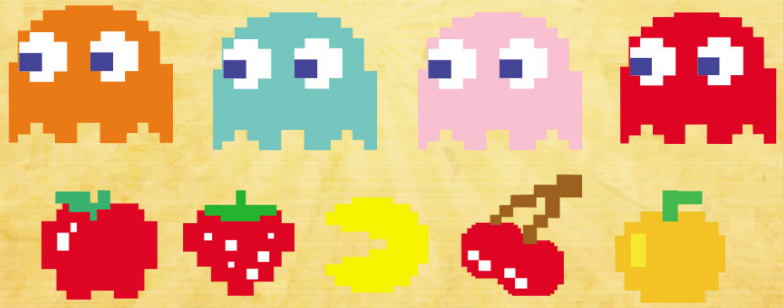
BUILD PACMAN

LEARN MODERN JAVASCRIPT, HTML5 CANVAS, AND A BIT OF EMBERJS



JEFFREY BILES

BUILD PACMAN

Learn Modern Javascript, HTML5 Canvas, and a bit of EmberJS

Jeffrey Biles

This book is for sale at http://leanpub.com/buildpacman

This version was published on 2016-05-02

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

In Media Res

[Content Warnings: Religion, Violence, Geopolitics, Creative Punctuation. Feel free to read just the instructional chapters if these things are gonna piss you off (or if you're at work)]

You're running through the jungle.

The why seems distant now, lost in the many dark mazes you've made. All you feel now is the whip of branches in your face, the crunch of twigs underfoot, and the chop chop chop of an approaching helicopter.

The helicopter suddenly comes into view. It's close. And there's a guy sitting there with a machine gun. "I've finally got you!" he yells. He starts firing, and trees splinter and fall under the spray of bullets.

One of those trees falls on you. Trapped. You're trapped. The helicopter is coming closer. He wants to see your face as you die.

But then a rocket bursts onto the helicopter, and it's in flames. Your pursuer leaps away from what is soon to be flaming rubble. He lands close to you, so close that you can hear the sickening crunch of his bones snapping. His power pellet had run out.

He still turns towards you, using his arms to drag his shattered body closer. His face strains with effort, pain, hate, but in it you see something familiar.

He collapses, out of breath, and it becomes clearer.

My god, it's him.

It's him.

He looks at you and sees the same thing. The rage drains from his face. Now… now it's just pain.

Just memory.

"I'll never forget," he says. "I'll never forget the caves."

And suddenly you're back there with him. Those dark caves, those blackened rooms, with only the pellets and the chase.

The constant fear.

They're abstractions now, but back then they were very real.

"I don't know why you turned," he says. "I guess I'll never know."

You start to explain, start to make excuses, but the words don't come. An untrained man would be crying, but you are not untrained. Instead, you only say: "You will be blessed in heaven for what you have done."

“Is it too late for you?” he asks. “Can I see you there?”

A bullet and he’s gone. The tree lifts off of you, and a hand extends, pulling you up.

You should feel relieved.

You’re safe.

Your fellow Ghosts have arrived.

1: Drawing A circle

This chapter is much like the previous one- weird, filled with characters you don't understand, and eventually coming back around to pac-man. The only difference is that in this chapter, we'll be doing lots of code.

It's a version of In Media Res- a storytelling technique used to get people into the action quickly. You open in the middle of the story, during a really exciting scene, and then you later go back and build up the characters and relationships that made that scene possible.

We're going to use a similar technique for this chapter- we're going to jump right into installing a bunch of tools and using some concepts you may not understand just yet. The upshot is that it's going to let us start drawing circles (little prototype pac-men) on our canvas really quickly. From there we can take it slow.

> A note about tools:
>
> We are going to be using lots of tools in this book- Javascript, Babel.js (a tool that lets us use the latest versions of Javascript before they're in all the browsers), HTML5 Canvas, Ember.js, ember-cli, and more. Originally the book was going to be a book for pros focusing on just ES2015 and HTML5 Canvas, but I discovered that using more of these tools actually makes our job easier.
>
> The goal of this book isn't to be a conclusive guide in any of these technologies- it's to give you enough knowledge in the tool to get started, and then use that tool to accomplish our goal (build a bad-ass game of pac-man). However, at the end of the book you'll be familiar with several really useful concepts- and if you decide to continue learning, I'll provide several resources for each.

Installation

The first step is to install Node. Go to https://nodejs.org/ and hit the big green "install" button. Do whatever your operating system makes you do to install things, and then you have node and npm on your system. Go to your command line, and then type in `node -v`. Then type in `npm -v`. If the commands worked and they gave you numbers, hurray! If not, please google for help. This isn't a book about node and npm… they're just tools we need to get started.

Next, we're going to install ember-cli. Type `npm install -g ember-cli` into the command line. And… that's it. Type `ember -v` into your command line, and it should give you the ember-cli version (>= 2.3.0) and the node and npm versions.

Creating your app

```
ember new pac-man
```

That command will create for you an ember app called pac-man. It's going to spend a short amount of time creating a directory structure and some config files for you, and then a bit longer installing npm and bower packages.

```
cd pac-man
```

```
ember server
```

These commands will move you into your app's folder and then start the server. Go to localhost:4200 and you should see "Welcome to Ember.js". If you do, then congratulations! You've created your first ember app.

Displaying a box

```
ember generate component pac-man
```

Type that into the command line and you will get a new ember component called pac-man. ember-cli will generate the javascript file, template file, and test for a component.

In app/templates/application.hbs, delete everything, replace with the component `{{pac-man}}`.

This tells ember to display the component. From now on, everything will happen inside the component.

```
1  <!-- in app/templates/components/pac-man.hbs -->
2  <canvas id="myCanvas" width="800" height="600"></canvas>
```

This creates an html5 canvas that we can draw on.

```
1  /*in app/styles/app.css*/
2  #myCanvas {
3    background-color: #EDC9AF;
4  }
```

This colors the canvas so we can see it. It's a desert-sand light brown.

If everything has gone correctly, at localhost:4200 you should see a rectangle the color of desert sand.

Drawing a circle

```
// in app/components/pac-man.js
import Ember from 'ember';

export default Ember.Component.extend({
  didInsertElement: function() {
    this.drawCircle();
  },

  drawCircle: function() {
    let canvas = document.getElementById("myCanvas");
    let ctx = canvas.getContext("2d");
    let x = 50;
    let y = 100;
    let radius = 20;

    ctx.fillStyle = '#000';
    ctx.beginPath();
    ctx.arc(x, y, radius, 0, Math.PI * 2, false);
    ctx.closePath();
    ctx.fill();
  },
});
```

`didInsertElement` runs whenever the component (the element) is loaded and put on the screen (more on this in the next chapter). In this case, we're just calling the drawCircle function, which is defined right below.

> Note for advanced Ember devs (everyone else ignore): using `didInsertElement` in this case will give a deprecation warning, but for now it's the best option. Using `init` will give an error because we need the html to have already been rendered. Using `didRender` will plunge you into an infinite loop.

In the drawCircle function, we're grabbing the canvas, telling it that we'll be drawing in 2d, and then doing the incantations that tell canvas to draw a circle. You won't have to worry about these specific incantations- this book will almost always hide shape-drawing code behind a function that you can just copy and paste without losing much.

Setup complete

Thus concludes our in-media-res introduction. It's expected that you won't understand a lot of what just happened- it's just meant to drop you into the world as quickly as possible. Some of the concepts introduced here will be given a lot more screen time later. Some of them were just necessary for setup and will never be brought up again.

Now let the training begin.

> Having trouble? Send me a question via email at jeffrey@emberscreencasts.com. I'll send you a response, and create an FAQ with answers to the most common questions

> You can also check out the github repositories for each chapter's code[1] or the finished addon[2].

[1] https://github.com/jeffreybiles/chapter-by-chapter-game
[2] https://github.com/jeffreybiles/pac-man

Training

The desert stretches farther than you can see. Only the occasional dust eddy disrupts the monotony.

“So this is the world outside.”

Elder Matteo nods.

“Is it all like this?”

“Most of it. Still patches we haven’t gotten to yet.”

“So it was different before…”

“When I was a boy, we had lush jungles, quiet lakes… food… we didn’t need to mine pellets back then.”

“How long has it been since you’ve seen this… food?”

A light goes on in the elder’s eyes. “I saw a cherry last year. When visiting the neighboring bubble. It was beautiful…. but enough of an old man’s reminisces. If this bubble is to be fed, we need more PAC operators. Are you ready?”

You turn your imagination away from what a cherry might be, and look at your PAC. Your Pellet Acquisition Capsule. You’ve been processing the pellets for years, but seeing a PAC up close for the first time is still an awe-inspiring experience. “Ready as I’ll ever be.”

You get in to the yellow sphere and let the wires envelop you. Matteo closes the hatch.

“Let’s get you acquainted with this contraption, alright?”

2: Getting up to speed

The last chapter threw you headlong into a large number of new technologies. I'm glad you made it out alright!

This is the part of the book where we introduce a bunch of characters, but we don't know much about them yet. Don't worry, you will eventually! Remember what you can, and come back to this chapter if you need to.

In this chapter we're going to zero in on the pac-man component, going over different parts of the code until it's (mostly) no longer mysterious.

Components

In `app/compents/pac-man.js`, you'll see the following code:

```
1  Ember.Component.extend({
2    ...
3  })
```

This means that PacMan is an Ember Component- it inherits all the properties of an Ember Component, while adding its own. An Ember Component is basically a container that is sealed from the outside world, except for if you poke a few holes into it. Most real-world programs will need to poke a couple holes, but we're not poking any holes right now, so all we need to care about is in `app/components/pac-man.js` and `app/templates/components/pac-man.hbs` (and `style/app.css`).

As someone writing an intro book, this is incredibly convenient. You may have heard about the Ember Router and how revolutionary and vital it is... well, we're going to write an entire game in Ember and never touch it. Same with Initializers, Controllers, Services, and innumerable other Ember features that are necessary in many apps but are also confusing to newcomers. We get to take advantage of the parts that are useful and leave the more confusing things for the second book.

