Making Games

With JavaScript

Christopher Pitt

Apress®

Making Games

With JavaScript

Christopher Pitt

Apress®

Making Games: With JavaScript

Christopher Pitt
Cape Town, Western Cape, South Africa

ISBN-13 (pbk): 978-1-4842-2492-2 ISBN-13 (electronic): 978-1-4842-2493-9
DOI 10.1007/978-1-4842-2493-9

Library of Congress Control Number: 2016961803

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Mark Powers
Copy Editor: Rebecca Rider
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover image: Designed by Freepik

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

For Sam, Nicole, Eva, and Simon.

Contents at a Glance

Contents

About the Author

Christopher Pitt is a developer and writer working at SilverStripe. He usually works on application architecture, though sometimes you'll find him building compilers or robots.

I've written this short book with Javascript developers in mind. For it to be valuable to you, you'll need to understand how to use JavaScript. Where possible, I've explained newer syntax and difficult concepts. But this is not the best book to learn Javascript from.

If you're new to JavaScript, I recommend reading [Learning JavaScript] (http://a.co/1mwOUfl) first. It will help you understand some of the tricky parts, and put you in the right mindset to learn what this book teaches.

If you have questions, don't hesitate to [ask me questions] (https://twitter.com/assertchris).

CHAPTER 1

Introduction

I'm a gamer. I've been a gamer since before I was a programmer. And yet I've never tried to build a game…until now. A few weeks ago, a coworker sketched a simple, beautiful platform game. The moment I saw it; I knew I wanted to build it.

Knowing where to begin is always hard. Many books and tutorials about making games are often written in Java, C++, or some other language you haven't learned yet. So the complexity of making games increases when you also need to learn new languages.

Instead, I'm going to use modern JavaScript to make games for web browsers. They're effortless to use and they work everywhere. If you've ever wanted to make a game, join me. We'll start with nothing and build fun games in no time at all.

If you have questions, feel free to ask me on Twitter: https://twitter.com/assertchris.

Electronic supplementary material The online version of this chapter (doi: 10.1007/978-1-4842-2493-9_1) contains supplementary material, which is available to authorized users.

C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_1

CHAPTER 2

The Game Loop

Game loops are an essential part of every game. In this chapter, we're going to set the stage for our game by creating a solid workflow and environment. We'll see a few helpful libraries and render our first game character. This is gonna be fun!

Setting the stage for our game

We're going to build games using the latest JavaScript standards and language features. This is usually where I'd show you how to create a JavaScript-built chain, but we're going to do something different....

For this book, we're going to do all of our coding in a hosted service called CodePen. It looks like the code in Figure 2-1.

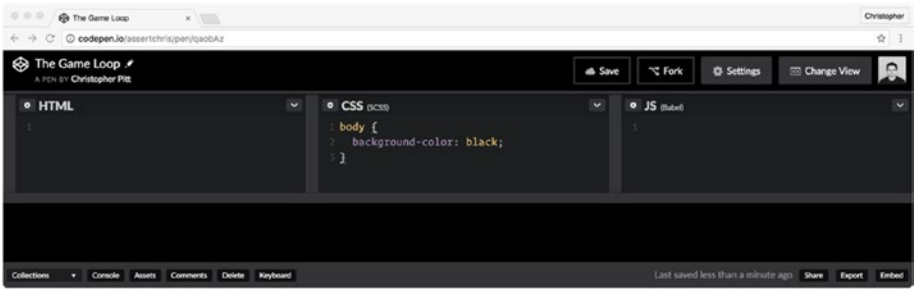

Figure 2-1. *CodePen*

There are a few benefits to this approach:

1. We don't need to cover JavaScript build chains. They are subjective, brittle, and distracting.
2. We can use SCSS (enhanced stylesheets) for free.
3. You don't have to set anything up to start interacting with the source code.

C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_2

We're going to use a JavaScript library, called PixiJS (www.pixijs.com). It's a rendering library that will smooth over the browser inconsistencies. It's not a game engine or a physics engine, so we're still going to learn and code those aspects ourselves. PixiJS will just allow us to get to those things sooner.

We're using PixiJS v4. It's entirely possible that newer versions are available by the time you read this. The concepts should be universal, though the syntax might have changed.

We can add PixiJS to our CodePens by clicking on Settings ➤ JavaScript, and adding the following URL to the PixiJS CDN script: https://cdnjs.cloudflare.com/ajax/libs/pixi.js/4.0.0/pixi.min.js. This process is shown in Figure 2-2.

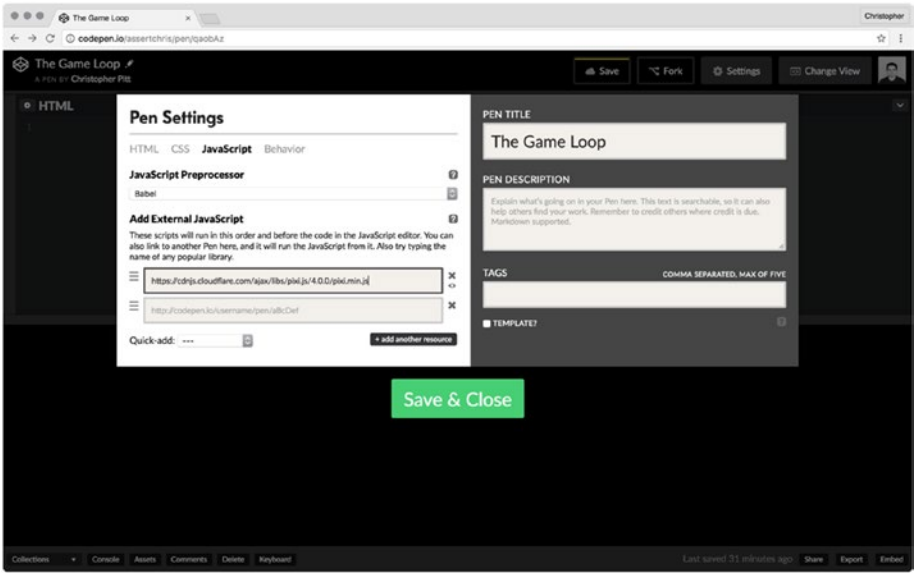

Figure 2-2. *Adding PixiJS to CodePen*

Creating sprites

Sprites are a common name for visual objects in a game. We can move them around (though often they are responsible for moving themselves around) on the screen. We can interact with them.

Mario (https://en.wikipedia.org/wiki/Mario) is a sprite, the platforms he walks on are sprites, and the clouds in the background are sprites. Think of sprites as slices of a design file, which we paint over abstract data structures. Those abstract data structures have a position, and sometimes a velocity. Those abstract data structures are what we apply game rules to. They are our power-ups and enemies.

So how do we make them? Let's start by creating a small PNG image. We're using PNG because it allows us to make parts of the sprite texture transparent. You can also use JPEG images for your sprites if you want to.

Then we need to create a renderer, a stage, and a sprite:

```
const renderer = new PIXI.autoDetectRenderer(
  window.innerWidth,
  window.innerHeight,
  {
    "antialias": true,
    "autoResize": true,
    "transparent": true,
    "resolution": 2,
  },
)

document.body.appendChild(renderer.view)

const sprite = new PIXI.Sprite.fromImage(
  "path/to/sprites/player-idle.png",
)

sprite.x = window.innerWidth / 2
sprite.y = window.innerHeight / 2

const stage = new PIXI.Container()
stage.addChild(sprite)

const animate = function() {
  requestAnimationFrame(animate)
  renderer.render(stage)
}

animate()
```

This is from http://codepen.io/assertchris/pen/qaobAz.

Ok, there's a lot going on here! Let's look at this code in steps:

1. We create a renderer. *Renderers* are what convert PixiJS abstractions (sprites, textures, etc.) into canvas or WebGL graphics. We don't have to interact with them, but we always have to have a renderer in order for our PixiJS stuff to show up.

   We tell the renderer to take up the full width and height of the browser window. We also tell it to anti-alias our graphics and bump them up to retina resolution. We tell it to have a transparent background, and to resize all graphics to fit on the screen.

2. Then we create a new instance of `PIXI.Sprite`, using the PNG image we created earlier. By default, it has a position of `x = 0` and `y = 0`. We can position it in the center of the screen instead.

   We have to create a root sprite container, often called a *stage*, and append the sprite to the stage. It's less complicated than it sounds. Think of HTML documents. They have a root `html` element, which we add all other elements to. This is the same kind of thing.

3. Finally, we create an animate function and make sure that the renderer renders our stage and sprite.

   We're using the `requestAnimationFrame` function as a way rendering without blocking the JavaScript thread. There's a long discussion we could have about that. For now, it's only important to know that `requestAnimationFrame` happens many times a second.

   The goal is for our game is to render between 30 and 60 frames per second, while doing all sorts of player and world calculations in the background. This is the function we need to use for that all to go smoothly.

The Game Loop

The game loop will control the flow of our game. It's a repetitive process of reading input, calculating changes in state, and rendering output to the screen.

So far we've set up a lot of scaffolding and all we're doing is rendering a static image. Let's move it around a bit! To begin with, we're going to create a `Player` class so we can track the position of the player:

```
class Player {
  constructor(sprite, x, y) {
    this.sprite = sprite
    this.x = x
    this.y = y

    this.sprite.x = this.x
    this.sprite.y = this.y
  }

  animate(state) {
    this.x += 5

    if (this.x > window.innerWidth) {
      this.x = 0
    }
```

```
    this.sprite.x = this.x
    this.sprite.y = this.y
  }
}
```

This is from http://codepen.io/assertchris/pen/qaobAz.

This is what JavaScript classes look like in ES2015 and beyond. The `Player` class has a constructor, which we use to store a reference to the sprite, and the initial `x` and `y` position.

It also has an `animate` method, which we will call many times a second. We can use the method to figure out if the player needs to change position or do something else. In this case, we increment the sprite `x` position a bit. If the player/sprite moves off the right-hand side of the screen, we move it back to the left-hand side of the screen.

We also have to change how we create the sprite:

```
const player = new Player(
  sprite,
  window.innerWidth / 2,
  window.innerHeight / 2,
)

stage.addChild(sprite)

let state = {
  "renderer": renderer,
  "stage": stage,
}

const animate = function() {
  requestAnimationFrame(animate)

  player.animate(state)
  renderer.render(stage)
}
```

This is from http://codepen.io/assertchris/pen/qaobAz.

Now we're beginning to see the loop in our game. As you can see in Figure 2-3, many times a second, we're telling our player to animate. The player takes into account some internal logic and adjusts its sprite. The renderer renders the output, and what we see is a smooth animation.

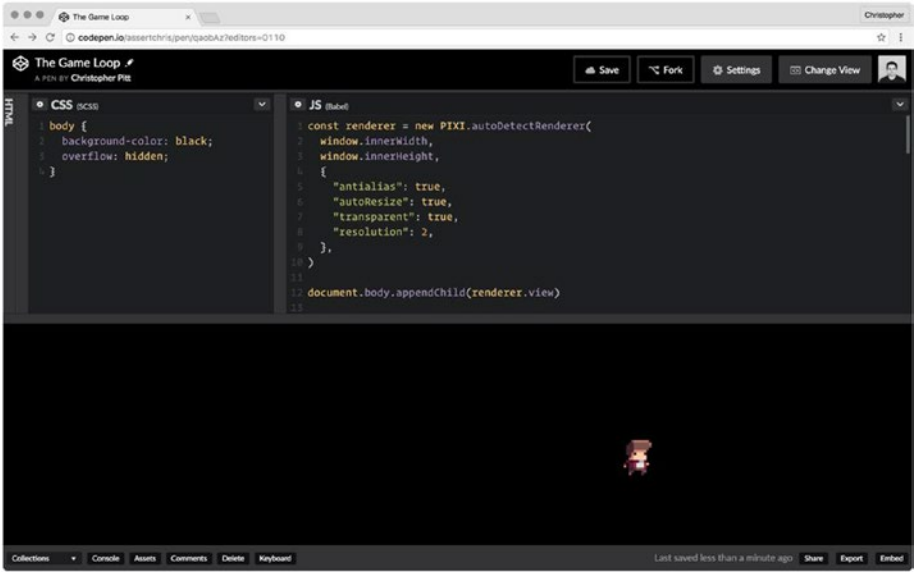

Figure 2-3. *Moving sprites*

The only thing we're lacking here is input, but we'll get to that soon…

Summary

In this chapter, we managed to set up a game environment (using CodePen). We were briefly introduced to PixiJS and saw how to render our player's sprite on the screen.

We also animated the player's sprite and learned how to use the `requestAnimationFrame` function.

