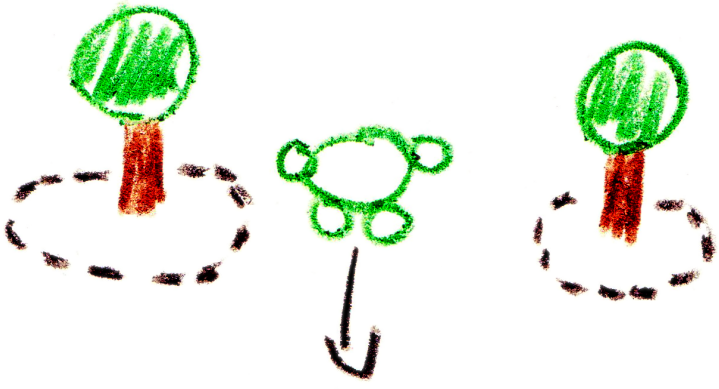
The way that we prevent our player from walking through trees is actually quite simple. Imagine an arrow pointing down from our player.

In geometry, we call an arrow point 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 player is and the direction is down. It's silly sometimes how we give names to such simple ideas. But make no mistake, it is very 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 is the crazy-simple way that we prevent our player from running into a tree: we don't! Instead, we prevent the player's ray from pointing through the tree's circle.

If, at any time, we find that the next movement would place the player's ray so that it pointed through the circle, we stop the player from moving. That's all there is to it!

Star Trek II: The Wrath of Khan

It may seem strange, but watching certain science fiction movies will make your life easier as a programmer. Sometimes, programmers say silly 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 used to thinking in three dimensions and this was used against him. In this case, we are doing *exactly the opposite*. In our three-dimensional game, we are thinking about collisions only in two dimensions (X and Z), completely ignoring the up-and-down Y dimension.

This is yet another example of cheating whenever possible. *Real* 3D collisions are very hard and require new JavaScript libraries. But we can cheat and get the same effect in many cases using easier tricks.

At this point, a picture should be beginning to form in your mind of what to do next. We are going to need a list of these tree circle boundaries that our player will not be allowed to enter. We will need to build those circle boundaries when we build the trees. We need to detect when the player is about to

enter a circle boundary. Last, we need to stop the player from entering these forbidden areas.

First, let's establish the list that will hold all forbidden boundaries. Just below the "START CODING ON THE NEXT LINE" line, add the following.

collisions/collisions.html
```
var not_allowed = [];
```

Recall from Section 7.4, *Listing Things*, on page ? that square brackets are JavaScript's way of making lists. Here, our empty square brackets create an empty list. The not_allowed variable is an empty list of spaces in which the player is not allowed.

Next, find where makeTree() is defined. When we make our tree, we are going to make the boundaries as well. Add the following code after the line that adds the treetop to the trunk, and before the line that sets the trunk position.

collisions/collisions.html
```
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);

not_allowed.push(boundary);
```

As an experienced 3D programmer, there is nothing too fancy there. We create our usual 3D Mesh—this time with a simple circle geometry. We rotate it so that it lies flat and position it below the tree. And, of course, we finish by adding it to the tree.

But unlike normal meshes, we are not quite done with our boundary mesh. We push it onto the list of not-allowed spaces. Now every time that we make a tree with the makeTreeAt() function we are building up this list. Let's do something with that list.

At the very bottom of our code, just above the </script> tag, add the following code to detect collisions.

collisions/collisions.html
```
function detectCollisions() {
  var vector = new THREE.Vector3(0, -1, 0);
  var ray = new THREE.Ray(marker.position, vector);
  var intersects = ray.intersectObjects(not_allowed);
  if (intersects.length > 0) return true;
  return false;
```

```
}
```

This function returns a boolean—a yes or no answer—depending on whether or not the player is colliding with a boundary. This is where we make our ray to see if it points through anything. As described earlier, a ray is the combination of a direction, or vector (down in our case) and a point (the player's marker.position). We then ask that ray if it goes through (intersects) any of the not_allowed objects. If the ray does intersect a not-allowed object, then the intersects variable will have a length that is greater than zero. In that case, we have detected a collision and we return true. Otherwise, there is no collision and we return false.

Collisions are a very hard problem to solve in many situations, so you are doing great just following along with this. But we are not quite done. We can detect when a player is colliding with a boundary, but we have not actually stopped the player yet. Let's do this in the keydown listener.

In the keydown listener, if an arrow key is pressed, we change the player's position.

collisions/collisions.html

```
case 37: // left
  marker.position.x = marker.position.x-5;
  is_moving_left = true;
  break;
case 37: // left
  is_moving_left = false;
  break;
```

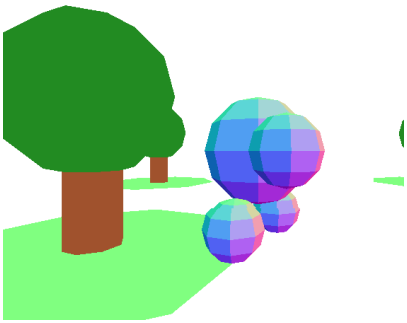
It is possible that such a change means that the player is now in the boundary. If so, we have to undo the move right away. Add the following code at the bottom of the keydown event listener (just before the event.preventDefault() line).

collisions/collisions.html

```
if (detectCollisions()) {
  if (is_moving_left) marker.position.x = marker.position.x+5;
  if (is_moving_right) marker.position.x = marker.position.x-5;
  if (is_moving_forward) marker.position.z = marker.position.z+5;
  if (is_moving_back) marker.position.z = marker.position.z-5;
}
```

Read through these lines to make sure that you understand them. That bit of code says that, if we detect a collision, then check the direction in which we are moving. If we are moving left, then reverse the movement that the player just did—go back the opposite direction the same amount.