> If you want to learn more about components, here is a series of videos about them[3].

Of course, components don't always consist of just a javascript file- they can have an associated handlebars (they can also consist of just a handlebars file, although that doesn't happen in this tutorial).

[3] https://www.emberscreencasts.com/tags/components

Handlebars

A handlebars file is kind of like an html file, but you can do some limited programming in it. You can input variables, use if statements, and more (but not much more). Each template is associated with an Ember class, and that is where it gets the variables it uses.

Later we'll use more of these features, but for right now all we're going to do is create the html5 Canvas (using just plain old html).

```
1  <!-- in app/templates/components/pac-man.hbs -->
2  <canvas id="myCanvas" width="800" height="600"></canvas>
```

Canvas

Canvas is well named- it's like a painter's canvas. You can paint on it... then you can paint on it again, partially painting over what you had previously drawn. And then, just like a real painter's canvas[4], you can create a bunch of robots that splatter paint over the canvas in precise intervals in order to create the illusion of motion.

This canvas defaults to pure white, but we've decided to tan it up a little using css.

Css

`app/styles/app.css` is where we're going to store the styling information. While css is a powerful tool, in this book we're basically just going to use it to turn things pretty colors, so there's no previous knowledge needed (and, likely, no knowledge to be gained). You can just copy and paste any css we present directly into `app/styles/app.css`.

Here's what you copy-pasted in the last chapter:

```
1  #myCanvas {
2    background-color: #EDC9AF;
3  }
```

It turns the canvas a nice tan color.

> CSS means 'Cascading Style Sheets', and is way more important in a regular web app than it is in pac-man. If you're interested in learning more, click here for some great introductory courses[5] (subscription required).

[4] If you're fabulously wealthy, a brilliant machinist, and don't care for the laws of physics.

[5] https://www.codeschool.com/paths/html-css

Canvas context and drawing

Of course, we don't just want that tan color... we want a circle!

That's where the following code comes into play

```
let canvas = document.getElementById("myCanvas");
let ctx = canvas.getContext("2d");

...

ctx.fillStyle = '#000';
ctx.beginPath();
ctx.arc(x, y, radius, 0, Math.PI * 2, false);
ctx.closePath();
ctx.fill();
```

The first line grabs the canvas, then the second line gets the 2d context, which we'll call `ctx` for short.

The last five lines are drawing the circle onto the 2d context of the canvas. You don't need to understand them. But you do need to know what `let` is.

Let

`let` is extremely well named. Take this example:

```
let chapterNumber = 2;
```

Read it out loud: 'Let chapterNumber equal two'. That's pretty much what it does. `chapterNumber` is now a variable equal to two. Another way to think of this is "assigning values to variables", where `2` is the value and `chapterNumber` is the variable.

If you've programmed in javascript before, you'll know about `var`. `let` is similar, but has slightly different scoping properties, which are generally less confusing. If you haven't programmed in javascript before, you can safely ignore that last sentence.

So right before we draw our circle we have a bunch of let statements, and then we use those variables in the drawing.

```
let x = 50;
let y = 100;
let radius = 20;
ctx.arc(x, y, radius, 0, Math.PI * 2, false);
```

is equivalent to

```
ctx.arc(50, 100, 20, 0, Math.PI * 2, false);
```

We name the variables because it's easier to read. We may also want to extract them later (say, next chapter).

> `let` is a feature of ES2015 which isn't supported in all browsers... why are we using it? Because we're using Babel, an ES2015+ transpiler. That means Babel takes your code that's written to the ES2015 (or ES2016, ES2017, etc.) and turns it into code that (almost) all browsers can read (sorry, IE7 users).
>
> "But wait," you ask, "how do I install Babel?"
>
> It's already installed. You started using it the moment you generated the project. This is (part of the) magic of ember-cli.

didInsertElement

`didInsertElement` is your way of telling the Ember component "hey, after you're done putting this component on the screen, I want you to do this."

"Putting this component on the screen" usually means "displaying whatever is in the handlebars file". In this case, the HTML5 Canvas.

Here's our full code:

```
didInsertElement: function() {
  this.drawCircle();
},
```

So what it's saying is "whenever you're done putting this component on the screen, draw a circle".

You may be wondering what's up with all the `function()` and the curly braces `{}` mean... well, those are parts of defining and using functions.

Functions

Functions are a great way to reuse code. We define it once, and then we never have to write that code again.

Here is us drawing a circle:

```
this.drawCircle();
```

Here is us drawing two circles:

```
this.drawCircle();
this.drawCircle();
```

It's easy, because we defined the `drawCircle` function. We'll get into the funny marks later.

Here's the basic format of defining a function:

```
functionName: function() {
  ...
}
```

> There's another slightly better way of defining functions that's new in ES2015. We're going to wait to introduce that technique, because introducing it now would make introducing hashes in the next chapter more confusing.

Here is us defining the drawCircle function:

```
drawCircle: function() {
  let canvas = document.getElementById("myCanvas");
  let ctx = canvas.getContext("2d");
  let x = 50;
  let y = 100;
  let radius = 20;

  ctx.fillStyle = '#000';
  ctx.beginPath();
  ctx.arc(x, y, radius, 0, Math.PI * 2, false);
  ctx.closePath();
  ctx.fill();
},
```

Everything in between { and } gets run when we call drawCircle.

We're defining the drawCircle function on the component. One way to think of it is to say that the component now knows how to draw a circle. Another more fancy way to think of it is 'the function drawCircle is assigned to the component scope'.

> Scope can be a scary word, but here's a basic way to think about it. If you're in your living room, and you say "I would like to sit on the couch", you don't have to specify which of the millions of couches you're sitting on. You're in the living room scope, so when you're trying to think of couches, the one in the living room comes to mind first. It's the same reason that if you talk to someone in the United States about "the civil war", they'll immediately think of the American Civil War, not the Spanish Civil War, the American Revolution, or Marvel Comic's Civil War (tm). That's because they're scoped to the United States[6].

Let's go back to where we were drawing circles and talk about those funny marks:

```
1  this.drawCircle();
```

`this`, in this case, means 'the component scope'. Hey, that's where we stored the drawCircle function. How lucky!

`this.drawCircle`, without the `()`, would return the code for the function. When we add the `()` it 'calls' the function, which means it runs the code in the function. And that code draws the circle.

> `didInsertElement` is a special type of function called a 'hook'. Don't worry about it for now, just noticed that we define it using the same patterns that we used to define `drawCircle`

import/export

One last thing is the import and export functionality. This is the ES2015 module system at work!

So what `import` says is "I'm gonna need some code. Give me the code."

```
1  import Ember from 'ember';
```

This says "Go to the place (library) known as 'ember', take the default thing there, and call it 'Ember'".

Then we have the following line:

```
1  export default Ember.Component.extend({
2    ...
3  })
```

This says "Whenever someone asks for something from this file (and doesn't specify any particular thing), give them the Component we're defining."

If we wanted to get PacMan in another file, we'd need to say:

[6](Okay, maybe some nerds here are scoped to Marvel Comics. I won't judge.)

```
1 import PacMan from 'pac-man/components/pac-man';
```

So we go into the pac-man library (our project), go into the components folder, and get the component named pac-man.

There's other ways to use imports and exports, but we won't need them in this book.

> Click here for more about ES2015 modules[7]

Feeling good yet?

We've learned a lot of new concepts in this chapter- Components, Handlebars, Functions, Let, Canvas, CSS, and ES2015 Modules. The rest of the book will build on these concepts (while introducing new ones, albeit at a slower pace).

Now, back to our story.

[7] https://www.emberscreencasts.com/posts/62-es2015-modules-import-export

Control

The first day in the PAC was spent familiarizing yourself with the different screens. You could have done it just fine at a desk, but you suspect that part of the training is getting used to being in the PAC.

The PAC was not as cramped inside as you would have guessed from looking at it, but it was definitely different than the engineering desks you were used to. Your torso and limbs, even your head, were trapped in their own web of wires. You could move them slowly in any direction, within limits, but anything sudden caused them to tauten. Matteo had just said, "You'll be glad of this once we start with movement. And collisions."

For now it was just disorienting. Thankfully the screen stayed in front of your face, giving something of a familiarity to the proceedings. You've spent most of your working life in front of a screen.

So yesterday had been the screens, and today would be the buttons.

As you approach the training field, you see that Elder Matteo is talking with another person. Getting closer you start to recognize him- Terrance Mayhew. Terrance, the man whose discoveries raised PAC efficiency 14%. One of only five people with significant achievements in both engineering and the PAC corps (although he had started in PAC and shifted to engineering, where he was now much more famous). The most honored of the ʿAhl Al-Kitab.

The talking ceases when you arrive.

"Salam," says Matteo, "Are you ready for an experiment?"

You nod, buzzing faintly with nervousness. The soreness in your limbs is forgotten in the excitement.

"Terrance and I have been working together on an enhancement to the training process, one that could decrease the disorientation and death that sometimes accompanies this stage of the training. We thought that you, with your background, would be the perfect test subject."

You nod again. Decreasing death sounds pretty good. And if it means you get to work on a project with Terrance…

You also begin to understand how big a deal Matteo is in the PAC world. Terrance could work with anyone, and it was Matteo. And you get to work with both of them. Maybe the summons wasn't such a curse after all.

"Where do I go?" you ask.

Matteo points to the PAC. You step in, standing still for the few seconds it takes for the wires to take hold.

"We're going to start with how you control the PAC."

3: Input

In this chapter, we'll be learning how to take keyboard input. We won't be moving yet, but we'll be showing the result in the log.

Steal This Code

Input has historically been difficult in javascript... but luckily, other people have done all the hard work for us! We just need a way to get their hard work into our project. Even more luckily, other people have designed easy ways for us to take their code! It's truly a wonderful world.