Take a moment to look for sprite graphics for your game. Fork the CodePen, and customize it a little. It's important that you get a feel for creating sprites and manipulating them. This is the start of your game!

CHAPTER 3

Player Input

What's the difference between a movie and a game? Player input! In fact, it's such a critical piece of game design that games are often defined by how they take player input.

Racing games rely on constant, subtle input. If you lift your fingers from the keyboard, or lift your foot off the pedal, you lose momentum. If you don't enter the right combination of keys, your flight simulation will crash.

Platform games, like the one we're building, need input from both hands. Often the right-hand side of the keyboard is for movement. Often the left-hand side of the keyboard is for player actions, like shooting or opening doors.

Perhaps you want to design a platform game like Terraria (https://terraria.org), which uses the mouse for player actions and projective aim. Perhaps you want to use WASD to move your player around the screen. Let's get to work!

Detecting input

When I first tried detecting input, I tried to add event listeners to everything. I tried adding event listeners to the player and event listeners to the stage. Aaaand things got out of hand.

The trick is to detect events at the highest level and save details of them to a game state object. Last time we created this kind of state object and passed it to the player's `animate` method. Let's expand on that state:

```
let state = {
  "renderer": renderer,
  "stage": stage,
  "keys": {},
  "clicks": {},
  "mouse": {},
}

window.addEventListener("keydown", function(event) {
  state.keys[event.keyCode] = true
})
```


C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_3

```
window.addEventListener("keyup", function(event) {
  state.keys[event.keyCode] = false
})

window.addEventListener("mousedown", function(event) {
  state.clicks[event.which] = {
    "clientX": event.clientX,
    "clientY": event.clientY,
  }
})

window.addEventListener("mouseup", function(event) {
  state.clicks[event.which] = false
})

window.addEventListener("mousemove", function(event) {
  state.mouse.clientX = event.clientX
  state.mouse.clientY = event.clientY
})
```

This is from http://codepen.io/assertchris/pen/rrdZYB.

We began by adding the renderer and state to the game state object. Now we've added `keys`, `clicks`, and `mouse` properties, which track buttons and movement.

■ **Note** For this next part, I had to adjust the CSS of `index.html` so that the canvas is absolutely positioned. This made it possible to track the mouse movement more accurately. Check out the source code to see the change.

As we add more objects to the game, we'll pass the game state object to each of them. They can use it to work out what they should do when the player interacts with the game. Let's remove the current player movement and add input-driven movement:

```
animate(state) {
  if (state.keys[37]) { // left
    this.x = Math.max(
      0, this.x - 5
    )
  }

  if (state.keys[39]) { // right
    this.x = Math.min(
      window.innerWidth - 64, this.x + 5
    )
  }
```

```
  if (state.clicks[1]) { // left click
    this.x = state.clicks[1].clientX
  }

  this.sprite.x = this.x
  this.sprite.y = this.y
}
```

This is from http://codepen.io/assertchris/pen/rrdZYB.

Little actually changes in the `animate` method. We check for two arrow keys and move if the player pressed one of them. We also check if the player has clicked and immediately move if so.

■ **Tip** Pay no attention to the `Math` functions and magic number `64`. These are just there to prevent the player sprite from going offscreen. We'll remove these things when we start building walls….

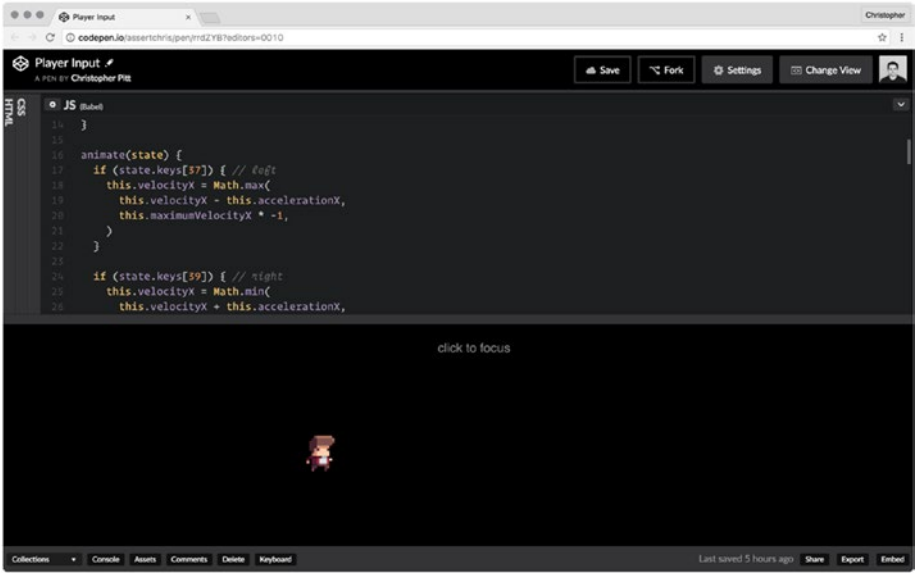

Figure 3-1. *Player input*

Natural player movement

The player can now move a bit. It doesn't feel great, though. The moment we let go of an arrow key, the action just grinds to a halt. The game has only one speed: slow.

What we need is some acceleration to allow the player to speed up in a given direction. We could also use some friction to slow the player down when they are no longer accelerating. Let's add these:

```
class Player {
  constructor(sprite, x, y) {
    this.sprite = sprite
    this.x = x
    this.y = y

    this.velocityX = 0
    this.maximumVelocityX = 8
    this.accelerationX = 2
    this.frictionX = 0.9

    this.sprite.x = this.x
    this.sprite.y = this.y
  }

  animate(state) {
    if (state.keys[37]) { // left
      this.velocityX = Math.max(
        this.velocityX - this.accelerationX,
        this.maximumVelocityX * -1
      )
    }

    if (state.keys[39]) { // right
      this.velocityX = Math.min(
        this.velocityX + this.accelerationX,
        this.maximumVelocityX
      )
    }

    this.velocityX *= this.frictionX

    this.x += this.velocityX

    // if (state.clicks[1]) { // left click
    //   this.x = state.clicks[1].clientX
    // }
```

```
    this.sprite.x = this.x
    this.sprite.y = this.y
  }
}
```

This is from `http://codepen.io/assertchris/pen/rrdZYB`.

Whoa there, tiger! Let's review this in steps:

1. We've created a handful of properties: velocity, maximum velocity, acceleration, and friction. We've set them to reasonable defaults. Feel free to experiment with their values until you appreciate what each one does.

2. We can begin to track the acceleration of our player in either direction. If we're pressing the left key, we start to accelerate the player in that direction. There's a maximum acceleration that the player can have so they don't continue to gain speed.

3. Without some kind of counterforce, the player will continue to move in the same direction. This is like what happens in space where there is no air resistance or gravity to counteract momentum.

   We define this counterforce as friction and multiply the velocity by it. Without acceleration, this means the momentum tends toward zero. This gives the appearance that the player has stopped.

Summary

We've started to make our game interactive. We can now press the arrow keys and see our player's sprite move smoothly around the scene.

We've also made things a little less jarring by adding acceleration and deceleration to the sprite's movement. This logic will inform how we create other mechanics (like gravity).

Take a moment to adjust your acceleration properties so your player's sprite moves just right for your game. You can also try mapping other keys to the left and right actions.

CHAPTER 4

Collision Detection

It's time for us to talk about *collision detection*. It affects obvious parts of our game, like walls we can't walk through. Like floors we can't fall through. It also affects obscure parts of our game like weapon projectiles and checkpoints.

We're not going to get to gravity yet. Nor are we going to look at player health or respawning. Floors, projectiles, and checkpoints are interesting, but they deserve sections of their own. In this chapter, we're going to create impassable objects. We're going to learn ways of knowing whether two objects occupy the same space.

I've spent a bit of time researching this topic. It seems there are many ways of working out whether two things occupy the same space. Some of them are easy to explain and implement. We'll look at those. Other ways are not easy. They're still cool, though.

Creating boxes

The player is only one of many objects that will exist on the screen at one time. Our game is a platform game, so we can expect at least one platform on screen at any given time.

Platforms have some interesting characteristics. Sometimes they allow players to fall down through them. Like when you're standing on a platform and you hold down and press jump (at the same time). Some games take that sequence to mean that you want to fall through the platform.

Similarly, some games allow players to jump through the bottom of platforms. This enables vertical movement without having gaps in overhead platforms.

Sometimes platforms even move!

Platforms are so special that we'll spend a few sections just implementing their different behaviors. But right now we're going to focus on another common object. The generic box.

Think of this box as an ancestor of the platform. It may share some platform functionality, but the main reason it exists is for things to collide with it. Especially things like the player.

The boxes we're going to make might not even look like boxes. When we get to implementing gravity, we'll need a wide, thin box to keep the player from falling out of the world. Well also need tall, thin boxes to stop the player from running off the side of the floor. We'll make walls out of them. We may even make boxes out of the boxes. Big, wooden, "jump on me to get to higher things" boxes.

C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_4

Ok, enough talking.

```
class Box {
  constructor(sprite, x, y, w, h) {
    this.sprite = sprite
    this.x = x
    this.y = y
    this.w = w
    this.h = h

    this.sprite.x = this.x
    this.sprite.y = this.y
  }

  animate(state) {
    this.sprite.x = this.x
    this.sprite.y = this.y
  }
}
```

This is from http://codepen.io/assertchris/pen/qaokJj.

To make this class, I copied and pasted the `Player` class and deleted a bunch of stuff. I did have to add a width and height property to it. We'll get to that in a bit.

Next, we need to add two of these boxes to the stage:

```
const playerSprite = new PIXI.Sprite.fromImage(
  "path/to/sprites/player-idle.png",
)

const player = new Player(
  playerSprite,
  window.innerWidth / 2,
  window.innerHeight / 2,
  44,
  56,
)

const blob1Sprite = new PIXI.Sprite.fromImage(
  "path/to/sprites/blob-idle-1.png",
)

const blob1 = new Player(
  blob1Sprite,
  (window.innerWidth / 2) - 150,
  (window.innerHeight / 2) - 35,
  48,
  48,
)
```

```
const blob2Sprite = new PIXI.Sprite.fromImage(
  "path/to/sprites/blob-idle-2.png",
)

const blob2 = new Player(
  blob2Sprite,
  (window.innerWidth / 2) + 150,
  (window.innerHeight / 2) + 35,
  48,
  48,
)

const stage = new PIXI.Container()
stage.addChild(playerSprite)
stage.addChild(blob1Sprite)
stage.addChild(blob2Sprite)

let state = {
  "renderer": renderer,
  "stage": stage,
  "keys": {},
  "clicks": {},
  "mouse": {},
  "objects": [
    player,
    blob1,
    blob2,
  ],
}
```

This is from http://codepen.io/assertchris/pen/qaokJj.

That's strange! I've created two new `Box` instances and called them `blobs`. That's because we're about to look at…

Detecting circle collisions

I want you to create circles for these first few boxes. The type of collision detection we're going to do first, is with circles. It's okay that the box has a width and height instead of a radius. You won't use this kind of collision detection often, unless your platform game has a lot of circles in it.

Let's see how this kind of detection works:

```
class Player {
  constructor(sprite, x, y, w, h) {
    this.sprite = sprite
    this.x = x
    this.y = y
    this.w = w
    this.h = h

    this.velocityX = 0
    this.maximumVelocityX = 8
    this.accelerationX = 2
    this.frictionX = 0.9

    this.sprite.x = this.x
    this.sprite.y = this.y
  }

  animate(state) {
    if (state.keys[37]) { // left
      this.velocityX = Math.max(
        this.velocityX - this.accelerationX,
        this.maximumVelocityX * -1,
      )
    }

    if (state.keys[39]) { // right
      this.velocityX = Math.min(
        this.velocityX + this.accelerationX,
        this.maximumVelocityX,
      );
    }

    this.velocityX *= this.frictionX

    let move = true

    state.objects.forEach((object) => {
      if (object === this) {
        return
      }

      var deltaX = this.x - object.x
      var deltaY = this.y - object.y

      var distance = Math.sqrt(
        deltaX * deltaX + deltaY * deltaY,
      );
```

```
      if (distance < this.w / 2 + object.w / 2) {
        if (this.velocityX < 0 && object.x <= this.x) {
          move = false
        }

        if (this.velocityX > 0 && object.x >= this.x) {
          move = false
        }
      }
    });

    if (move) {
      this.x += this.velocityX
    }

    this.sprite.x = this.x
    this.sprite.y = this.y
  }
}
```

This is from http://codepen.io/assertchris/pen/qaokJj.