With that, our player 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.

10.3 The Code So Far

collisions/collisions.html

```
<body></body>
<script src="http://gamingJS.com/Three.js"></script>
<script src="http://gamingJS.com/Tween.js"></script>
<script src="http://gamingJS.com/ChromeFixes.js"></script>
<script>
  // This is where stuff in our game will happen:
  var scene = new THREE.Scene();

  // This is what sees the stuff:
  var aspect_ratio = window.innerWidth / window.innerHeight;
  var camera = new THREE.PerspectiveCamera(75, aspect_ratio, 1, 10000);
  camera.position.z = 500;
  //scene.add(camera);

  // This will draw 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 not_allowed = [];

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

  var cover = new THREE.MeshNormalMaterial();
  var body = new THREE.SphereGeometry(100);
  var player = new THREE.Mesh(body, cover);
  marker.add(player);

  var hand = new THREE.SphereGeometry(50);

  var right_hand = new THREE.Mesh(hand, cover);
```

```
right_hand.position.set(-150, 0, 0);
player.add(right_hand);

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

var foot = new THREE.SphereGeometry(50);

var right_foot = new THREE.Mesh(foot, cover);
right_foot.position.set(-75, -125, 0);
player.add(right_foot);

var left_foot = new THREE.Mesh(foot, cover);
left_foot.position.set(75, -125, 0);
player.add(left_foot);

marker.add(camera);

// Trees
makeTreeAt( 500,  0);
makeTreeAt(-500,  0);
makeTreeAt( 750, -1000);
makeTreeAt(-750, -1000);

function makeTreeAt(x, z) {
  var trunk = new THREE.Mesh(
    new THREE.CylinderGeometry(50, 50, 200),
    new THREE.MeshBasicMaterial({color: 0xA0522D})
  );

  var top = new THREE.Mesh(
    new THREE.SphereGeometry(150),
    new THREE.MeshBasicMaterial({color: 0x228B22})
  );
  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);

  not_allowed.push(boundary);

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

```
}

// Now, animate what the camera sees on the screen:
var clock = new THREE.Clock(true);
function animate() {
  requestAnimationFrame(animate);
  TWEEN.update();
  walk();
  turn();
  acrobatics();
  renderer.render(scene, camera);
}
animate();

function walk() {
  if (!isWalking()) return;
  var position = Math.sin(clock.getElapsedTime()*5) * 50;
  right_hand.position.z = position;
  left_hand.position.z = -position;
  right_foot.position.z = -position;
  left_foot.position.z = position;
}

function turn() {
  var direction = 0;
  if (is_moving_forward) direction = 0;
  if (is_moving_back) direction = Math.PI;
  if (is_moving_right) direction = Math.PI / 2;
  if (is_moving_left) direction =  -Math.PI / 2;

  spinAvatar(direction);
}

function spinAvatar(direction) {
  new TWEEN
    .Tween({y: player.rotation.y})
    .to({y: direction}, 100)
    .onUpdate(function () {
      player.rotation.y = this.y;
    })
    .start();
}

var is_cartwheeling = false;
var is_flipping = false;
function acrobatics() {
  if (is_cartwheeling) {
    player.rotation.z = player.rotation.z + 0.05;
  }
  if (is_flipping) {
```

```
    player.rotation.x = player.rotation.x + 0.05;
  }
}

var is_moving_left, is_moving_right, is_moving_forward, is_moving_back;
function isWalking() {
  if (is_moving_right) return true;
  if (is_moving_left) return true;
  if (is_moving_forward) return true;
  if (is_moving_back) return true;
  return false;
}

document.addEventListener('keydown', function(event) {
  switch (event.keyCode) {
    case 37: // left
      marker.position.x = marker.position.x-5;
      is_moving_left = true;
      break;
    case 38: // up
      marker.position.z = marker.position.z-5;
      is_moving_forward = true;
      break;
    case 39: // right
      marker.position.x = marker.position.x+5;
      is_moving_right = true;
      break;
    case 40: // down
      marker.position.z = marker.position.z+5;
      is_moving_back = true;
      break;
    case 67: // C
      is_cartwheeling = !is_cartwheeling;
      break;
    case 70: // F
      is_flipping = !is_flipping;
      break;
  }
  if (detectCollisions()) {
    if (is_moving_left) marker.position.x = marker.position.x+5;
    if (is_moving_right) marker.position.x = marker.position.x-5;
    if (is_moving_forward) marker.position.z = marker.position.z+5;
    if (is_moving_back) marker.position.z = marker.position.z-5;
  }

  event.preventDefault();
});

document.addEventListener('keyup', function(event) {
  switch (event.keyCode) {
```

```
      case 37: // left
        is_moving_left = false;
        break;
      case 38: // up
        is_moving_forward = false;
        break;
      case 39: // right
        is_moving_right = false;
        break;
      case 40: // down
        is_moving_back = false;
        break;
    }
    event.preventDefault();
  });

  function detectCollisions() {
    var vector = new THREE.Vector3(0, -1, 0);
    var ray = new THREE.Ray(marker.position, vector);
    var intersects = ray.intersectObjects(not_allowed);
    if (intersects.length > 0) return true;
    return false;
  }
</script>
```

10.4 What's Next

Collision detection in games is a really tricky problem to solve, so congratulations on getting this far. It gets even harder once you have to worry about moving up and down in addition to left/right/back/forward. But the concept is the same.

Usually we rely on code libraries written by other people to help us with those cases. In some of the games coming shortly, we will use just such a code library.

But first, we are going to put the finishing touch on our player game. In the next chapter, we are going to add a heads up display (HUD) and use that to let our player go on a little scavenger hunt.