> We're not really "taking" the code, since it is freely given. These are more accurately called "code-sharing" tools, but for the duration of this book we won't be sharing... just taking. But I encourage you to start sharing your code as soon as you feel comfortable doing so!

Since you're using ember-cli, you already have access to several of these wonderful code-taking tools- and another tool that ties them together.

To take the particular piece of code we'll need, type `ember install ember-keyboard-shortcuts` into the command line. Now you have the library "ember-keyboard-shortcuts[8]" available to you!

> This will give you the npm package 'ember-keyboard-shortcuts' and the bower package 'mousetrap'. Npm and bower are both vital tools in everyday programming life, but we won't see them again in this book.

Be sure to restart your Ember server to get the keyboard shortcuts to load- stop the server in the command line (Ctl + C on OSX), then restart (`ember s`) after installing the addon.

Bringing it in

You can now access the 'ember-keyboard-shortcuts' codebase from within your project, but there's an intermediate step before you can use any of their code. You need to import it.

At the top of our pac-man.js component, under the Ember import, we'll import the KeyboardShortcuts mixin.

[8] https://github.com/Skalar/ember-keyboard-shortcuts

```
import Ember from 'ember';
import KeyboardShortcuts from 'ember-keyboard-shortcuts/mixins/component';
```

In this code, we're importing just one mixin- specifically, the mixin that is meant to be used in an Ember component.

> If you remember our discussion of modules from chapter 2, you may be able to guess which file in the addon we're importing this from. If you can't, no worries- understanding this isn't core to this book.

What is a mixin? Glad you asked.

Mixins

Remember when we said that `PacMan` would "inherit" a bunch of properties from `Ember.Component`? That basically said "Hi PacMan, I'm Ember.Component. You're going to be just like me, but don't worry- you can change things". Mixins say "Hey PacMan, here's a bundle of new related code you can use if you want."

How do you mix in a Mixin?

```
import KeyboardShortcuts from 'ember-keyboard-shortcuts/mixins/component';
export default Ember.Component.extend(KeyboardShortcuts, {
  ...
})
```

So after you call `extend`, but before the `{`, you put your Mixins (separated by commas). You can mix in as many Mixins as you want.

Inheritance Chain

Now PacMan has a mix of stuff on it from Ember.Component, KeyboardShortcuts, and the PacMan object itself. What happens if something is defined twice? In the general case, a definition on the object itself takes precedence over a definition on the Mixin, which takes precedence over a definition on the parent class. Applied to this specific case, PacMan beats KeyboardShortcuts, which beats Ember.Component. We've seen this already with `didInsertElement`, which was previously defined on Ember.Component but then overwritten on PacMan.

Defining keyboard shortcuts

After you've mixed in the KeyboardShortcuts mixin, you can put the following code into your PacMan class:

```
keyboardShortcuts: {
  up: function() { console.log('up');},
},
```

The effect of this is that when you hit the 'up' key, your browser says 'up' (in the console, which I'll show you how to access soon).

Let's go over each element of this, starting with hashes.

Hashes

Here's an example of a hash:

```
{
  firstName: "Jeffrey",
  lastName: "Biles",
  sideBusiness: "EmberScreencasts.com",
  firstTechnicalBook: "BUILD PACMAN"
}
```

The way to read this is: "first name is Jeffrey, last name is Biles, side business is emberscreencasts.com, and first technical book is BUILD PACMAN".

In fancy technical terms, you could talk about 'keys' and 'values', like "The firstName key has a value of Jeffrey", or "'firstName: "Jeffrey"' is a key-value pair". I bring up those terms partially to be fancy, but also so I can give important messages like "a hash is a set of key-value pairs" and "key-value pairs in a hash are separated by commas".

```
{
  key: value,
  anotherKey: 'anotherValue'
}
```

> Beginners: Read the last paragraph and code sample a couple times if you have to. Advanced folks: email me any easier explanations you've found.

The value in a key-value pair could be a function. That's what we'll use in our keyboardShortcuts hash.

```
keyboardShortcuts: {
  up: function() { console.log('up');},
  down: function() { console.log('down');},
  left: function() { console.log('left');},
  right: function() { console.log('right');},
},
```

Now that you know what a hash looks like, you may notice one other hash we've been using:

```
export default Ember.Component.extend(KeyboardShortcuts, {
  didInsertElement: function(){...},
  drawCircle: function(){...},
  keyboardShortcuts: {...},
})
```

So we can see that the value in each key-value pair can be a function or another hash. It can also be a simpler property, such as a number or a string.

Now that we understand hashes, let's look at the results of our work.

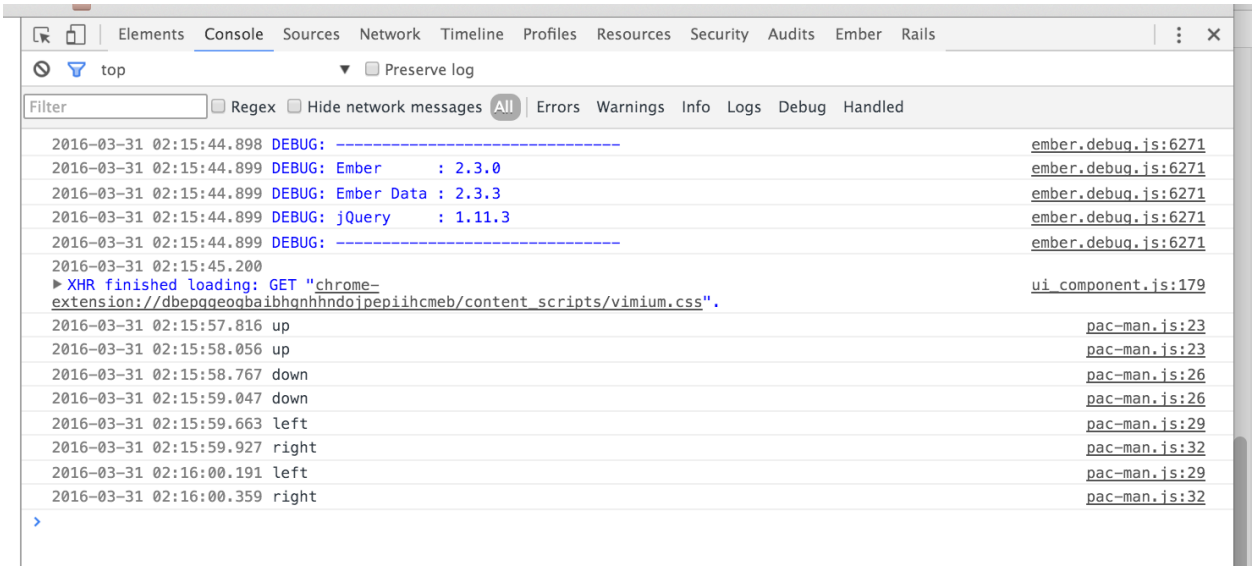
Logging

You'll notice that after each of the directions in keyboardShortcuts we've used a `console.log`. This says, "Hey browser, put this stuff in the console so the programmer can look at it."

But where in the browser does it put it?

1. Open up Chrome.
2. Right click anywhere on the screen.
3. Choose "Inspect Element" from the drop-down menu
4. Either click the 'Console' tab or press the escape key.

Now you're looking at the console. Hurrah! Click back into the game, then hit some direction keys. If you've set up your code like we've described, you'll be seeing the directions printed out.



You can put almost anything into `console.log`. You can put classes, functions, numbers, strings (what we currently have in there), hashes, and more. You can also put more than one thing in there- just separate them by commas.

Summary

In this chapter, we've given our code some... direction. We learned about how to import external libraries, how to use mixins, how to use hashes, and how to log stuff to the console.

The next chapter will use our button pressing to actually change stuff in the game world.

Displacement

“You ready to try this live?” Matteo says. The radio crackles slightly- the PAC’s shell interferes much more than you would expect.

“Yea,” you say. “If I wanted to print stuff on screen I would’ve stayed in engineering.”

Conveniently left unsaid is that you didn’t have much of a choice. After that last paper, you had been “cordially” “invited” to join the PAC corps. You don’t turn down an invitation from the elders council, no matter how bizarre or out of place it seems.

You can’t see anything outside the PAC except on your map, but you can hear Matteo and Terrance talking distantly on the radio.

“One hundred in a row with no trouble. We’re fine,” said Matteo.

“Slight jolting on subject five.”

“Well, we’ve got to get him trained, and even with ‘slight jolting’, this is far safer than using production PACs.”

“Not the old production PACs.”

Matteo grunted. “You know why those had to be abandoned.”

There’s silence for a while, and then the PAC around you starts to shake. It quickly settles down to a low buzz.

Matteo’s voice comes in over the radio- louder than when you were eavesdropping. Much louder. You turn the volume back down. “Alright, you should be good to go. Careful not to press too many buttons at once. This thing is… powerful.”

You cautiously press the “up” button and, after a quick shake, you see that your place on the map has changed. “Did I just go north?”

“See,” says Terrance, “He couldn’t even tell it was happening.”

“Take it around for a spin, see how it feels”, says Matteo. “Try not to go outside screen… the computers won’t know where you are, and the training PAC will shut down. We’ll have to tow you back.”

You press the buttons- maybe a bit too fast at first, as just a few clicks take you almost to the edge of the radar. Better to go the other way for a while.

It’s an amazing piece of tech, but it does have some strange limitations. You decide to ask. “So, do I just get four directions? What if I want to go a different distance?”

“That is not a limitation down in the caves. There, it is almost a blessing.”

You had seen diagrams of the caves before, but assumed that the right angles and even spacing were a simplification. Maybe they weren't. But it would be bizarre, a world shaped just so, a grid-world with no curves or variation.

You notice that the machine no longer shakes when you press the button, and realize that you've gone off the perimeter.

The elders have noticed it to. Matteo is cursing.

"See? Take away the rough feedback and they get careless."

"I promise you," says Terrance, "The benefits will outweigh the drawbacks."

"They better."