The first thing we need to do is define a width and height. Although we're pretending that our players and boxes are circles, we only need half a width as a radius.

Next we check each object in the state. We can ignore the player object because we don't need to know when something collides with itself. We do need to check everything else, though.

Circles collide when the distance between their origins is less than their combined radii. Their middle points are so close that their lines have to be overlapping.

We do a quick check to see whether the direction the player is moving in is where the box is. If that's the case, then we prevent the player from moving in that direction.

Give it a go. It's pretty fun to see how non-squares block each other. Of course they all have to be perfect circles for this simple algorithm to work.

Detecting rectangle collisions

Detecting collisions of rectangles is almost as easy as circles. Go ahead and swap the blob image for a box image. You can even adjust the sprite names to reflect the squareness of your boxes.

This time around, we'll treat the player like a rectangle. Instead of radii, we need to check if there are gaps between the player rectangle and either box. We call this *axis-aligned bounding box collision detection* (or AABB for short).

If there is no gap, and the player wants to move in the direction of the box, then we prevent that from happening:

```
let move = true

state.objects.forEach((object) => {
  if (object === this) {
    return
  }

  if (this.x < object.x + object.w &&
      this.x + this.w > object.x &&
      this.y < object.y + object.h &&
      this.y + this.h > object.y) {

    if (this.velocityX < 0 && object.x <= this.x) {
      move = false
    }

    if (this.velocityX > 0 && object.x >= this.x) {
      move = false
    }
  }
})

if (move) {
  this.x += this.velocityX
}
```

This is from http://codepen.io/assertchris/pen/qaokJj

These are simple methods for detecting collisions, but there are others. There's one that uses projection-based vector mathematics (www.sevenson.com.au/actionscript/sat) to determine overlap. There's another that checks each line, in a couple of polygons, to see if any lines intersect (http://stackoverflow.com/questions/9043805/test-if-two-lines-intersect-javascript-function). It's crazy.

You can even experiment with groups of circles colliding together. That might be fun. In Figure 4-1, I'm going to run this little character into these little boxes for a bit…

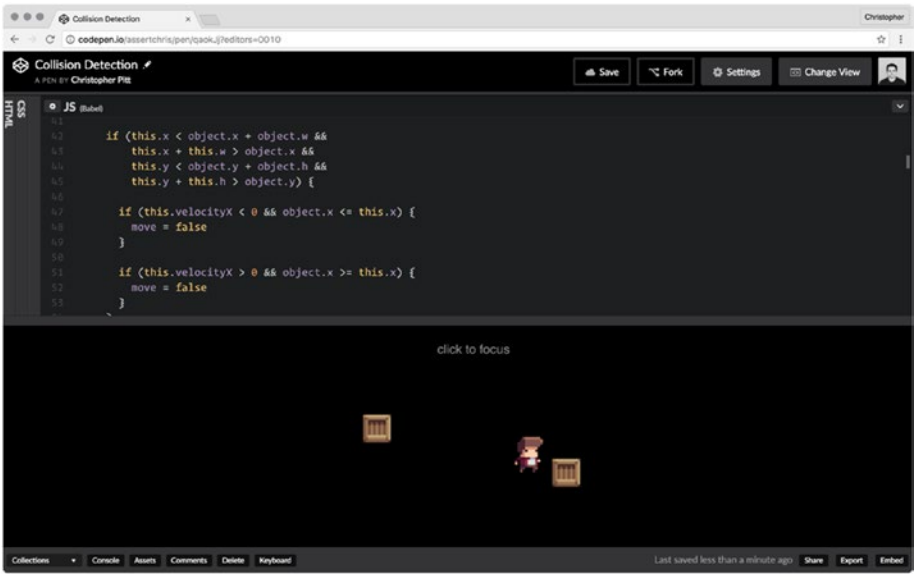

Figure 4-1. *Players and boxes*

Summary

In this chapter, we looked at a couple methods we can use to detect collisions between player spites and various other objects in the game.

Take some time to rearrange the boxes and blobs so you can get a feel for how your first game level might look.

CHAPTER 5

Gravity

In this chapter, we're going to work on our code structure and add gravity to our game. We've already done most of the work for gravity, so it should be relatively straightforward to finish it up.

Cleaning up our existing code

We need to clean up a few things! First, let's swap out `x`, `y`, `w`, and `h` (both in `Box` and `Player`) with `PIXI.Rectangle`. They have these properties, but they also interact with the rest of PIXI in interesting ways.

```
class Player {
  constructor(sprite, rectangle) {
    this.sprite = sprite
    this.rectangle = rectangle

    this.velocityX = 0
    this.maximumVelocityX = 8
    this.accelerationX = 2
    this.frictionX = 0.9
  }

  animate(state) {
    if (state.keys[37]) { // left
      this.velocityX = Math.max(
        this.velocityX - this.accelerationX,
        this.maximumVelocityX * -1,
      )
    }

    if (state.keys[39]) { // right
      this.velocityX = Math.min(
        this.velocityX + this.accelerationX,
        this.maximumVelocityX,
      );
    }
```


C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_5

```
    this.velocityX *= this.frictionX

    let move = true

    state.objects.forEach((object) => {
      if (object === this) {
        return
      }

      const me = this.rectangle
      const you = object.rectangle

      if (me.x < you.x + you.width &&
          me.x + me.width > you.x &&
          me.y < you.y + you.height &&
          me.y + me.height > you.y) {

        if (this.velocityX < 0 && you.x <= me.x) {
          move = false
        }

        if (this.velocityX > 0 && you.x >= me.x) {
          move = false
        }
      }
    })

    if (move) {
      this.rectangle.x += this.velocityX
    }

    this.sprite.x = this.rectangle.x
    this.sprite.y = this.rectangle.y
  }
}

class Box {
  constructor(sprite, rectangle) {
    this.sprite = sprite
    this.rectangle = rectangle
  }

  animate(state) {
    this.sprite.x = this.rectangle.x
    this.sprite.y = this.rectangle.y
  }
}
```

This is from http://codepen.io/assertchris/pen/ALyXKq.

Notice how much code we can delete? The comparisons get a little verbose, but they're nothing a local variable or two can't fix. I also realized that we can move the initial x and y set operations to animate.

Next, I want to encapsulate the event, renderer, and stage logic into a Game class:

```
class Game {
  constructor() {
    this.state = {
      "keys": {},
      "clicks": {},
      "mouse": {},
      "objects": [],
    }

    this.animate = this.animate.bind(this)
  }

  get stage() {
    if (!this._stage) {
      this._stage = this.newStage()
    }

    return this._stage
  }

  set stage(stage) {
    this._stage = stage
  }

  newStage() {
    return new PIXI.Container()
  }

  get renderer() {
    if (!this._renderer) {
      this._renderer = this.newRenderer()
    }

    return this._renderer
  }

  set renderer(renderer) {
    this._renderer = renderer
  }
```

```
  newRenderer() {
    return new PIXI.autoDetectRenderer(
      window.innerWidth,
      window.innerHeight,
      this.newRendererOptions(),
    )
  }

  newRendererOptions() {
    return {
      "antialias": true,
      "autoResize": true,
      "transparent": true,
      "resolution": 2,
    }
  }

  animate() {
    requestAnimationFrame(this.animate)

    this.state.renderer = this.renderer
    this.state.stage = this.stage

    this.state.objects.forEach((object) => {
      console.log(object)
      object.animate(this.state)
    })

    this.renderer.render(this.stage)
  }

  addEventListenerTo(element) {
    element.addEventListener("keydown", (event) => {
      this.state.keys[event.keyCode] = true
    })

    element.addEventListener("keyup", (event) => {
      this.state.keys[event.keyCode] = false
    })

    element.addEventListener("mousedown", (event) => {
      this.state.clicks[event.which] = {
        "clientX": event.clientX,
        "clientY": event.clientY,
      }
    })
```

```
    element.addEventListener("mouseup", (event) => {
      this.state.clicks[event.which] = false
    })

    element.addEventListener("mousemove", (event) => {
      this.state.mouse.clientX = event.clientX
      this.state.mouse.clientY = event.clientY
    })
  }

  addRendererTo(element) {
    element.appendChild(this.renderer.view)
  }

  addObject(object) {
    this.state.objects.push(object)
    this.stage.addChild(object.sprite)
  }
}
```

This is from http://codepen.io/assertchris/pen/ALyXKq.

Notice the class getters and setters. They're useful for filling optional dependencies as needed. We can override `renderer` and `stage` if we need to, but they have sensible defaults too.

The only notable difference here is that we no longer require sprites to be added to the stage separately from adding objects to the state.

I'm in two minds about this. On the one hand, what's going on is much clearer if we add the sprites to the stage by hand. On the other, will we be adding sprites (joined to objects) without adding the objects? I don't think so.

Perhaps we'll come back and change that later. For now, it makes things a little cleaner. Let's add the sprites we had before:

```
const game = new Game()

game.addObject(
  new Box(
    new PIXI.Sprite.fromImage(
      "path/to/sprites/box.png",
    ),
    new PIXI.Rectangle(
      (window.innerWidth / 2) - 150,
      (window.innerHeight / 2) - 35,
      44,
      44,
    ),
  ),
)
```

```
game.addObject(
  new Box(
    new PIXI.Sprite.fromImage(
      "path/to/sprites/box.png",
    ),
    new PIXI.Rectangle(
      (window.innerWidth / 2) + 150,
      (window.innerHeight / 2) + 35,
      44,
      44,
    ),
  ),
)

game.addObject(
  new Player(
    new PIXI.Sprite.fromImage(
      "path/to/sprites/player-idle.png",
    ),
    new PIXI.Rectangle(
      window.innerWidth / 2,
      window.innerHeight / 2,
      44,
      56,
    ),
  ),
)

game.addEventListenerTo(window)
game.addRendererTo(document.body)
game.animate()
```

This is from http://codepen.io/assertchris/pen/ALyXKq.

That looks much better! We now have enough control to set starting points for each game object. But you'll see that after the first frame, the `animate` methods take over. Things like gravity and collisions will start to control how the game progresses. It was always going to be like that, however, so this file now feels like it too.

At this point, we could confine the events and renderer to a smaller set of elements. We could also add any number of objects to the game from here. Everything else game-related is inside the `Game` class. Everything else player- or box-related is inside those classes. It's neat!

Adding gravity to the world

One of the things that make platform games fun is that a moderate amount of physics is at play. First, let's add walls and a floor:

```
const game = new Game()
```

```
game.addObject(
  new Box(
    new PIXI.extras.TilingSprite.fromImage(
      "path/to/sprites/floor-tile.png",
      window.innerWidth,
      64,
    ),
    new PIXI.Rectangle(
      0,
      window.innerHeight - 64,
      window.innerWidth,
      64,
    ),
  ),
)

game.addObject(
  new Box(
    new PIXI.Sprite.fromImage(
      "path/to/sprites/box.png",
    ),
    new PIXI.Rectangle(
      0 + 32,
      window.innerHeight - 44 - 64,
      44,
      44,
    ),
  ),
)

game.addObject(
  new Box(
    new PIXI.Sprite.fromImage(
      "path/to/sprites/box.png",
    ),
    new PIXI.Rectangle(
      window.innerWidth - 32 - 44,
      window.innerHeight - 44 - 64,
      44,
      44,
    ),
  ),
)
```

```
game.addObject(
  new Player(
    new PIXI.Sprite.fromImage(
      "path/to/sprites/player-idle.png",
    ),
    new PIXI.Rectangle(
      window.innerWidth / 2,
      window.innerHeight / 2,
      44,
      56,
    ),
  ),
)
```

This is from http://codepen.io/assertchris/pen/ALyXKq.