"How many days until he's ready?"

"To reach 238? A week, if the other tech you've come up with works as expected."

"And how long till *they* get here?"

"You know that's classified."

"You know that won't matter if we don't succeed."

A long silence.

"They're sweeping through India right now. It all depends on how long our boys can keep them busy."

A longer silence.

Matteo's voice comes in again, loud. This time he means to be heard. You scramble to turn down the volume.

"We're almost there. We'll pull you back. Oh, and we're going to be speeding up your training. We're going underground."

4: Movement

In this chapter we're going to make our PAC move. For our hero it was just pressing a few buttons, but as programmers we know that making things that simple can be hard work.

Properties

Remember our drawCircle function?

```
drawCircle: function() {
  let canvas = document.getElementById("myCanvas");
  let ctx = canvas.getContext("2d");
  let x = 50;
  let y = 100;
  let radius = 20;

  ctx.fillStyle = '#000';
  ctx.beginPath();
  ctx.arc(x, y, radius, 0, Math.PI * 2, false);
  ctx.closePath();
  ctx.fill();
}
```

It's a pretty great function, if I do say so myself. It's really great at drawing one black circle at [50, 100] with a radius of 20. However, we want to do more than just draw one circle. Let's take some steps to make ourselves a multi-circle species.

First, we're going to need to separate out where we store the values of `x`, `y`, and `radius`. Instead of defining them within the `drawCircle` function, we're going to be storing them on our `PacMan` component. That way we can access (and change) them from elsewhere. (We're also going to be storing `squareSize` instead of `radius`, for our near-future convenience)

```
//...
export default Ember.Component.extend(KeyboardShortcuts, {
  x: 50,
  y: 100,
  squareSize: 40,
  drawCircle: function() {
    let canvas = document.getElementById("myCanvas");
    let ctx = canvas.getContext("2d");
    let x = this.get('x');
    let y = this.get('y');
    let radius = this.get('squareSize')/2;

    ctx.fillStyle = '#000';
    ctx.beginPath();
    ctx.arc(x, y, radius, 0, Math.PI * 2, false);
    ctx.closePath();
    ctx.fill();
  }
  //...
}
```

So here we're storing our coordinates and squareSize on the (Pacman Component) hash, in the same way that we're storing the drawCircle function on the hash. Within the drawCircle function, we're accessing the x, y, and squareSize properties using the 'get' function.

So `this.get('x')` means "In *this* component there is something called x, and please *get* its value". So the program dutifully goes into this component instance and finds the value associated with the key 'x'.

Updating Properties

We just learned how to get properties on the hash, and this is progress. However, to move the PAC we're going to have to learn how to change those properties.

The most general way to do this is with the `set` method.

```
this.set('x', 150)
```

That piece of code will set the value of `x` on `this` (the component) to 150.

Here's a line of code that will not only set the value of x, but change it each time the line is called:

```
this.set('x', this.get('x') + this.get('squareSize'))
```

This grabs the value of x then changes it by the size of a square (the diameter of our PAC).

We're going to be adding and subtracting from values quite a bit, so we're going to use some of Ember's `incrementProperty` and `decrementProperty`.

> These are sometimes called "convenience methods". Convenience methods are functions that don't do anything really new and exciting (often replacing just one or two lines of code with one shorter line of code), but are common enough idioms that it makes sense to provide a shorthand for performing them.

So the previous line would be

```
this.incrementProperty('x', this.get('squareSize'))
```

Even though it's almost as long as it was before, it's's much easier to understand at a glance.

Moving the PAC

So let's use this new knowledge to move our PAC. We'll update the code for `right` on our keyboardShortcuts hash:

```
keyboardShortcuts: {
  up: function() { console.log('up');},
  down: function() { console.log('down');},
  left: function() { console.log('left');},
  right: function() { this.incrementProperty('x', this.get('squareSize')); this.\
drawCircle();},
},
```

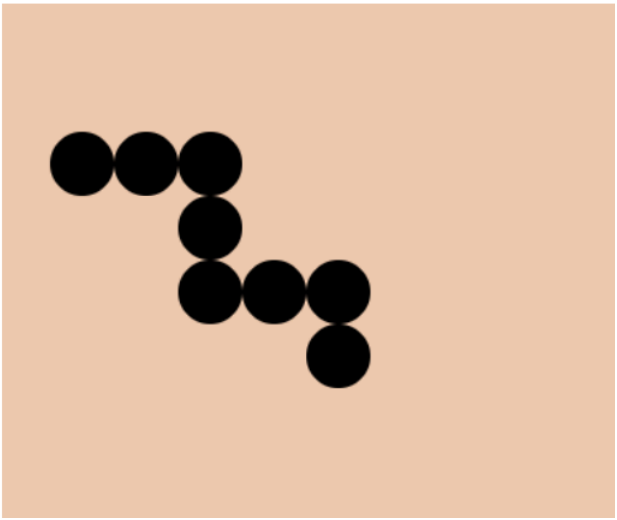
So when we press the `right` arrow key the x-value will increase by the size of a square (the diameter of the PAC). We'll also redraw the circle by calling `drawCircle`. Without calling `drawCircle` we wouldn't be able to see the changes. This has the visual effect of moving the circle one "square" to the right of where it was before.

> A quick syntax note: the semicolon is usually optional, but it's required when separating two different instructions that are on the same line.

Let's do that for all the other directions.

```
keyboardShortcuts: {
  up: function() { this.incrementProperty('y', -1 * this.get('squareSize')); thi\
s.drawCircle()},
  down: function()  { this.incrementProperty('y', this.get('squareSize')); this.\
drawCircle()},
  left: function() { this.incrementProperty('x', -1 * this.get('squareSize')); t\
his.drawCircle()},
  right: function() { this.incrementProperty('x', this.get('squareSize')); this.\
drawCircle()},
},
```

Try this out and you'll be able to make the PAC move around the screen. However, you'll see that while the new circle gets drawn, the old circles stick around. Let's take care of that.



Clearing the screen

We'll create a `clearScreen` function:

```
clearScreen: function(){
  let canvas = document.getElementById("myCanvas");
  let ctx = canvas.getContext("2d");
  let screenWidth = 800;
  let screenHeight = 600;

  ctx.clearRect(0, 0, screenWidth, screenHeight)
}
```

Like in our `drawCircle` function, we'll grab the canvas and then get the 2d context. Then we'll take an action on the context- in this case, clearing a rectangle the size of the screen.

Now we just have to call `clearScreen` every time we move the PAC:

```
keyboardShortcuts: {
  up: function() { this.incrementProperty('y', -1 * this.get('squareSize')); thi\
s.clearScreen(); this.drawCircle()},
  down: function()  { this.incrementProperty('y', this.get('squareSize')); this.\
clearScreen(); this.drawCircle()},
  left: function() { this.incrementProperty('x', -1 * this.get('squareSize')); t\
his.clearScreen(); this.drawCircle()},
  right: function() { this.incrementProperty('x', this.get('squareSize')); this.\
clearScreen(); this.drawCircle()},
},
```

When you run the program you'll be able to move the PAC, and the `clearScreen` function erases your old positions.

So it works, but the code is not only messy and repetitive, it also repeats itself.

Let's fix that before we move on.

Extracting methods

We're going to take our messy, repetitive code and extract a method from it. We'll do this by noticing what the patterns are and abstracting them away. We'll call our new method `movePacMan`.

```
movePacMan: function(direction, amount){
  this.incrementProperty(direction, amount);
  this.clearScreen();
  this.drawCircle();
},
```

`movePacMan` takes the three methods we call whenever we're moving our PAC and puts them into one method. We can then use that method when we're defining what to do after keystrokes.

```
keyboardShortcuts: {
  up: function() { this.movePacMan('y', -1 * this.get('squareSize'));},
  down: function()  { this.movePacMan('y', this.get('squareSize'));},
  left: function() { this.movePacMan('x', -1 * this.get('squareSize'));},
  right: function() { this.movePacMan('x', this.get('squareSize'));},
},
```

Not only is that shorter, but it's also much easier to understand. As an added bonus, if we want to change the process of moving the PAC then we only have to change it in one place (the `movePacMan` method) rather than four places.

Let's go ahead and make one more extraction.

A computed property

We've gotten the Canvas' context twice, and we'll probably be getting it even more in the future, so let's make it easier for us to do that.

```
ctx: Ember.computed(function(){
  let canvas = document.getElementById("myCanvas");
  let ctx = canvas.getContext("2d");
  return ctx;
}),
```

There are two new things here, and they work in concert.

The first is the `return` statement, which makes this function different than the ones we've created before. The previous functions have done something (made a change in the world), but doesn't change anything, and instead simply returns a value.

The second is the `Ember.computed` method which wraps our function. This turns it into 'computed property'. We'll learn more about these later, but what it means for us is that we can now treat `ctx` like a property instead of like a function- like `x` instead of like `drawCircle`.

So while a regular function is called like this: `this.drawCircle()`.

A computed property is retrieved like this: `this.get('ctx')`.

Let's use this computed property in clearScreen and drawCircle.

```
clearScreen: function(){
  let ctx = this.get('ctx');
  let screenWidth = 800;
  let screenHeight = 600;

  ctx.clearRect(0, 0, screenWidth, screenHeight)
}
```

```
drawCircle: function() {
  let ctx = this.get('ctx');
  let x = this.get('x');
  let y = this.get('y');
  let radius = this.get('squareSize')/2;

  ctx.fillStyle = '#000';
  ctx.beginPath();
  ctx.arc(x, y, radius, 0, Math.PI * 2, false);
  ctx.closePath();
  ctx.fill();
}
```

Summary

In this chapter we learned how to move our PAC.

To do that we had to learn about properties, which we can retrieve with `this.get` and change with `this.set` (or the convenience method `this.incrementProperty`).

We also created our own methods, such as `clearScreen` and `movePacMan`, which we used to separate out code into logical, reusable, easily-named blocks.

Finally, we got our first taste of computed properties- a very useful concept that we'll be expanding on in future chapters.

In the next chapter, we'll use grids and arrays to start preparing our PAC for the walled-off underground world it's meant to inhabit.

Barriers

Your first night in the barracks was uneventful, but you could feel the whispers.

In the mess hall it had been tables filled with grizzled veterans. You were down about eighty pounds from the average- most of the extra mass was muscle, but there was some extra padding and a layer of scar tissue thrown in for good measure. It was clear that you didn't fit in.

Inductions to the PAC-men happened three times a year, and were preceded by months of physical conditioning. The bonds built there were the core of the PAC-man experience. But you- you were brought in on an off-month, clearly with little preparation.

Prayers had been lonely- doing the Dhuhr and Asr observances without your team beside you was the most you had missed them yet- but the dining hall was worse.

Sitting alone, trying not to listen to the others' conversation above the din. Trying not to complete the snippets of conversation you hear, trying not to make them about you. Trying to justify the laughter as just a good joke that someone told.

Eventually someone comes and sets his tray in front of you. "Salam?"

You look at him, squint. Something about that face. You mentally take off a layer of muscle, take off two years of heavy wear and tear.

"Jerome?"

He smiles. "And they said my old friends wouldn't recognize me. So, what the hell are you doing here? I thought you were the golden child of pellet processing."

"Still am. Don't know exactly why they want me, but the elders didn't really ask my opinion about it."

"Well, as long as they don't kill the golden child who lays the golden research."

"Have you been continuing yours?"

"I moved into crash tests. To make the PACs easier to handle, less… brutal. I see by your lack of black eyes that my work has paid off."

"Brutal? So far mine has been silky smooth."

"You're in a training PAC. They're gentle- even gentler than the old style, thanks to me- but they eat up energy. They used to all be like that, but as we went deeper and deeper into the earth it became harder and harder to make it work, economically. We needed something more efficient… and that meant sacrificing some quality of life for the drivers. Hence all these guys you see around you. Battered bloody beefy heroes who get us all the pellets we need."

"How long till I look like that?"

Jerome shrugs. “My newest prototype has a quarter the injury rate as the standard- and five percent the rate of critical injuries. So… longer than it used to. But honestly, they’re not ready for a scientist’s build.”

The rest of the dinner went on uneventfully, a simple exchange between friends- if anything here could be called simple- but you took that line as a challenge. Jerome had been almost as thin as you when he left engineering for the PAC-men. Maybe you weren’t ready for the full bone-rattling experience of PAC life, but you were going to do your damnedest to accomplish whatever mission the elders had for you.

In the morning Matteo is all business.

“Today you’re going to start training underground. It takes some getting used to. The living quarters we’ve carved out, rounded the edges in order to satisfy our psyche. But down in the pellet mines, it’s cold hard angles, ninety degrees, equal spacing.”

“Got it, it’s a grid.”

“You say you get it, but no one really gets it until they’ve been operating down there for a while.”

“Then let’s operate.”

He laughs. “I like your style.”

You walk into a room filled with PACs. Most are training PACs like the one you’ll be using today, but a few look like they’ve been through the wringer a couple times. Matteo points to a training PAC and you get in. The inside isn’t quite as foreign as the first time- progress.

The computer rolls you into a perfectly rectangular room, and the training begins.

5: Enclosure

In Chapter 4 we got the PAC moving, but there was a problem: you could go off-screen. For Salam it meant that his training PAC shut down, but for our player it meant that they were lost with no idea of where they were.

In this chapter we're going to enclose the PAC within the screen- but first, we're going to have to make some changes with how we think about space in the game.

Alignment

The first change we'll make will align us more closely with the logic of the game.

Currently, we keep track of which pixels the PAC is placed at- the x and y pixels at which the circle is centered. Then we move it up, down, left, or right by the diameter of the circle. But that's not how we think of it. We think of it as a grid, and the PAC is moving between squares on the grid.

Thinking of the PAC as occupying a square on a grid, rather than just as a circle in space, takes a bit more effort up front while we build the abstraction, but will save us a lot of work very soon when we add things like walls and pellets and ghosts. Let's get started.

First we'll change how we record the starting position, from the pure pixel position:

```
1  x: 50,
2  y: 100,
```

to the spot on the grid:

```
1  x: 1,
2  y: 2,
```

> It's not an exact comparison, because 50 and 100 don't go evenly into 40. Grids gonna grid.

Then we'll draw the circle using the new grid system:

```
drawCircle: function() {
  let ctx = this.get('ctx');
  let x = this.get('x');
  let y = this.get('y');
  let squareSize = this.get('squareSize');

  let pixelX = (x+1/2) * squareSize;
  let pixelY = (y+1/2) * squareSize;

  ctx.fillStyle = '#000';
  ctx.beginPath();
  ctx.arc(pixelX, pixelY, squareSize/2, 0, Math.PI * 2, false);
  ctx.closePath();
  ctx.fill();
},
```

Notice how we're calculating `pixelX` and `pixelY` from the `x` and `y` grid positions. We're adding 1/2 to each, because `ctx.arc` wants the center of the circle, and simply multiplying by `squareSize` would give the top-left position on that grid square.

> Don't understand the above paragraph? Try doing without the 1/2 and seeing for yourself! The great thing about game development is that you can make changes to the code and then immediately see the effects for yourself in a very visual way.

Next, we'll take another look at the places where we're using `x` and `y`. For example, in keyboardShortcuts we're changing them by 40 each time we click a key.

```
// One keyboardShortcut under the old way
up: function() { this.movePacMan('y', -1 * this.get('squareSize'));},
```

With our new system, changing them by 40 is no longer a great idea. This is what the new code will look like:

```
keyboardShortcuts: {
  up: function() { this.movePacMan('y', -1);},
  down: function() { this.movePacMan('y', 1);},
  left: function() { this.movePacMan('x', -1);},
  right: function() { this.movePacMan('x', 1);},
},
```

The intent of this is much more clear than the old way, where we had to look past myriad `this.get('squareSize')` lookups in order to get to the essence of it.

In the last part of our transition to the grid, we'll tackle the `screenHeight` and `screenWidth` variables in `clearScreen`.

```
screenWidth: 20,
screenHeight: 15,
clearScreen: function(){
  let ctx = this.get('ctx');
  let screenPixelWidth = this.get('screenWidth') * this.get('squareSize');
  let screenPixelHeight = this.get('screenHeight') * this.get('squareSize');

  ctx.clearRect(0, 0, screenPixelWidth, screenPixelHeight)
},
```

Here we have the `screenWidth` and `screenHeight` variables scoped as properties on the component (so we can use them elsewhere later), and then we're using them to calculate `screenPixelWidth` and `screenPixelHeight` (800 and 600, respectively).

While nothing has changed in how the game works right now, this change does guarantee that as long as `screenWidth` and `screenHeight` are integers, the screen area will fit the grid.

Thus far this chapter we've refactored our code to fit a new mental model, but we haven't learned any new tricks yet. That won't last long.

If statements

Now that we've got our grid-like positioning system set up, our next goal is to make sure the PAC never leaves the screen.

So far when we press a button, it's a one-for-one action. Press right, go right... no matter what. But now we're going to be setting some conditions on that action.

To do that, we'll be using a conditional statement. Specifically, we'll be using the `if` statement. Here's the basic outline of how to use an `if`

```
// feeding in a boolean directly
if(booleanValue){
  console.log('booleanValue is true!')
}
// creating a boolean through an expression
if(myNumber == 4){
  console.log('my number is 4')
}
```

So the form is `if`, then a boolean expression between the parentheses, and then some instructions between the curly braces.

> A boolean value is `true` or `false`. A boolean expression is a piece of code that eventually simplifies down to `true` or `false`. It can just be a value, (a plain statement of true or false), a variable that means true or false, or it can be something like `myNumber == 4`, which will be true if `myNumber` is the integer `4`, and false if it's anything else.

Let's outline how we're going to make sure the PAC stays within the bounds. Basically, we're going to move the PAC, then we'll check to see if the PAC is outside the bounds, and if it is we'll reverse our previous move. This all happens before the circle is drawn, so to the user it looks like the PAC stayed in the same place.

Here's what it would look like.

```
movePacMan(direction, amount){
  this.incrementProperty(direction, amount);

  if(this.collidedWithBorder()) {
    this.decrementProperty(direction, amount)
  }

  this.clearScreen();
  this.drawCircle();
},
```

We'll be defining `collidedWithBorder` as a function on the pac-man component that returns a boolean value:

```
collidedWithBorder: function(){
  let x = this.get('x');
  let y = this.get('y');
  let screenHeight = this.get('screenHeight');
  let screenWidth = this.get('screenWidth');

  let pacOutOfBounds = x < 0 ||
                       y < 0 ||
                       x >= screenWidth ||
                       y >= screenHeight
  return pacOutOfBounds
}
```

The || means 'or', so reading this out loud it would say "pacOutOfBounds is true if x is less than zero or y is less than zero or x is greater than or equal to the screen width or y is greater than or equal to the screen height". Quite a mouthful, but remarkably easy to parse when you use symbols.

We then take the result of that boolean expression and set it as the return value of the function. Then that result is what determines whether the code in the if statement runs or not.

Fully functional collisions

Run that code and try to go out of bounds- you're always stopped at the border.

This is our first piece of code that only runs conditionally- it only runs if certain things about the world are true. This is a very useful tool that we'll be making liberal use of through the rest of the book.

Handlebars Binding

Our new functionality is done for this chapter, but there's a part of our code that we need to make more robust.

Go to the file `templates/components/pac-man.hbs`. You'll see this:

```
1  <canvas id="myCanvas" width="800" height="600"></canvas>
```

That's saying that we have a canvas element with a width of 800 and a height of 600. So far so good!- it matches up with the pixelWidth and pixelHeight in our file.

But what happens if we want to change the size of our level? Let's say that, instead of being 20 squares wide and 15 squares tall, it's 10 squares wide and 20 squares tall.