Here, we've added a floor as wide as the whole scene. I've used a tiling texture and `new PIXI.extras.TilingSprite.fromImage` to render it on in the scene. I've also moved the two crates to block the player from running off the edge of the floor. They collide in the same way as before. Now, let's see about adding that gravity:

```
class Player {
  constructor(sprite, rectangle) {
    this.sprite = sprite
    this.rectangle = rectangle

    this.velocityX = 0
    this.maximumVelocityX = 8
    this.accelerationX = 2
    this.frictionX = 0.9

    this.velocityY = 0
    this.maximumVelocityY = 30
    this.accelerationY = 3
    this.jumpVelocity = -30

    this.isOnGround = false
  }

  animate(state) {
    if (state.keys[37]) { // left
      this.velocityX = Math.max(
        this.velocityX - this.accelerationX,
        this.maximumVelocityX * -1,
      )
    }
```

```
if (state.keys[39]) { // right
  this.velocityX = Math.min(
    this.velocityX + this.accelerationX,
    this.maximumVelocityX,
  )
}

this.velocityX *= this.frictionX

this.velocityY = Math.min(
  this.velocityY + this.accelerationY,
  this.maximumVelocityY,
)

state.objects.forEach((object) => {
  if (object === this) {
    return
  }

  var me = this.rectangle
  var you = object.rectangle

  if (me.x < you.x + you.width &&
      me.x + me.width > you.x &&
      me.y < you.y + you.height &&
      me.y + me.height > you.y) {

    if (this.velocityY > 0 && you.y >= me.y) {
      this.velocityY = 0
      return
    }

    if (this.velocityY < 0 && you.y <= me.y) {
      this.velocityY = this.accelerationY
      return
    }

    if (this.velocityX < 0 && you.x <= me.x) {
      this.velocityX = 0
      return
    }

    if (this.velocityX > 0 && you.x >= me.x) {
      this.velocityX = 0
      return
    }
  }
})
```

```
    this.rectangle.x += this.velocityX
    this.rectangle.y += this.velocityY

    this.sprite.x = this.rectangle.x
    this.sprite.y = this.rectangle.y
  }
}
```

This is from http://codepen.io/assertchris/pen/ALyXKq.

We begin by creating a set of properties to match the ones we made to track horizontal movement. We don't need vertical friction since that level of detail is often omitted from platform games.

We also have to track vertical and horizontal collisions. When the collision is with the player and a platform/floor, then we stop the downward velocity. When it's against a ceiling, we replace upward velocity with the force of gravity.

Allowing players to jump

Jumping is simply reversing gravity for a short time:

```
animate(state) {
  if (state.keys[37]) { // left
    this.velocityX = Math.max(
      this.velocityX - this.accelerationX,
      this.maximumVelocityX * -1,
    )
  }

  if (state.keys[39]) { // right
    this.velocityX = Math.min(
      this.velocityX + this.accelerationX,
      this.maximumVelocityX,
    )
  }

  this.velocityX *= this.frictionX

  this.velocityY = Math.min(
    this.velocityY + this.accelerationY,
    this.maximumVelocityY,
  )

  state.objects.forEach((object) => {
    if (object === this) {
      return
    }
```

```
    var me = this.rectangle
    var you = object.rectangle

    // ...snip

  if (state.keys[32] && this.isOnGround) {
      this.velocityY = this.jumpVelocity
      this.isOnGround = false
  }

  this.rectangle.x += this.velocityX
  this.rectangle.y += this.velocityY

  this.sprite.x = this.rectangle.x
  this.sprite.y = this.rectangle.y
}
```

This is from http://codepen.io/assertchris/pen/ALyXKq.

With this code, we've mapped the space key to jump. We add the keyboard check after the collision check because we only want our player to jump if they are standing on a platform or floor.

Now it' possible to create levels out of boxes, to give some of them visible textures, and to jump around them. Spend some time making a level and jumping around in it!

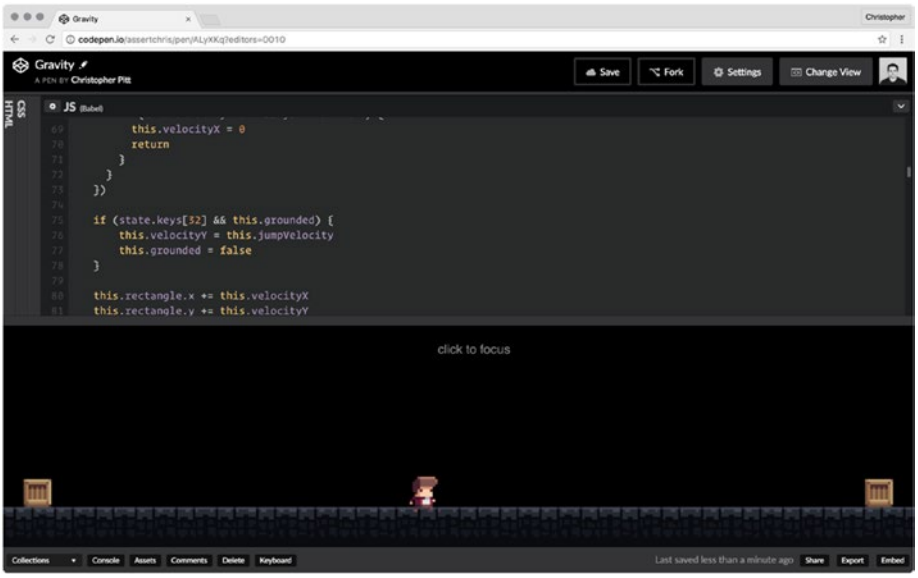

Figure 5-1. *Adding gravity*

Summary

In this chapter, we cleaned up the game code. Each part is now self-contained—from the box to the player to the game itself. That will make it easier for the individual game objects to manage themselves.

We also created a floor and basic walls (out of crates) so that the player won't fall out of the world. Finally, we added the ability for the player to jump. It's now possible to construct a traversable level!

CHAPTER 6

Ladders

It would be pretty limiting if players could only move upward by jumping through gaps, or by jumping on boxes. Those aren't the only options, though. We still have to learn about elevators, stairs, and ladders. Let's start by building a ladder!

Creating our first ladder

Let's begin our ladder by copying the `Box` class:

```
class Ladder {
  constructor(sprite, rectangle) {
    this.sprite = sprite
    this.rectangle = rectangle
  }

  animate(state) {
    this.sprite.x = this.rectangle.x
    this.sprite.y = this.rectangle.y
  }
}

// ...later

game.addObject(
  new Box(
    new PIXI.Sprite.fromImage(
      "path/to/sprites/platform.png",
    ),
    new PIXI.Rectangle(
      window.innerWidth - 400,
      window.innerHeight - 64 - 200,
      256,
      64,
    ),
  ),
)
```


C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_6

```
game.addObject(
  new Ladder(
    new PIXI.extras.TilingSprite.fromImage(
      "path/to/sprites/ladder.png",
      44,
      200,
    ),
    new PIXI.Rectangle(
      window.innerWidth - 250,
      window.innerHeight - 64 - 200,
      44,
      200,
    ),
  ),
)
```

This is from http://codepen.io/assertchris/pen/jrzrPw.

You'll need to make `ladder.png` and `platform.png` images. Be sure to add these to the game before adding the player, or the ladder sprites will be in front of the player sprite.

You'll notice that the player bumps into the ladder as if it were a box. We'll need to add a few getters to our boxes and ladders so that the collision detection can decide whether they still collide with each other:

```
state.objects.forEach((object) => {
  if (object === this) {
    return
  }

  const me = this.rectangle
  const you = object.rectangle
  const collides = object.collides

  if (me.x < you.x + you.width &&
      me.x + me.width > you.x &&
      me.y < you.y + you.height &&
      me.y + me.height > you.y) {

    if (collides && this.velocityY > 0 && you.y >= me.y) {
      this.isOnGround = true
      this.velocityY = 0
      return
    }

    if (collides && this.velocityY < 0 && you.y <= me.y) {
      this.velocityY = this.accelerationY
      return
    }
```

```
    if (collides && this.velocityX < 0 && you.x <= me.x) {
      this.velocityX = 0
      return
    }

    if (collides && this.velocityX > 0 && you.x >= me.x) {
      this.velocityX = 0
      return
    }
  }
})
```

This is from http://codepen.io/assertchris/pen/jrzrPw.

We give `Box` and `Ladder` a `collides` property so that `Player.animate` can ignore collisions with objects that players shouldn't collide with. If we were going to allow multiple players in the same game/level, then we would also add a `collides` property to `Player`. That is, unless we wanted multiple players to collide with each other.

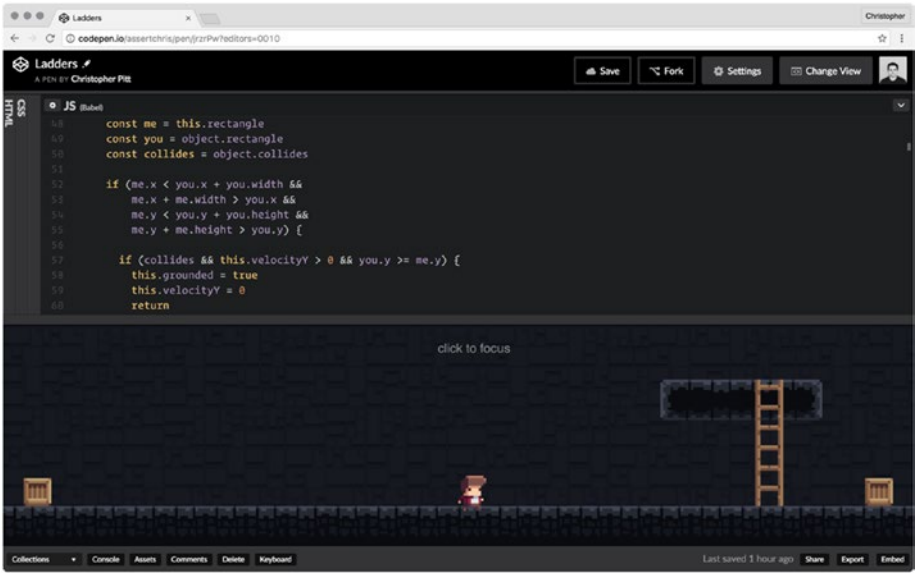

Figure 6-1. *Platforms and ladders*

Allowing players to climb ladders

In order for players to be able to climb ladders, we have to be able to tell if they're trying to climb one. We also then have to suspend gravity and side motion so they don't fall off or slide off:

```
class Player {
  constructor(sprite, rectangle) {
```

```
    this.sprite = sprite
    this.rectangle = rectangle

    this.velocityX = 0
    this.maximumVelocityX = 8
    this.accelerationX = 2
    this.frictionX = 0.9

    this.velocityY = 0
    this.maximumVelocityY = 30
    this.accelerationY = 3
    this.jumpVelocity = -30

    this.climbingSpeed = 10

    this.isOnGround = false
    this.isOnLadder = false
  }

  animate(state) {
    if (state.keys[37]) { // left
      this.velocityX = Math.max(
        this.velocityX - this.accelerationX,
        this.maximumVelocityX * -1,
      )
    }

    if (state.keys[39]) { // right
      this.velocityX = Math.min(
        this.velocityX + this.accelerationX,
        this.maximumVelocityX,
      )
    }

    this.velocityX *= this.frictionX

    this.velocityY = Math.min(
      this.velocityY + this.accelerationY,
      this.maximumVelocityY,
    )

    state.objects.forEach((object) => {
      if (object === this) {
        return
      }

      const me = this.rectangle
      const you = object.rectangle
```

```
      const collides = object.collides

      if (me.x < you.x + you.width &&
          me.x + me.width > you.x &&
          me.y < you.y + you.height &&
          me.y + me.height > you.y) {

        if (object.constructor.name === "Ladder") {
          if (state.keys[38] || state.keys[40]) {
            this.isOnLadder = true
            this.isOnGround = false
            this.velocityY = 0
            this.velocityX = 0
          }

          if (state.keys[38]) {
            this.rectangle.y -= this.climbingSpeed
          }

          if (state.keys[40] && ➥
            me.y + me.height < you.y + you.height) {
            this.rectangle.y += this.climbingSpeed
          }
        }

        // ...snip

      }
    })

    if (state.keys[32] && this.isOnGround) {
        this.velocityY = this.jumpVelocity
        this.isOnGround = false
    }

    this.rectangle.x += this.velocityX

    if (!this.isOnLadder) {
      this.rectangle.y += this.velocityY
    }

    this.sprite.x = this.rectangle.x
    this.sprite.y = this.rectangle.y
  }
}
```

This is from http://codepen.io/assertchris/pen/jrzrPw.

In this code, we create an `isOnLadder` variable so we can tell if the player is standing still on a ladder. During the usual collision detection, we note whether the object the player is colliding with is a ladder or not. If so, and they are pressing the up arrow, we start them climbing. `isOnLadder` is only set if they are pressing the up arrow, which is why we need that variable to begin with.

We then reset player velocity and the properties related to jumping. We also directly alter the player rectangle. If the up arrow is being pressed, we move the player up.

Summary

In this chapter, we added ladders and stationary platforms. Our players can now jump and climb ladders to navigate levels. These are two essential elements of 2D platform games.

You should know what you'd like your game to look like at this point. I've invested in a great sprite pack. You can find others at Graphic River (`https://graphicriver.net`).

CHAPTER 7

Stairs

We've implemented jumping and climbing as the main ways to traverse a level vertically. These actions are great, but they limit level design to consist of platforms and ladders. Let's expand our design options by implementing stairs (or the similar action of walking up slopes).