Try it- you'll soon see that the canvas (where all the graphics are displayed) has different dimensions than the allowed movement space. So while we can't go more than halfway across the canvas to the right, we could go down until we can't see the PAC anymore.

The solution is to have the width and height of the canvas change based on the width and height of our level.

First we'll compute those properties in the component's Javascript:

```
screenPixelWidth: Ember.computed(function(){
  return this.get('screenWidth') * this.get('squareSize');
}),
screenPixelHeight: Ember.computed(function() {
  return this.get('screenHeight') * this.get('squareSize');
}),
// Be sure to use these calculated values in `clearScreen`
clearScreen: function(){
  let ctx = this.get('ctx');
  ctx.clearRect(0, 0, this.get('screenPixelWidth'), this.get('screenPixelHeight'\
))
},
```

Nothing in the computations has changed this time, but we've scoped them to the component.

> For advanced practitioners: Using Ember.computed also caches the result, so it can result in a small speed boost (rather than having to do the calculation every time) Also for advanced practitioners: I left off using any compute keys in order to make explanations easier. It should be fine in this case since these computed properties will never need to be re-evaluated.

Then, we'll use those values in the component's Handlebars file:

```
<canvas id="myCanvas" width={{screenPixelWidth}} height={{screenPixelHeight}}></\
canvas>
```

In the code above, we're setting the width and the height of the canvas equal to the value of the `screenPixelWidth` and `screenPixelHeight` variables. In Handlebars (the templating language we're using), anything between `{{` and `}}` stops acting like html and starts acting like code.

This code is in the Handlebars templating language. The Handlebars templating language is different than Javascript, and much simpler. For this chapter, all we need to know is that we can display a variable.

> Where did the variable come from? Each template is paired with an Ember Object (sometimes auto-generated), and it has access to all the variables and properties that are on that object. This template is paired with the pac-man component we've been working on, so it has access to the screenPixelWidth and screenPixelHeight properties.

Go ahead and change the `screenWidth` and `screenHeight`- the canvas and the bounded area will both change in response. Pretty cool, yea?

Summary

In this chapter we aligned our code to a grid-like abstraction, set boundaries using an if statement and boolean logic, and then synced up our canvas display and our code using Handlebars.

In the next chapter we'll go from grid-like to grid, set up walls, and learn about arrays and loops.

The Maze Inside

You've spent the last hour moving your training PAC inside the bare rectangular room. At first you had stuck mostly to the middle, since collision with the outside wall prompted the training PAC to give you a shock on the side where you collided, but Matteo insisted you circle the perimeter.

"You gotta learn maneuvering. You need to live with the risk. You may not like the shock now, but when you get in a real PAC and run into a real wall, you'll goddamn miss it."

After an hour you are getting shocks less and less, although it still takes a lot of concentration to circle the perimeter without running into a wall.

Matteo pushes you on anyways.

"These training areas are the only places on earth with a completely clear grid. Damn near perfect, if getting bored but not hurt is your thing. Back when there were things worth getting on the surface we had to worry about ground cover density, curves, stuff like that. It was wild. Now underground it's just passageways and walls. But walls, they can be hard if you're not a PAC-man."

What did that mean? No time to think about it, as he continues talking. "You're gonna encounter walls, so we're gonna get you some walls."

And wall he delivers. He presses a few buttons and a pillar rises out of the floor, a perfect square pushed from floor to ceiling.

"Circle the pillar," he says.

You move the PAC toward the pillar, then start the series of movements that will make you move in the correct circuit. There are a couple of cruel shocks as you trying to figure out the best way to carry out the pattern.

"I don't know if I'm ready yet," you say.

Matteo harumphs. "What, you want to be perfect before you move on? Maybe run some equations?"

"I just think that if I spent more time…"

"Look, you've got the basic skill well enough, and this is classified, but you're on a tight training schedule. Not my choice to send you down there, but it's my job to make sure you don't die on the way to your mission."

"On the way to my mission?"

"Yea. The way back, not nearly as important, as long as someone has the results, but let's call that a stretch goal."

"So you're planning on me dying down there?"

“That’s up to you- I’d like you to survive, professional pride as a trainer and all that, but no, it’s not mission critical that you survive on the way up. It all depends on how hard you want to work.”

Is this a… pep talk? Military people are weird.

“Anyways, you’re gonna get some unpleasant shocks in training, but after a certain amount of time practicing the basics it’s more efficient to go for skills that build on those basics in interesting ways. If you feel comfortable, it’s safe to move on. For you, if you feel like you’re getting into a routine, anything even close to a routine, then the only safe option is to learn more. Not learning will get you killed.”

“Really?”

He shrugs. “Probably. Now get to training.”

You trace a circuit around the pillar. Then you do it again. And again.

6: Internal Barriers

In the last chapter we set our PAC on a grid-like structure and blocked it so it can't get out of the playing area. But pac-man is about so much more than just running around in a padded room- there's also some internal barriers that you can run into. This chapter is all about erecting those barriers.

Drawing a wall

Let's take a look at what it would take to draw one wall:

```
1  let squareSize = this.get('squareSize');
2  let ctx = this.get('ctx');
3  ctx.fillStyle = '#000';
4
5  let wall = {x: 1, y: 1}
6  ctx.fillRect(wall.x * squareSize,
7               wall.y * squareSize,
8               squareSize,
9               squareSize)
```

First we grab our `squareSize` and `context` and make sure the context is coloring with black (`#000`).

Then we create a wall- a hash containing an x value and a y value.

Then we combine all that to draw the wall at the correct place using `fillRect`. The arguments to `fillRect` are the x value of the left side, the y value of the top, the width, and the height. We fill those in, making sure to multiply by `squareSize` for the first two in order to convert grid values to pixel values.

Notice that we can access the values on a hash by using the appropriate key (the `x` or `y` key on the `wall`). This is the simplest way to access values on a hash (although it's not appropriate in every situation, as we'll see later).

Walls, arrayed before us

Of course, we don't want just one wall. We want LOTS of walls. And, since we won't just be drawing them, we'll want them stored on the component rather than created in the drawing function.

We'll start off simple and just create two walls.

```
walls: [
  {x: 1, y: 1},
  {x: 8, y: 5}
],
```

This property is just two wall hashes arranged in a list. The fancy term for a list like this is an `array`. We're declaring the array by putting a series of things in between `[` and `]`, with each item separated by a comma.

Here's a simple array: `let fruit = ['cherry', 'strawberry', 'orange'].`

Our wall array is slightly more complicated since we're creating an array of hashes instead of strings. We make it easier to view by giving each wall hash its own line instead of listing them all together on one line.

Now let's put that array of walls to use and draw them.

Drawing all the walls

We'll put our code in the drawWalls function:

```
drawWalls: function(){
  let squareSize = this.get('squareSize');
  let ctx = this.get('ctx');
  ctx.fillStyle = '#000';

  let walls = this.get('walls');
  walls.forEach(function(wall){
    ctx.fillRect(wall.x * squareSize,
                 wall.y * squareSize,
                 squareSize,
                 squareSize)
  })
},
```

This is very similar to the first time we drew a wall, but with some key differences. First, rather than define a singular wall within the function, we get the list of walls from the component that we created earlier. Then, we use the `forEach` method in order to draw all of those walls.

A bit more about how the (extremely well-named) `forEach` method works: You take an array, and then you call the `forEach` method on the array, and then you give that method a function. The function is what you're going to do to each item in the array.

For our purposes, that function will take one argument- the item in the array. In our code, that would be `wall`.

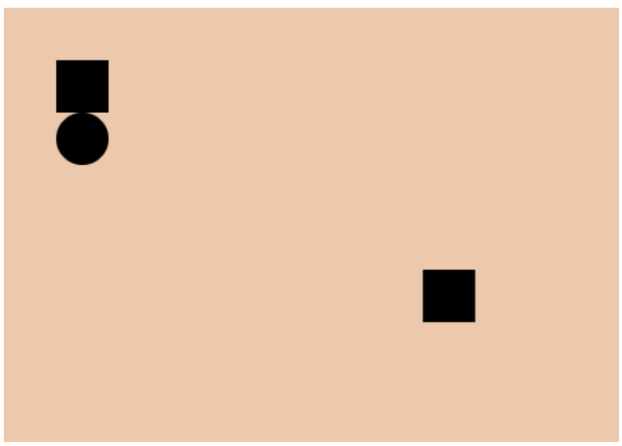
Then, you'll recognize the `fillRect` code from our previous example. We'll just be using `fillRect` to draw each of the walls in the array.

Call the Draw

Now that we've got the `drawWalls` method, we need to call it. We'll just add it right before drawCircle in both places where drawCircle is found.

```
didInsertElement() {
  this.drawWalls();
  this.drawCircle();
},

movePacMan(direction, amount){
  this.incrementProperty(direction, amount);

  if(this.collidedWithBorder()) {
    this.decrementProperty(direction, amount)
  }
  this.clearScreen();
  this.drawWalls();
  this.drawCircle();
},
```

This gets us the walls displaying on our grid. Now it's time to make them count.



Wall Collisions

In the movePacMan method, we've got this snippet of code:

```
if(this.collidedWithBorder()) {
  this.decrementProperty(direction, amount)
}
```

We also want to go back if we collide with a wall. Let's make that happen.

```
if(this.collidedWithBorder() || this.collidedWithWall()) {
  this.decrementProperty(direction, amount)
}
```

No, we don't have a `collidedWithWall` method yet... we're writing how we want to use it first, and then we're going to make the method.

> The 'call methods and then define them' style of programming can be very freeing. Instead of writing all the implementation details at the start, you can write a clear set of instructions and worry about the specifics later. Of course, like all programming styles, it is very helpful in some situations while being harmful in others.

The `collidedWithWall` method combines the boolean logic techniques we used in `collidedWithBorder` with a twist on the array handling we used in `drawWalls`.

```
collidedWithWall: function(){
  let x = this.get('x');
  let y = this.get('y');
  let walls = this.get('walls');

  return walls.any(function(wall){
    return x == wall.x &&
           y == wall.y
  })
}
```

The `any` method returns a boolean value. It goes through the array and runs a function on each item. If the return value from *any* of those is true, then the `any` method returns true.

The function used in this `any` method is checking whether the PAC is on top of a wall (The `&&` symbol means 'and').

So, if the PAC is on top of any walls, then `collidedWithWall` returns true. Run the code, try to run into a wall... and it works!

Summary

In this chapter we used the `fillRect` method on our context in order to draw a wall. Then we learned about arrays, and created an array of walls. We created the `drawWalls` method which looped over that array (using `forEach`) to draw each wall. We then created the `collidedWithWall` method which used the `any` method to check whether we were on top of any walls. Along the way we also learned about boolean logic and the `||` (or) and `&&` (and) logical operators.

I heard you liked the part about arrays, so in the next chapter we'll put some arrays in our arrays. We'll also be eating our first pellets. Exciting!

Clearing a Path

"How'd training go today?" asks Jerome.

You don't want to think about it, much less talk about it.

After Matteo's pep-talk about your future death, you had spent a few uneventful hours navigating circles (squares?) around a wall. After that you did the figure eight exercise. It was mostly fine, but even though you held it together and steadily improved your shock rate, you were getting tired and your nerves started to frazzle.

Then came the diagonal line. A series of walls where you were forced to go right, then down, then right, then down, all the way down the line. One wrong move, and it was time for a shock.

You made lots of wrong moves.

Eventually you collapsed on the control panel, ramming into the same wall over and over while the training PAC shocked you for each infraction. The last thing you remember is Matteo yelling, and then some people in white shirts pulling you from the PAC.

There's no way you're telling Jerome how training went today.

"Got your first infirmary visit?"

Is it that obvious? Jerome is being nice because he knew you before this, back when you were good at something, but you can only imagine what he's thinking. He knows how bad you suck. If he can see it, can the others see it? And they won't be so nice. They would just ignore you. Like they're doing now. And if it was obvious you sucked, Jerome would be nice and enjoy the company of an old friend while he was still around, which he wouldn't be for long. And that was what he was doing now.

Oh god. They all know you suck.

Maybe someone told them. Maybe the people in the white shirts carried you unconscious through the halls and people saw. Maybe everyone saw. At least, everyone seems to know, even if they didn't see it themselves. Why else would they be studiously ignoring her? Why else would Jerome be so friendly?

Jerome coughs. "The shocks aren't supposed to make you mute, just unconscious."

You snap back, and a few words catch. The shocks are supposed to make you unconscious. And he had said "first" infirmary visit, meaning people made multiple visits. It was normal to make multiple visit. Right?

"How'd you know I went to the infirmary?" you ask.

He points to your arm. "Shock burns. It happens when you collapse inside a training PAC."

"So… this happens a lot? I mean, is this supposed to happen?"

"We've built frustration and pain into the training, because when you're down in the pellet mines there will be plenty of frustration and pain. If you get everything right the first try, or if you practice until it's perfect, then all you'll learn in how to operate a PAC. But being a PAC-man is so much more than that. It's being rattled but staying in control. It's having a frustrating day, maybe even going to the infirmary, and then getting back up and trying again tomorrow."

"But I don't want to be frustrated. I feel like a failure right now. In the infirmary I watched videos of these PAC-men navigating the mines and they make it look so effortless. I don't think I'll ever be as good as these guys. And if the guys here ever find out how bad I am…"

"Look. We've all felt that way. It may be more intimidating for you since you're from a different background and you're not sure if you really belong here, but we all spent time confused, learning, banging around in the training rooms. You see all the scars on everyone around you? They didn't get those by being perfect. Even *after* training."

"But I'm scared that if I tell anyone here what happened today, they'll laugh."

"Yea, they will laugh. And then everyone who's not an asshole will share a story where they messed up even worse. And then you'll laugh."

"So you're saying I'm just as good as anyone here."

Jerome laughs. "Not yet, but I know you, I've seen your mind, I've seen you work. If you stick with it you'll be among the best. Now let me show you how to think like a PAC-man."

He pulls out a sheet of paper and starts drawing a grid…

7: The Grid

Seeing the grid

In the last chapter we created some walls that we could put in the middle of our playing field. That was pretty cool, but the way we specified them made it hard to visualize:

```
walls: [
  {x: 1, y: 1},
  {x: 8, y: 5}
]
```

Do you know what the playing field will look like? I can only imagine it if I think really hard. Add in ten more walls and you can forget about understanding your level.

We're going to replace that walls array with a grid:

```
// 0 is a blank space
// 1 is a wall
grid: [
  [0, 0, 0, 0, 0, 0, 0, 1],
  [0, 1, 0, 1, 0, 0, 0, 1],
  [0, 0, 1, 0, 0, 0, 0, 1],
  [0, 0, 0, 0, 0, 0, 0, 1],
  [0, 0, 0, 0, 0, 0, 0, 1],
  [1, 0, 0, 0, 0, 0, 0, 1],
],
```

As the comment says, '1' represents a wall and '0' represents a blank space. Now, even though we have ten walls, it's easy to understand our level.

The structure we're using to do this is an array of arrays.

> Here we're optimizing for the visual look of the grid, so that it's easily parsed into 2d space. Numbers are perfect for this.
>
> However, the downside is that '1' is a terrible representation for 'wall', and we have to use a comment to explain this. If we process that '1' later in the code, it will be without context and may trip you up.
>
> There are techniques for gaining both clarity in the grid representation and clarity elsewhere in the code, but they will not be covered in this book.

Arrays of arrays

Last chapter we started using arrays. Array is just a fancy term for 'list'. It's a bunch of items in between `[` and `]`, and separated by commas.

Here's an array of strings: `['cherry', 'strawberry', 'orange']`

Here's an array of numbers: `[0, 1, 0, 1, 0, 0, 0, 1]`

That array of numbers happens to represent a row (since we know what `0` and `1` mean in this context).

An array of arrays is just a list of other lists. In this case, a list of rows in our grid.

Displaying the grid

Previously, our `drawWalls` function looped through the list of walls (using `forEach`) and drew each one:

```
drawWalls: function(){
  let squareSize = this.get('squareSize');
  let ctx = this.get('ctx');
  ctx.fillStyle = '#000';

  let walls = this.get('walls');
  walls.forEach(function(wall){
    ctx.fillRect(wall.x * squareSize,
                 wall.y * squareSize,
                 squareSize,
                 squareSize)
  })
}
```

With the grid representation, the `drawWalls` function will loop through every cell with a nested loop, and then draw a wall if the value is '1' (don't worry, we'll explain every part of this code):

```
drawWalls: function(){
  let squareSize = this.get('squareSize');
  let ctx = this.get('ctx');
  ctx.fillStyle = '#000';

  let grid = this.get('grid');
  grid.forEach(function(row, rowIndex){
    row.forEach(function(cell, columnIndex){
      if(cell == 1){
        ctx.fillRect(columnIndex * squareSize,
                     rowIndex * squareSize,
                     squareSize,
                     squareSize)
      }
    })
  })
},
```

So first we see that though we're still using `forEach`, the function we give it now takes a second argument. That argument is the index of the array. The function could have taken that argument last chapter, but we didn't need it so we left it off. Now that we need it, we're putting it back on again. There is a third argument available, which is the entire array that the `forEach` is acting on.

> If you want to have the third argument, you need the first two. If you want the second argument, the first is required but the third is optional. If all you're using is the first argument, you do not need to specify any of the other arguments.

So we're taking each row, and for each row we're looking at each cell (each number in the row array). If the number is 1, then we're drawing a rectangle. Previously we knew where to draw the rectangle using the x and y value of our wall hash, but now we're using the indices of the cell on the grid.

One thing to note is that arrays in javascript are zero-based, which means that the index for the first item in the array is 0, and the index for the second item in the array is 1, and the index for the third item in the array is 2, and so on. This works out for us, since we're drawing rectangles from the upper left corner- if we want the first cell in the first array to be a wall, we'll want to specify [0,0] and not [40, 40] as the top left corner.

That's how we display the walls- but you'll notice that we're still drawing a much larger space for the level. Let's fix that.

> Notice that because we kept all of our changes with `drawWalls`, we didn't need to change anything in the two places where we called `drawWalls`. Three cheers for abstraction!

If you try to play right now, it won't work. That's because we haven't updated the collision detection. We'll fix that before the chapter is done!

Calculating the level width and height

In chapter 5 we set our the width and height of our canvas element like this:

```
<canvas id="myCanvas" width={{screenPixelWidth}} height={{screenPixelHeight}}></\
canvas>
```

And then we calculated the pixelWidth and pixelHeight like this:

```
pixelWidth: 20,
pixelHeight: 15,
screenPixelWidth: Ember.computed(function(){
  return this.get('screenWidth') * this.get('squareSize')
}),
screenPixelHeight: Ember.computed(function(){
  return this.get('screenHeight') * this.get('squareSize')
})
```

The problem here is that the width and height have changed. They're no longer simply declared-they're now based on the grid. So we're going to use the grid to calculate them. Let's start with the height.

```
screenHeight: Ember.computed(function(){
  return this.get('grid.length');
}),
```

This is setting the height to length of the grid array, which is the the number of rows that are in the grid. In the grid we created earlier in the chapter, that would be 6.

'length' is a property that is on every javascript array, and does exactly what you would expect.

Deep gets

Note that `this.get('grid.length')` is the same thing as `this.get('grid').length`. Ember's `get` method is able to chain properties by separating them with a '.' in the string.

> The combined version has the advantage, besides being easier to read, of not blowing up if grid is null or undefined.