We'll need to create even more sprite artwork for this part and build more complex boxes composed of multiple sprite types. Let me show you what I mean....

Building a slope

I've gone ahead and created new slope sprites. We also need to add a couple of new object types: decals and slopes. Decals are going to be a noncolliding type, whereas slopes are going to be identifiable by the player's collision detection algorithm:

```
class LeftSlope {
  get collides() {
    return false
  }

  constructor(sprite, rectangle) {
    this.sprite = sprite
    this.rectangle = rectangle
  }

  animate(state) {
    this.sprite.x = this.rectangle.x
    this.sprite.y = this.rectangle.y
  }
}

class RightSlope {
  get collides() {
    return false
  }
```


C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_7

```
  constructor(sprite, rectangle) {
    this.sprite = sprite
    this.rectangle = rectangle
  }

  animate(state) {
    this.sprite.x = this.rectangle.x
    this.sprite.y = this.rectangle.y
  }
}

class Decal {
  get collides() {
    return false
  }

  constructor(sprite, rectangle) {
    this.sprite = sprite
    this.rectangle = rectangle
  }

  animate(state) {
    this.sprite.x = this.rectangle.x
    this.sprite.y = this.rectangle.y
  }
}

// ...later

game.addObject(
  new LeftSlope(
    new PIXI.Sprite.fromImage(
      "path/to/sprites/slope-left.png",
    ),
    new PIXI.Rectangle(
      0 + 250,
      window.innerHeight - 64 - 64 + 1,
      64,
      64,
    ),
  ),
)

game.addObject(
  new RightSlope(
    new PIXI.Sprite.fromImage(
      "path/to/sprites/slope-right.png",
    ),
```

```
    new PIXI.Rectangle(
      0 + 250 + 64 + 128,
      window.innerHeight - 64 - 64 + 1,
      64,
      64,
    ),
  ),
)

game.addObject(
  new Decal(
    new PIXI.Sprite.fromImage(
      "path/to/sprites/hill-base.png",
    ),
    new PIXI.Rectangle(
      0 + 250,
      window.innerHeight - 64 + 1,
      128,
      64,
    ),
  ),
)

game.addObject(
  new Box(
    new PIXI.Sprite.fromImage(
      "path/to/sprites/hill-top.png",
    ),
    new PIXI.Rectangle(
      0 + 250 + 64,
      window.innerHeight - 64 - 64 + 1,
      128,
      64,
    ),
  ),
)
```

This is from http://codepen.io/assertchris/pen/dpmOEJ.

We need to add these objects after the floor, or the floor will obscure them. Together, they create a pleasing little hill. We can jump on top of it, but things start to get hairy when we try to climb up or down these objects. Still, it looks good, as you can see in Figure 7-1:

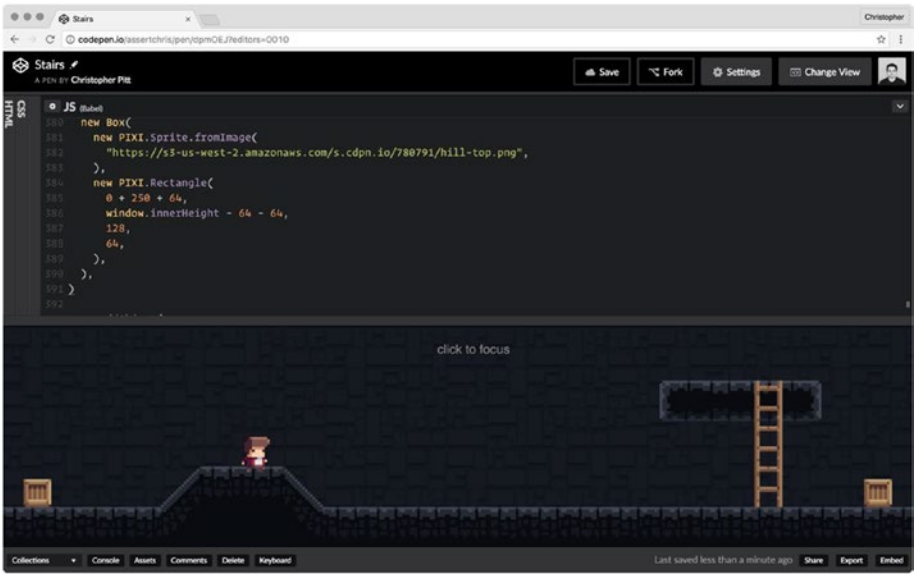

Figure 7-1. *A Little hill*

Walking up slopes

To get our player moving up slopes, we need to once again adjust the collision detection algorithm:

```
if (me.x < you.x + you.width &&
    me.x + me.width > you.x &&
    me.y < you.y + you.height &&
    me.y + me.height > you.y) {

  if (object.constructor.name === "LeftSlope") {
    const meCenter = Math.round(me.x + (me.width / 2))
    const youRight = you.x + you.width
    const youBottom = you.y + you.height
    const highest = you.y - me.height
    const lowest = youBottom - me.height

    this.isOnGround = true
    this.isOnSlope = true

    me.y = lowest - (meCenter - you.x)
    me.y = Math.max(me.y, highest)
    me.y = Math.min(me.y, lowest)

    if (me.y >= lowest || me.y <= highest) {
```

```
      this.isOnSlope = false
    }

    return
  }

  if (object.constructor.name === "RightSlope") {
    const meCenter = Math.round(me.x + (me.width / 2))
    const youBottom = you.y + you.height
    const highest = you.y - me.height
    const lowest = youBottom - me.height

    this.isOnGround = true
    this.isOnSlope = true

    me.y = highest + (meCenter - you.x)
    me.y = Math.max(me.y, highest)
    me.y = Math.min(me.y, lowest)

    if (me.y >= lowest || me.y <= highest) {
      this.isOnSlope = false
    }

    return
  }

  if (collides && this.velocityY > 0 && you.y >= me.y) {
    this.isOnGround = true
    this.velocityY = 0
    return
  }

  // ...remaining collision detection code
}

if (state.keys[32] && this.isOnGround) {
    this.velocityY = this.jumpVelocity
    this.isOnGround = false
    this.isOnSlope = false
}
```

This is from http://codepen.io/assertchris/pen/dpmOEJ.

If the player collides with the `LeftSlope`, we don't collide as we did with boxes. Instead, we decrease the player's y coordinate (which moves the player sprite upward) by however far to the right of the slope the player is. In other words, as the player's `x` increases, their top decreases.

For the `RightSlope`, we just swap the `x` and `y` relationship, moving the player down as their `x` increases. We also want the player to be able to jump up the slope, so we need to set `isOnGround` to `true`.

Finally, if the player's bottom edge is the same (or less) than the slope's top edge, we set `isOnSlope` to `false`, so that the player will move correctly on the horizontal axis.

Staying above the floor

If you've been following along with the code, you've probably noticed a strange visual bug. It seems the player can fall partway through the floor before the collision detection stops them.

We can get around this bug by resetting the `y` player coordinates for the appropriate collision:

```
if (collides && this.velocityY > 0 && you.y >= me.y) {
  me.y = you.y - me.height + 1
  this.isOnGround = true
  this.velocityY = 0
  return
}

if (collides && this.velocityY < 0 && you.y <= me.y) {
  this.velocityY = this.accelerationY
  return
}

if (collides && this.velocityX < 0 && you.x <= me.x) {
  this.velocityX = 0
  return
}

if (collides && this.velocityX > 0 && you.x >= me.x) {
  this.velocityX = 0
  return
}
```

This is from `http://codepen.io/assertchris/pen/dpmOEJ`.

Now, when the player bumps up against something, it'll be pushed just outside again. This will stop the player from sinking into the floor.

Summary

In this chapter, we learned how to make slopes the player can walk up (without needing to jump). These concepts translate to the implementation of stairs in an old castle.

Try creating stairs of your own. Perhaps your stairs will be steeper or shallower than mine, and you'll need to adjust the rate at which your player ascends or descends.

CHAPTER 8

Camera Locking

So far, we've limited the levels to the size of our browser window. That's okay for a proof of concept, but real levels require more space than we're currently able to give.

I've got a couple things planned to overcome this limitation. First we're going to lock the camera to the player so that it moves as the player moves. Then we're going to increase the size of the level so that it can be any size we need.

Wrapping with a camera

PixiJS doesn't support any kind of free-moving camera through which to render the scene, but we don't need one. Instead of moving the renderer or scene, we can wrap it in an HTML element and move that. We'll need to change how we embed the game:

```
<div class="camera"></div>
<div class="focus-target">click to focus</div>

------

.camera, .focus-target {
  width: 100%;
  height: 100%;
  position: absolute;
  top: 0;
  left: 0;
}

.focus-target {
  padding: 25px;
  text-align: center;
}

------

game.addEventListenerTo(window)
game.addRendererTo(document.querySelector(".camera"))
game.animate()
```

These are from http://codepen.io/assertchris/pen/WGzkym.

C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_8

We begin by creating a camera that takes up the entire screen space. Then, instead of appending the scene to `document.body`, we append it to `.camera` instead. This gives us the ability to transform `.camera` in interesting ways:

```
animate() {
  requestAnimationFrame(this.animate)

  this.state.renderer = this.renderer
  this.state.stage = this.stage

  this.state.objects.forEach((object) => {
    object.animate(this.state)
  })

  if (this.player) {
    const offsetLeft = Math.round(
      this.player.rectangle.x - (window.innerWidth / 2)
    ) * -1

    const offsetTop = Math.round(
      this.player.rectangle.y - (window.innerHeight / 2)
    ) * -1

    this.element.style = `
      transform:
      scale(1.2)
      translate(${offsetLeft}px)
      translateY(${offsetTop}px)
    `
  }

  this.renderer.render(this.stage)
}

// ...later

const player = new Player(
  new PIXI.Sprite.fromImage(
    "path/to/sprites/player-idle.png",
  ),
  new PIXI.Rectangle(
    Math.round(window.innerWidth / 2),
    Math.round(window.innerHeight / 2),
    44,
    56,
  ),
)
```

```
game.addObject(player)
game.player = player
```

This is from http://codepen.io/assertchris/pen/WGzkym.

We need to create the `player` object slightly differently. Instead of adding it directly to `game`, we declare it as a constant. We still add it to `game`, but we also assign it as a property.

This means we can reference it inside `Game.animate`. We get the `x` and `y` coordinates of the player, and then subtract half the screen `innerWidth` and `innerHeight` so that the player will be roughly in the center of the screen.

Then we use CSS transformations to move `.camera` left and up by the amounts we just calculated. We can also scale the camera up a little, so things look slightly zoomed in (see Figure 8-1). As this code is in `Game.animate`, it will update every time the player is rendered, which means it will move as the player moves.

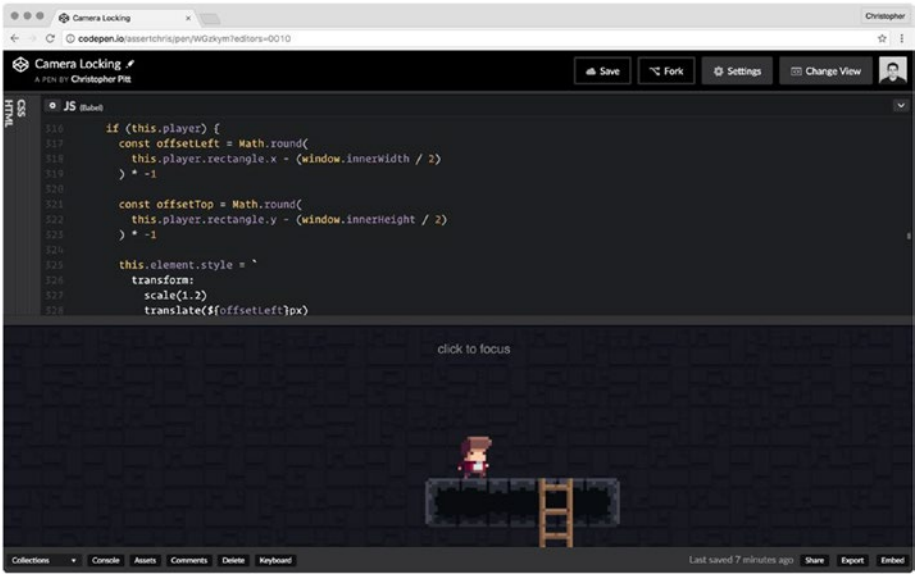

Figure 8-1. *Zoomed and locked*

Growing levels

This leads to an interesting discovery. When the player is standing on top of my platform and then jumps, they go through the top of the rendered scene (and disappear). This is due to the fixed height of the scene (which is the same as the window height).

To fix this, we need to change how we define the game's dimensions and how we place objects inside it:

```
constructor(w, h) {
  this.w = w
  this.h = h
```

```
  this.state = {
    "keys": {},
    "clicks": {},
    "mouse": {},
    "objects": [],
  }

  this.animate = this.animate.bind(this)
}

newRenderer() {
  return new PIXI.autoDetectRenderer(
    this.w, this.h, this.newRendererOptions(),
  )
}

// ...later

const width = window.innerWidth
const height = window.innerHeight + 200

const game = new Game(
  width,
  height,
)

game.addObject(
  new Box(
    new PIXI.extras.TilingSprite.fromImage(
      "path/to/sprites/floor-tile.png",
      width,
      64,
    ),
    new PIXI.Rectangle(
      0,
      height - 64,
      width,
      64,
    ),
  ),
)
// ...remaining object definitions
```

This is from http://codepen.io/assertchris/pen/WGzkym.

We change the `Game` to accept `width` and `height` constructor parameters. Then, instead of using `window.innerWidth` and `window.innerHeight` everywhere, we use the same width and height we provide to `game`.

Until now, we've added our game objects relative to the screen edges. This allows us to change the width and height quite easily, without have to reposition all our game objects. Now we need to replace all instances of the previous (global) widths and heights with the `width` and `height` constants.

The only place we don't remove the global references is in the camera locking code we just added. These need to be relative to the window (not the entire level size), so they can stay as they are.

Summary

In this chapter, we learned how to expand our levels beyond the confines of the window. Now that we can lock the camera to the player, we can create sprawling labyrinths and awe-inspiring castles.

Take some time to consider how you'd improve the background and borders of each level so they're not as empty as mine are. Perhaps you might want to create animated backgrounds that express the tone and location of each level.

CHAPTER 9

Projectiles

Our game is full of keyboard control, but what about the mouse? This chapter is all about projectiles; how to fire them and how they move.

In this chapter, we're going to add a few new, popular keyboard controls (WASD), and mouse aiming and shooting. Along the way we'll add a custom crosshair. This is going to be fun....

Custom crosshairs

Before we can display custom crosshairs, we need a way to disable the default mouse cursor. Fortunately for us, CSS already includes this mechanism:

```
body {
  background: url("path/to/sprites/background.png");
  color: grey;
  font-family: helvetica, arial;
  font-size: 20px;
  cursor: none;
}
```

This is from http://codepen.io/assertchris/pen/YGaYvy.

Just by adding `cursor: none` to `body`, we can hide the default cursor from the game screen. Then, let's add the crosshair as a new decal:

```
const crosshair = new Decal(
  new PIXI.Sprite.fromImage(
    "path/to/sprites/crosshair.png",
  ),
  new PIXI.Rectangle(
    0, 0, 18, 18,
  ),
)
```


C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_9

```
game.addObject(crosshair)
game.crosshair = crosshair
```

This is from http://codepen.io/assertchris/pen/YGaYvy.

The crosshair can now be seen at the top left (zoom out in your browser window) of the game screen. That's not where we want it, though. What we really want is for it to be in proximity to the player, but at an angle directly between the center of the player and the mouse cursor.

Let's capture the cursor position and calculate the angle between the mouse cursor and the player:

```
element.addEventListener("mousemove", (event) => {
  this.state.mouse.clientX = event.clientX
  this.state.mouse.clientY = event.clientY

  const rect = this.player.rectangle

  const centerX = (window.innerWidth / 2) + (rect.width / 2)
  const centerY = (window.innerHeight / 2) + (rect.height / 2)

  const deltaX = event.clientX - centerX
  const deltaY = centerY - event.clientY

  this.state.angle = Math.atan2(deltaY, deltaX)
})
```

This code is from http://codepen.io/assertchris/pen/YGaYvy.

This code is a little tricky to read, but all it's doing is getting the width and height of a triangle between the mouse cursor and the player. We get the angle from that triangle and store it in the game state. The cursor is going to use this (once we replace `cursor = new Decal` with `cursor = new Cursor`):

```
class Ladder extends Box {
  get collides() {
    return false
  }
}

class LeftSlope extends Box {
  get collides() {
    return false
  }
}
```

```
class RightSlope extends Box {
  get collides() {
    return false
  }
}

class Decal extends Box {
  get collides() {
    return false
  }
}

class Crosshair extends Decal {
  animate(state) {
    const rect = state.player.rectangle

    const centerX = rect.x + (rect.width / 2)
    const centerY = rect.y + (rect.height / 2)
    const radius = 70

    const targetX = centerX + Math.cos(state.angle) * radius
    const targetY = centerY - Math.sin(state.angle) * radius

    this.sprite.x = targetX
    this.sprite.y = targetY
  }
}
```

This code is from http://codepen.io/assertchris/pen/YGaYvy.

Before we dive into the `Crosshair.animate` method, notice how I've shortened the definitions of `Ladder`, `LeftSlope`, `RightSlope`, and `Decal`? We can use the `extends` keyword to inherit the behavior of `Box`.

■ **Note** Inheritance doesn't always lead to good code architecture, but in this simple case, it works well for us.

Using the angle we calculated (in the `mousemove` event listener), we calculate the point where the crosshair needs to be rendered (see Figure 9-1).

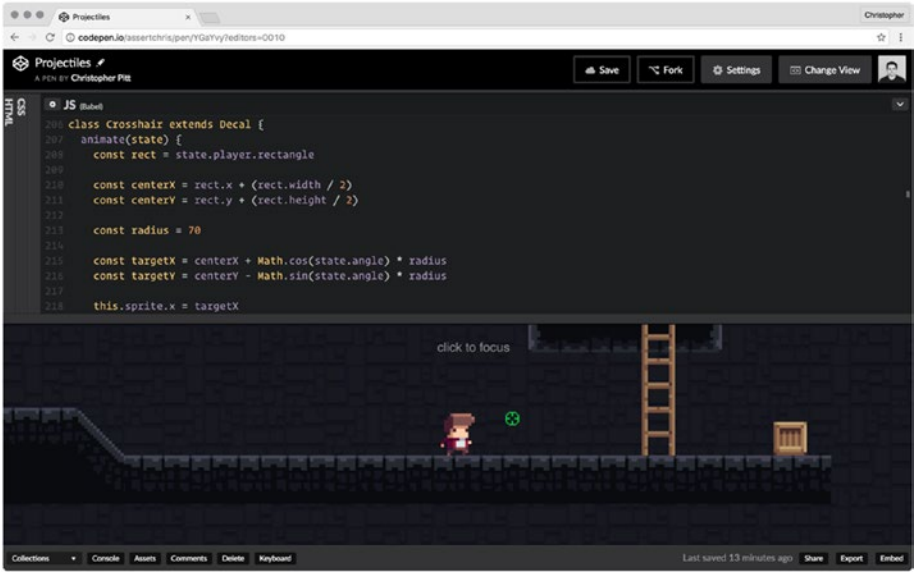

Figure 9-1. *Fixed crosshair*

Custom keys

Until now, we've been moving the player with the arrow keys. That's okay if we only need to use the keyboard, but we've just added mouse aim. Let's add the popular WASD movement keys to `Player.animate`:

```
animate(state) {
  if (state.keys[37] || state.keys[65]) { // left
    this.velocityX = Math.max(
      this.velocityX - this.accelerationX,
      this.maximumVelocityX * -1,
    )
  }

  if (state.keys[39] || state.keys[68]) { // right
    this.velocityX = Math.min(
      this.velocityX + this.accelerationX,
      this.maximumVelocityX,
    )
  }

   // ...velocity calculations
```

```
  state.objects.forEach((object) => {
    if (object === this) {
      return
    }

    const me = this.rectangle
    const you = object.rectangle
    const collides = object.collides

    if (me.x < you.x + you.width &&
        me.x + me.width > you.x &&
        me.y < you.y + you.height &&
        me.y + me.height > you.y) {

      if (object.constructor.name === "Ladder") {
        if (state.keys[38] || state.keys[40] || ↩
          state.keys[87] || state.keys[83]) {
          this.isOnLadder = true
          this.isOnGround = false
          this.velocityY = 0
          this.velocityX = 0
        }

        if (state.keys[38] || state.keys[87]) {
          this.rectangle.y -= this.climbingSpeed
        }

        if (state.keys[40] || state.keys[83] && ↩
          me.y + me.height < you.y + you.height) {
          this.rectangle.y += this.climbingSpeed
        }

        if (me.y <= you.x - me.height) {
          this.isOnLadder = false
        }

        return
      }

      // ...remaining collision detection
    }
  })

  // ...remaining calculations
}
```

This is from http://codepen.io/assertchris/pen/YGaYvy.

We've added `state.keys[65]` and `state.keys[68]` as alternative left and right keys. Then, for moving up and down ladders, we've also added `state.keys[87]` and `state.keys[83]`.

Shooting

Let's finish up by adding the ability to shoot in the direction of the crosshair. To do so, we need to create another type of object:

```
class Bullet extends Decal {
  animate(state) {
    const rect = state.player.rectangle

    this.x = this.x || rect.x + rect.width
    this.y = this.y || rect.y + (rect.height / 2)

    this.angle = this.angle || state.angle
    this.rotation = this.rotation || state.rotation

    this.radius = this.radius || 0
    this.radius += 15

    const targetX = this.x + Math.cos(this.angle) * this.radius
    const targetY = this.y - Math.sin(this.angle) * this.radius

    this.sprite.x = targetX
    this.sprite.y = targetY
    this.sprite.rotation = this.rotation
  }
}
```

This is from http://codepen.io/assertchris/pen/YGaYvy.

`Bullet` animates differently to most of the other object types in that it modifies its state with each tick. The pattern of `this.x = this.x || something` is useful for initializing values once. It's important that we store things like `x`, `y`, `angle`, and `rotation` only when the bullet is fired, or they will continue to change as we move the player or crosshair.

The `radius` starts at the front of the player and increases in the direction the crosshair was when the bullet was fired. We also rotate the bullet sprite to match the same angle. This requires us to store `rotation` alongside `angle` in the mouse event listener. We also have to create new bullets when the mouse is clicked:

```
element.addEventListener("mousedown", (event) => {
  this.state.clicks[event.which] = {
    "clientX": event.clientX,
    "clientY": event.clientY,
  }
```

```
  if (event.button === 0) { // left click
    const rect = this.player.rectangle

    const bullet = new Bullet(
      new PIXI.Sprite.fromImage(
        "path/to/sprites/bullet.png",
      ),
      new PIXI.Rectangle(
        rect.x + rect.width, rect.y, 8, 8,
      ),
    )

    this.addObject(bullet)

    setTimeout(() => {
      this.removeObject(bullet)
    }, 250)
  }
})

element.addEventListener("mousemove", (event) => {
  this.state.mouse.clientX = event.clientX
  this.state.mouse.clientY = event.clientY

  const rect = this.player.rectangle

  const centerX = (window.innerWidth / 2) + (rect.width / 2)
  const centerY = (window.innerHeight / 2) + (rect.height / 2)

  const deltaX = event.clientX - centerX
  const deltaY = centerY - event.clientY

  const rotationX = event.clientX - centerX
  const rotationY = event.clientY - centerY

  this.state.angle = Math.atan2(deltaY, deltaX)
  this.state.rotation = Math.atan2(rotationY, rotationX)
})

// ...remaining event listeners
```

This is from http://codepen.io/assertchris/pen/YGaYvy.

We calculate the `rotation` similarly to how we calculate the `angle`, but we flip the vertical axis. When the player clicks, we create a new `bullet` (positioned in front of the player) and add it to the game.

After `250` milliseconds, we want to remove the bullet. This is so that bullets don't slow down the animation cycle when they're no longer required. We need to add the `removeObject` method to the `Game` class:

```
removeObject(object) {
  this.state.objects = this.state.objects.filter(
    function(next) {
      return next !== object
    }
  )

  this.stage.removeChild(object.sprite)
}
```

This is from http://codepen.io/assertchris/pen/YGaYvy.

We'll explore more projectile behavior when we create mobs to shoot in the next chapter.

Summary

In this chapter, we looked at ways to create custom cursors and shoot projectiles. We discovered a bit of helpful trigonometry to constrain a crosshair to a certain radius around the player and then to push bullets beyond that.

Experiment with the speed and appearance of your projectiles. Perhaps you're building a medieval game, and your projectiles are magical bolts. Think about adding vertical acceleration so your bullets drop to the ground after some distance.

CHAPTER 10

Mobs

What fun would this game be if the player were to go through it all alone? What we need now are mindless mobs (or blobs if you prefer) to patrol and/or otherwise complicate the hero's path.

Patrolling blobs

In this chapter, we're going to add patrolling blobs, which means we need another class to encapsulate their logic:

```
class Blob extends Box {
  constructor(sprite, rectangle) {
    super(sprite, rectangle)

    this.limit = 200
    this.left = true
  }

  animate(state) {
    if (this.left) {
      this.rectangle.x -= 2
    }

    if (!this.left) {
      this.rectangle.x += 2
    }

    this.limit -= 2
```


C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_10

```
    if (this.limit <= 0) {
      this.left = !this.left
      this.limit = 200
    }

    this.sprite.x = this.rectangle.x
    this.sprite.y = this.rectangle.y
  }
}
```

This is from http://codepen.io/assertchris/pen/XjEExz.

This time we're setting a couple of properties in the constructor. We still want the parent constructor applied, so we call `super`, providing the expected `sprite` and `rectangle` parameters.

Then, in the `animate` method, we move the blob `2` pixels to the left or right (depending on whether the blob is moving `left` or not). Once the blob moves `200` pixels in the same direction, we turn it around (and reset the `200` pixels for the other direction).

We should dot a few of these around our level:

```
game.addObject(
  new Blob(
    new PIXI.Sprite.fromImage(
      "path/to/sprites/blob-idle-1.png",
    ),
    new PIXI.Rectangle(
      width - 450,
      height - 64 - 48,
      48,
      48,
    ),
  ),
)
```

This is from http://codepen.io/assertchris/pen/XjEExz.

This code will place the blob next to the starting position of the player as shown in Figure 10-1 (at least in my level). It's fun to watch it move back and forth and even have the player jump on its head.

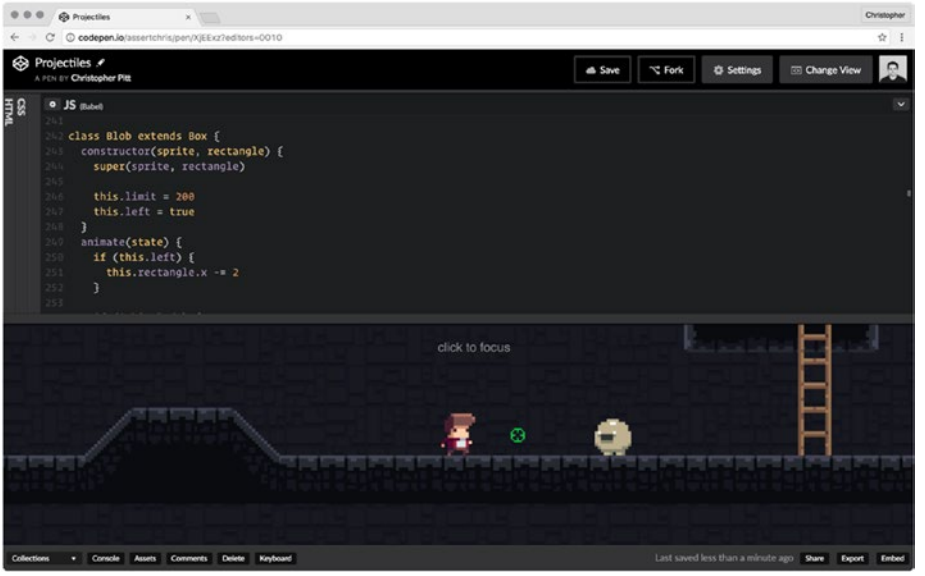

Figure 10-1. *Patrolling blob*

Shooting mobs

In the last chapter, we added the ability for our player to shoot projectiles. Let's put those to use by adding some collision detection to the bullets themselves:

```
class Bullet extends Decal {
  animate(state) {
    const rect = state.player.rectangle

    this.x = this.x || rect.x + rect.width
    this.y = this.y || rect.y + (rect.height / 2)
    this.angle = this.angle || state.angle
    this.rotation = this.rotation || state.rotation
    this.radius = this.radius || 0

    this.radius += 10

    const targetX = this.x + Math.cos(this.angle) * this.radius
    const targetY = this.y - Math.sin(this.angle) * this.radius

    this.rectangle.x = targetX
    this.rectangle.y = targetY
```

```
    this.sprite.x = targetX
    this.sprite.y = targetY
    this.sprite.rotation = this.rotation

    state.objects.forEach((object) => {
      if (object === this) {
        return
      }

      const me = this.rectangle
      const you = object.rectangle

      if (me.x < you.x + you.width &&
          me.x + me.width > you.x &&
          me.y < you.y + you.height &&
          me.y + me.height > you.y) {

        if (object.constructor.name === "Blob") {
          state.game.removeObject(object)
          state.game.removeObject(this)
        }
      }
    })
  }
}
```

This is from http://codepen.io/assertchris/pen/XjEExz.

This collision detection logic is similar to the first pass we did in the `Player` class. We don't need to check for things like slopes or ladders. All we're interested in is whether the bullet collides with a blob. If so, we remove both from the game.

■ **Note** I've also slightly decreased the bullet speed. It's not very realistic to think that a player can almost catch up to a speeding bullet, but it does feel a bit better this way.

It's not as elegant as if the blob had steadily decreasing health, but we'll revisit them topic in a bit.

Summary

In this chapter, we added a simple kind of mob to the game. We also empowered our bullets to dispatch these mobs. They're not the most intelligent (or even durable) mobs, but they are a start.

This is where you can be really creative. What if you added gravity to the movement of the mobs? Or made them stationary until the player moved closer? What if they could shoot back?

CHAPTER 11

Health

I've played very few games where the first mistake leads to instant failure. Usually, there's quite a lead-up to the final moment of failure. Sonic (https://en.wikipedia.org/wiki/Sonic_the_Hedgehog) loses his rings, Mario loses his powers.

The goal of this chapter is for us to implement a health system so that the player has a chance to make mistakes and learn from them.

Taking damage

There is only one way the player can fail (in our game, so far): jumping off the edge of the level. That's not a realistic failure condition, however, because well-constructed levels will be built up in such a way that the player can never reach outside the bounds of the level.

Before we can work out the details of loss of health, we need to introduce another mechanism to cause the health loss. Here, we're going to introduce a mechanic that causes us to lose health (and momentarily lose control) when we come into contact with a slime.

To start with, we're going to have to introduce more collision detection logic in `Player.animate`:

```
if (object.constructor.name === "Blob" && !this.invulnerable) {
  if (this.velocityX >= 0) {
    this.velocityX = -10
  } else {
    this.velocityX = 10
  }

  this.velocityY *= -1

  this.invulnerable = true
  this.sprite.alpha = 0.5

  setTimeout(() => {
    this.invulnerable = false
    this.sprite.alpha = 1
  }, 2000)
```


C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_11

```
  if (typeof this.onHurt === "function") {
    this.onHurt.apply(this)
  }
}
```

This is from http://codepen.io/assertchris/pen/qaoyPo.

We previously added collision detection, specific to the `Blob`, to `Bullet.animate`. Now, we're adding it to `Player.animate` so that the player will "take damage" when they come into contact with a `Blob`.

■ **Note** I've hard-coded a lot of things here and previously. You are free to abstract the hard-coded values out, but I've opted not to do that in every instance to save time and to keep things simple. For instance, you could remove the `alpha` logic entirely and provide the constructor with the `invulnerable` duration.

Now when the player and the blob connect, the player is thrown up and backward from the blob. The player also enters an invulnerable state, which means they won't lose all health within 2000 milliseconds. Now that we have a way to hurt the player, let's do something about it.

Showing health

Notice that bit about `onHurt`? I didn't want to hard-code the interface changes that will display the player's current health. By invoking a user-provided function, we can outsource that behavior to the code that creates the player.

It's tempting to try and fit everything into the PixiJS model, but we're coding in a web environment. Instead of rendering the player's health through our PixiJS renderer and scene, we're going to display the health bar using HTML and CSS. And, since we have a way to tie the internal damage behavior to the external environment (through `onHurt`), this shouldn't be too difficult.

Let's create the HTML elements required:

```
<div class="camera"></div>
<div class="hud">
  <div class="heart heart-1"></div>
  <div class="heart heart-2"></div>
  <div class="heart heart-3"></div>
</div>
<div class="focus-target">click to focus</div>
```

```
-------
```

```
.camera, .hud, .focus-target {
  width: 100%;
  height: 100%;
  position: absolute;
  top: 0;
```

```
  left: 0;
}

.hud {
  .heart {
    width: 32px;
    height: 28px;
    background-image: url("path/to/sprites/heart-red.png");
    position: absolute;
    top: 15px;
  }

  .heart-1 {
    left: 15px;
  }

  .heart-2 {
    left: 57px;
  }

  .heart-3 {
    left: 99px;
  }

  .heart-grey {
    background-image: url("path/to/sprites/heart-grey.png");
  }
}
```

This is from http://codepen.io/assertchris/pen/qaoyPo.

This mark-up adds three red hearts to the top left of the screen. We're going to turn each gray as the player gets hurt:

```
let hearts = 3

player.onHurt = function() {
  document.querySelector(".heart-" + hearts) ↩
    .className += " heart-grey"

  hearts--

  if (hearts < 1) {
    alert("game over!")
    game.removeObject(player)
    game.removeObject(crosshair)
  }
}
```

This is from http://codepen.io/assertchris/pen/qaoyPo.

This is a lot simpler than I expected at first. Each time the `onHurt` function is invoked, we fetch the element related to the number of hearts we have left, and turn it gray as shown in Figure 11-1 (thanks to that `.heart-grey` class we added earlier).

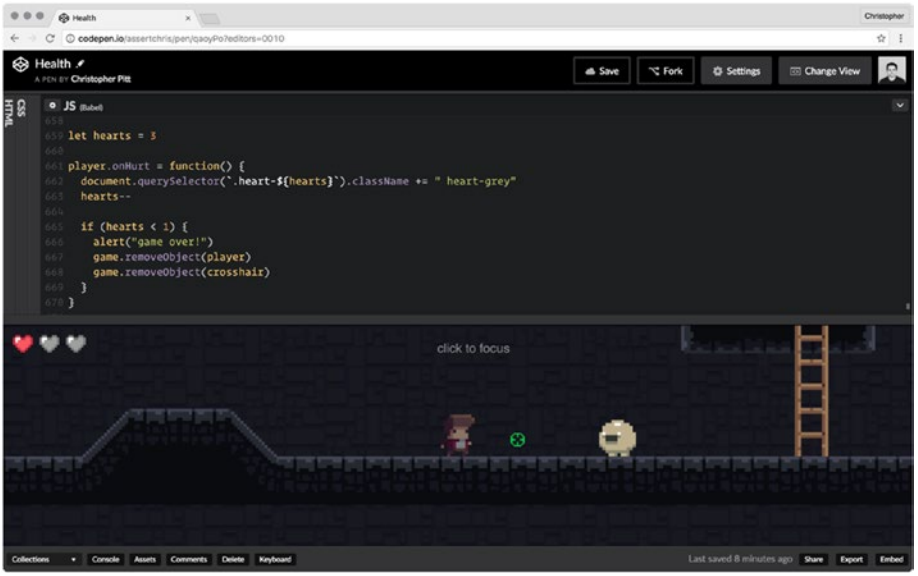

Figure 11-1. *Taking damage*

If the player has used up their last heart, we pop up an alert (though this would probably be better as a stylized "game over" message), and remove the player and crosshair from the game.

Summary

In this chapter, we added the first legitimate way for players to fail. Now that there's some danger, the experience should be more enjoyable.

We also create a way to tie internal events (like the player getting hurt) with external behavior. You don't have to kill the player off when they get hurt. You could try teleporting them back to the start of the level or reducing their abilities. The choice is yours.

CHAPTER 12

Animation

We're nearing the end of our brief but enjoyable journey. It's time to start adding some finishing touches. For instance, our sprites are too stoic. Let's animate them!

Animating the player

The player sprite is one of the first and most important parts of the entire game. We could spend ages staring at the little critter, and it would be very boring if we didn't add a bit of animation.

Fortunately, PixiJS provides a few tools, to make this process easier. Let's start using one of them:

```
const playerIdleLeftImages = [
  "path/to/sprites/player-idle-1.png",
  "path/to/sprites/player-idle-2.png",
  "path/to/sprites/player-idle-3.png",
  "path/to/sprites/player-idle-4.png",
]

const playerIdleLeftTextures =
  playerIdleLeftImages.map(function(image) {
    return PIXI.Texture.fromImage(image)
  })

const playerIdleLeftSprite =
  new PIXI.MovieClip(playerIdleLeftTextures)

playerIdleLeftSprite.play()
playerIdleLeftSprite.animationSpeed = 0.12

const player = new Player(
  playerIdleLeftSprite,
  new PIXI.Rectangle(
    Math.round(width / 2),
    Math.round(height / 2),
```


C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_12

```
    48,
    56,
  ),
)

player.idleLeftSprite = playerIdleLeftSprite
```

This is from http://codepen.io/assertchris/pen/ALyPGw.

We've replaced the `new PIXI.Sprite.fromImage` with `new PIXI.MovieClip`. We started by creating an array of images (which any good sprite pack should have) and creating new `PIXI.Texture` objects for each.

We need to call `play` for the animation to start, and adjusting the animation speed can't hurt. We'll see why it's a good idea to store a reference to the animation, shortly.

■ **Tip** No static image can do this animation justice. Check out the CodePen to see it move!

Swapping animations

We can animate more than just how the player looks standing still. For instance, we can add animation for walking, jumping, and even for getting hurt. Let's begin by adding the running animation. Together, the animation initialization code looks like this:

```
const playerIdleLeftImages = [
  "path/to/sprites/player-idle-1.png",
  "path/to/sprites/player-idle-2.png",
  "path/to/sprites/player-idle-3.png",
  "path/to/sprites/player-idle-4.png",
]

const playerIdleLeftTextures =
  playerIdleLeftImages.map(function(image) {
    return PIXI.Texture.fromImage(image)
  })

const playerIdleLeftSprite = new PIXI.MovieClip(playerIdleLeftTextures)

playerIdleLeftSprite.play()
playerIdleLeftSprite.animationSpeed = 0.12

const playerRunLeftImages = [
  "path/to/sprites/player-run-1.png",
  "path/to/sprites/player-run-2.png",
  "path/to/sprites/player-run-3.png",
  "path/to/sprites/player-run-4.png",
```

```
  "path/to/sprites/player-run-5.png",
  "path/to/sprites/player-run-6.png",
  "path/to/sprites/player-run-7.png",
  "path/to/sprites/player-run-8.png",
  "path/to/sprites/player-run-9.png",
  "path/to/sprites/player-run-10.png",
]

const playerRunLeftTextures =
  playerRunLeftImages.map(function(image) {
    return PIXI.Texture.fromImage(image)
  })

const playerRunLeftSprite = new PIXI.MovieClip(playerRunLeftTextures)

playerRunLeftSprite.play()
playerRunLeftSprite.animationSpeed = 0.2

const player = new Player(
  playerIdleLeftSprite,
  new PIXI.Rectangle(
    Math.round(width / 2),
    Math.round(height / 2),
    48,
    56,
  ),
)

player.idleLeftSprite = playerIdleLeftSprite
player.runLeftSprite = playerRunLeftSprite

game.addObject(player)
game.player = player
```

This is from http://codepen.io/assertchris/pen/ALyPGw.

Adding animation is really fun, but it can also be pretty tedious. We need to crop and export each frame of each action. Then we need to stitch them all together in many image arrays and animated sprites.

It's good to assign a reference to each animated sprite to `player`. That way, we can swap them inside `Player.animate`:

```
this.rectangle.x += this.velocityX

if (!this.isOnLadder && !this.isOnSlope) {
  this.rectangle.y += this.velocityY
}
```

```
if (this.isOnGround && Math.abs(this.velocityX) < 0.5) {
  state.game.stage.removeChild(this.sprite)
  state.game.stage.addChild(this.idleLeftSprite)
  this.sprite = this.idleLeftSprite
}

if (this.isOnGround && Math.abs(this.velocityX) > 0.5) {
  state.game.stage.removeChild(this.sprite)
  state.game.stage.addChild(this.runLeftSprite)
  this.sprite = this.runLeftSprite
}

this.sprite.x = this.rectangle.x
this.sprite.y = this.rectangle.y
```

This is from http://codepen.io/assertchris/pen/ALyPGw.

At the end of the `animate` method, we can check if the player is still moving (much). If not, we swap whatever sprite is presently visible with the idle animation.

If the player is still moving, we swap the visible sprite for the run animation. This logic only works in a single direction, but we can extrapolate the behavior to encompass a broad range of animations and directions.

Summary

In this chapter, we added a bit of animation (not the patrolling mob kind) to the game. We could spend hours watching the player's hair bob up and down.

There are many different kinds of animation we could add:

- Jumping and/or falling
- Getting hurt
- Landing (with a puff of dust)
- Climbing ladders
- Shooting a projectile (holding a weapon)

We've added just two, but there are tons more you could add. The key to these is finding (or designing) a good sprite pack; and stitching everything together nicely. If you can do that, you can make a beautifully animated game!

CHAPTER 13

Sounds

Some of my fondest gaming memories are of the music and sounds I listened to as I played my favorite games. Whether it is the music of Bastion and Fez or the sounds of Stardew, our ears help us fully appreciate the game.

Adding background music

There's nothing like a good soundtrack to immerse you in a game. The best games swap background music based on the player's mood and location. We're going to start with something simpler than that:

```
game.addEventListenerTo(window)
game.addRendererTo(document.querySelector(".camera"))
game.animate()

const music = new Audio("path/to/sounds/background.mp3")
music.loop = true
music.play()
```

Before you click this link, turn your volume down – it plays music.
This is from http://codepen.io/assertchris/pen/wzmQWb.

It's actually pretty easy to play sounds in modern browsers. A quick search for a looping MP3 and this new `Audio` object are just about all we need. We do, however, need to set the `loop` property to `true` if we want the background music track to loop.

Note There's no easy way to make `Audio` objects loop without the possibility of having a gap in between. You may want to consider an alternative solution, like SoundManager2 (https://github.com/scottschiller/SoundManager2).

C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_13

Adding action and event sounds

Adding game sound effects (for events and action initiated by the player), requires that we jump into where the actions and events happen:

```
element.addEventListener("keydown", (event) => {
  this.state.keys[event.keyCode] = true

  if (event.keyCode === 32) {
    new Audio("path/to/sounds/jump.wav").play()
  }
})
```

This is from http://codepen.io/assertchris/pen/wzmQWb.

The first time we try to jump, the sound effect seems to take a moment to load before playing. We could solve this by preloading all sounds before the player can start to move.

In fact, it's generally good practice to preload all game assets (like fonts, images, and sounds) before the game begins. In the case of sounds, we only need to add the following code, before the game starts to preload it:

```
new Audio("path/to/sounds/jump.wav").load()
```

This will start the loading process, and the sound file should be loaded by the time it is used (as long as the Internet connection is fast enough). There are other ways to ensure all sound files are loaded, but they complicate the process quite a bit. It's a topic for another time, I think…

Summary

In this chapter, we took a brief look at how to embed background music and action/event sounds into our game. Modern browsers provide good tools for this, but we can always fall back to libraries like SoundManager2 if we need to support older browsers or if we need greater control over playback.

Buying many sounds, music files, and sprites (or sprite packs) can be expensive. You may want to involve graphic and sound artists in the process of making a game instead.

CHAPTER 14

Gamepads

We're almost done now. Before we part ways, I thought it would be fun to experiment with gamepads. They're only supported via JavaScript in a few browsers, but they sure are fun to use!

Handling events

Gamepad events work slightly differently than the keyboard and mouse events we've seen so far. Due to their experimental support, we need to capture them in a certain way, inside `Game.animate`:

```
constructor(w, h) {
  this.w = w
  this.h = h

  this.state = {
    "game": this,
    "keys": {},
    "clicks": {},
    "mouse": {},
    "buttons": {},
    "objects": [],
    "player": null,
    "crosshair": null,
  }

  this.animate = this.animate.bind(this)
}

// ...later

animate() {
  requestAnimationFrame(this.animate)

  this.state.renderer = this.renderer
```


C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9_14

```
  this.state.stage = this.stage
  this.state.player = this.player
  this.state.crosshair = this.crosshair

  let gamepads = []

  if (navigator.getGamepads) {
    gamepads = navigator.getGamepads()
  }
  if (navigator.webkitGetGamepads) {
    gamepads = navigator.webkitGetGamepads
  }

  if (gamepads) {
    const gamepad = gamepads[0]

    gamepad.buttons.forEach((button, i) => {
      this.state.buttons[i] = button.pressed
    })
  }

  // ...remaining animation code
}
```

This is from http://codepen.io/assertchris/pen/WGzYgA.

We need to try a few different approaches before we find the gamepad list supported by the browser we're using. I'm using a modern version of Chrome, which supports the JavaScript Gamepad API.

We capture the pressed state of each button and store it in the `state.buttons` object (that we initialized in the `constructor`). Considering how much complexity we've added to `Player.animate`, I think it's time we refactored a bit of it:

```
animate(state) {
  const leftKey = state.keys[37] || state.keys[65]
  const leftButton = state.buttons && state.buttons[13]
  const rightKey = state.keys[39] || state.keys[68]
  const rightButton = state.buttons && state.buttons[14]
  const upKey = state.keys[38] || state.keys[87]
  const upButton = state.buttons && state.buttons[11]
  const jumpKey = state.keys[32]
  const jumpButton = state.buttons && state.buttons[0]

  if (leftKey || leftButton) {
    this.velocityX = Math.max(
      this.velocityX - this.accelerationX,
      this.maximumVelocityX * -1,
    )
  }
```

```
if (rightKey || rightButton) {
  this.velocityX = Math.min(
    this.velocityX + this.accelerationX,
    this.maximumVelocityX,
  )
}

this.velocityX *= this.frictionX

this.velocityY = Math.min(
  this.velocityY + this.accelerationY,
  this.maximumVelocityY,
)

state.objects.forEach((object) => {
  if (object === this) {
    return
  }

  const me = this.rectangle
  const you = object.rectangle
  const collides = object.collides

  if (me.x < you.x + you.width &&
      me.x + me.width > you.x &&
      me.y < you.y + you.height &&
      me.y + me.height > you.y) {

    if (object.constructor.name === "Ladder") {
      if (upKey || upButton) {
        this.rectangle.y -= this.climbingSpeed
        this.isOnLadder = true
        this.isOnGround = false
        this.velocityY = 0
        this.velocityX = 0
      }

      if (me.y <= you.x - me.height) {
        this.isOnLadder = false
      }

      return
    }

    // ...remaining collision detection code
  }
})
```

```
  if ((jumpKey || jumpButton) && this.isOnGround) {
      this.velocityY = this.jumpVelocity
      this.isOnGround = false
      this.isOnSlope = false
  }

  // ...remaining movement code
}
```

This is from http://codepen.io/assertchris/pen/WGzYgA.

Here we've defined a number of constants (to represent pressed keyboard keys and gamepad buttons). It's easy to assume a keyboard is present, but gamepads are less of a certainty.

That's why we combine a couple different keyboard keys into a single check, but each gamepad button check requires that we first make sure any gamepad buttons have been defined.

With this code, we've mapped the direction buttons (D-Pad) and the jump button (A on PS-compatible controller) to the corresponding player actions.

Triggers and joysticks

Triggers and joysticks are significantly more difficult (and prone to differences in gamepad design) than buttons. I'm using a PS-compatible Logitech gamepad, and the triggers are mapped to the `gamepad.axes` object, as are the joysticks.

The captured values range from `-1` through to `1`, on all 6 axes. At rest, the joysticks are not exactly `0`, which means we need to use some Epsilon value (https://en.wikipedia.org/wiki/Epsilon), or rounding, to determine whether the axis is at rest.

We'd also need to revise our trigonometric equations to account for the difference in input values/scale. I guess what I'm saying is we need to consider which gamepads we care to support, and the browsers our players will need in order to use them.

Summary

In this chapter, we dipped our toes into the ocean of gamepads. We're lucky enough to live in a time where browsers are gaining support for using gamepads with JavaScript, but there's still a lot of work to be done before it becomes easy or common.

If you're feeling brave, perhaps you would like to try your hand at mapping your gamepad's triggers and joysticks to game actions (like targeting and shooting projectiles).

I'd like to take a moment to thank you for reading this book. It's been a brief but thrilling project for me, and I hope you have learned and enjoyed it as much as I have (if not more).

As I said in the beginning: feel free to contact me with questions or suggestions for improvement. Given the dynamic nature of the examples, I will be able to fix bugs and add comments about how they can be improved.

My Twitter profile is https://twitter.com/assertchris.

On occasion, I'll also stream as I code. You can join in, ask questions (in real-time) and learn along with me.

My Twitch channel ishttps://www.twitch.tv/assertchris.

Index

C. Pitt, *Making Games*, DOI 10.1007/978-1-4842-2493-9

H, I

J, K

L

M

N, O

P, Q

R

S

T, U, V

W, X, Y, Z