We'll do something similar with the width:

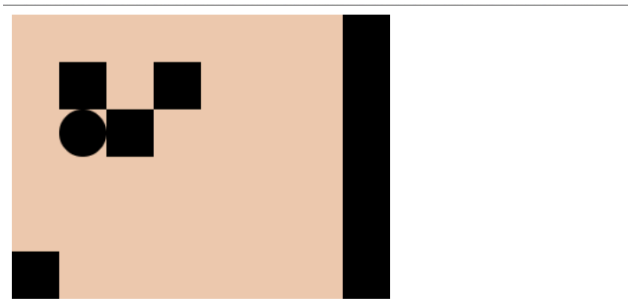
```
screenWidth: Ember.computed(function(){
  return this.get('grid.firstObject.length')
}),
```

This takes the grid, grabs the first row (with `firstObject`), and gets the length of that first row (in this example: 8).

'firstObject' is a property that is on every Ember array (but not every javascript array) that gets the first item in the array.

Note that `this.get('grid.firstObject.length')` is the same thing as `this.get('grid').get('firstObject').le`

> Why use a `get` for firstObject but not length? Ember properties must be retrieved with the `get` method, not accessed directly. Javascript properties may be accessed directly, or they may be retrieved with the `get` keyword. So the last expression could also be expressed as `this.get('grid').get('firstObject').get('length')`.



Now our canvas is fitted to our grid, because we've correctly calculated the width and height of the grid. To finish up this refactor, let's get our PAC moving again.

Moving through the space

The reason movement stopped working is because the `collidedWithWall` method still expects an array of walls. It doesn't get an array of walls anymore, so it stops the program from continuing. Now we're going to change it to work with our grid.

Remember, the `collidedWithWall` method checks to see if our PAC had hit any of the walls. Here's how we did it before:

```
1  collidedWithWall: function(){
2    let x = this.get('x');
3    let y = this.get('y');
4    let walls = this.get('walls');
5
6    return walls.any(function(wall){
7      return x == wall.x &&
8             y == wall.y
9      })
10   })
11 },
```

Here's how we're doing it now:

```
collidedWithWall: function(){
  let x = this.get('x');
  let y = this.get('y');
  let grid = this.get('grid');

  return grid[y][x] == 1
},
```

Before, we were looping through the list of walls and checking against each one. Now, because we're at a specific point in that grid, we can just check that grid.

We're checking that point in the grid through array accessors. They're a shortcut for finding an item at a specific point in an array. So here's how that would work with one array:

```
let fruits = ['cherry', 'strawberry', 'orange']
fruits[0] //'cherry'
fruits[2] //'orange'
```

Remember, the arrays are zero-indexed, so `[0]` gets the first item, and `[2]` gets the third item.

So let's apply that to a simplified two-dimensional grid:

```
let grid = [
  [0, 0, 0, 0, 0],
  [0, 1, 0, 1, 0],
  [0, 0, 1, 0, 0],
  [0, 0, 0, 0, 0]
]
grid[0] //[0, 0, 0, 0, 0]
grid[1] //[0, 1, 0, 1, 0]
grid[1][1] //1
grid[1][2] //0
```

So the first accessor gets the row (the y value), and the second access gets the column of that row (the x value).

Our new `collidedWithWall` function just finds the numeral that's at that coordinate and returns true if it's equal to 1 (if it's a wall).

Summary

So in this chapter we didn't build any new functionality in the game... but we completely changed how we think about working with levels in the game, and made it easier for us to understand them.

During the course of the refactor we learned about arrays of arrays, array accessors (through the following syntax: `[integer]`), the `length` and `firstObject` properties on arrays, and extra arguments for the function we feed `forEach`.

We also set the stage for adding in pellets- something that would have been extremely difficult to do without the grid system we created this chapter.

First Harvest

Jerome's explanation of the grid really helped- it's like you can see the whole room in your head now, like a map, and anticipate where the pillars will be.

You are now perfectly prepared for yesterday's training.

"Alright," Matteo says, "Are you ready for your first day in the mines?"

Your jaw drops. "A day in the mines? Already?"

Matteo laughs. "Not the big boy mines. The training mine. Come."

You walk to a different doorway than yesterday and take an elevator. A training PAC is rolling into the room as you get there, and he motions you to get in.

After you're in the PAC, you roll into what Matteo is calling the training mine. The walls are not quite as smooth -they've been chipped in places- and the pillars are made of solid earth and aren't able to go up and down like the ones in the training room.

It's also considerably darker, lit only by faint fluorescents in the ceiling and the soft glow of an unhusked pellet.

"These pellets, they're what gives us everything we enjoy in our modern life, including our food, our power, our…"

"I know," you interrupt. "I've spent my entire life coming up with new and more efficient ways of using the pellets."

"Ah. Good. Yes. That's why they brought you, isn't it."

"What can you tell me about that?"

He pauses, stopped short. You can almost see the information being stopped by security as he starts and stops speaking several times.

Another voice cuts in, that of Terrance Mayhew. "Salam," it says, "what do you know of the implications of your most recent paper?"

You stop and think… it was about a new form of radiation that came from especially dense pellets, and the interactions it had with organic materials and metal. Nothing special, merely theoretical musings on a rare form of matter.

"It… may provide us a more efficient form of energy?" It had been a side project, one you had thought about for years but only recently been given resources for, so nothing big had come of it yet. It was early enough and esoteric enough that your lab-mates hadn't even bothered to figure out what you were working on.

Terrance laughs. “Yes, that’s one way of saying it. You haven’t been given access to this, but back before the Caliphate, when Jihad was external, there was a weapon. An energy weapon, a very efficient energy weapon. It could level an entire town. After the Final Jihad was complete we destroyed all we could find, so that they would never be used again. We destroyed the plans that were used to create them. We even… well, after we were done, no one on earth knew how to build one of them. We truly thought we would have no other enemies. And for centuries, this was true.”

“But now the Ghosts have arrived.”

“Yes, the Ghosts have arrived. And we need weapons. Badly.”

“So, why not let me stay in my lab? Why send me down there?”

“The dense pellets you were working with… those were only a taste. There are ultra-dense pellets down there, deep in the mines, that we don’t know how to transport yet. We’re going to get you a special PAC that can handle your lab equipment, so you can travel down there and study them yourself.”

You nod. Suddenly it makes sense, why you were suddenly summoned. Why Terrance Mayhew himself is overseeing your training. Why you were given funding for the project after years of petitioning. There are millions of thoughts spinning around your head, but for now you have just one question for Terrance “Will you be accompanying me down into the mines?”

“Yes. I can’t pretend to understand the details of your work,” (you internally jump with pride), “but I’ll be there to oversee the mechanical details.”

“When do we leave?”

“Well, that’s up to you and Matteo. Matteo?”

Matteo has clearly been dumbstruck by the revelations that were just shared, but he regains his composure quickly. He’s a professional.

“We should, uh, we should work on that. It should be… we will try to make it quick.” He turns to you, and as he starts speaking of training his voice loses the quaver and becomes calm and steady.

“The pellets here, the pellets you’ve been studying, here they are in their unhusked state. They usually grow in tighter clusters, but this mine is harvested every time a new trainee comes through.” He pauses to collect himself, and looks up to the office bloc where Terrance must be. “Now get collecting.”

8: Pellets

In the last chapter we refactored from using just an array of walls to using a grid. While our game ended up looking exactly the same afterwards, we were better able to reason about how the game world worked. In this chapter we'll be taking advantage of that refactor to add pellets to the game world.

Here's the grid after we fill it with pellets:

```
// 0 is a blank space
// 1 is a wall
// 2 is a pellet
grid: [
  [2, 2, 2, 2, 2, 2, 2, 1],
  [2, 1, 2, 1, 2, 2, 2, 1],
  [2, 2, 1, 2, 2, 2, 2, 1],
  [2, 2, 2, 2, 2, 2, 2, 1],
  [2, 2, 2, 2, 2, 2, 2, 1],
  [1, 2, 2, 2, 2, 2, 2, 1],
],
```

Imagine doing that with just an array of walls and an array of pellets- it would be very difficult to imagine, and would likely lead to errors like putting multiple pellets on the same cell, or putting a wall and a pellet on top of each other. Here we see that there can be exactly one thing in each cell, and (right now) it's either a wall or a pellet.

But just having them in our grid doesn't do much. We need to draw the pellets.

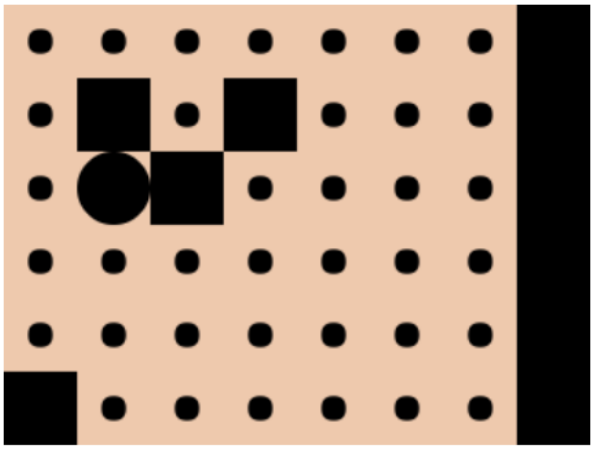
Drawing Pellets

In the `drawWalls` method, we loop through all the cells in the grid via a nested loop, and then if a cell in the grid is a wall (`cell == 1`) we draw a rectangle.

Let's take advantage of all that machinery we've already built and just add drawing a pellet to it:

```
drawWalls: function(){
  let squareSize = this.get('squareSize');
  let ctx = this.get('ctx');
  ctx.fillStyle = '#000';

  let grid = this.get('grid');
  grid.forEach(function(row, rowIndex){
    row.forEach(function(cell, columnIndex){
      if(cell == 1){
        ctx.fillRect(columnIndex * squareSize,
                     rowIndex * squareSize,
                     squareSize,
                     squareSize)
      }
      //new stuff starts here
      if(cell == 2){
        let pixelX = (columnIndex + 1/2) * squareSize;
        let pixelY = (rowIndex + 1/2) * squareSize;

        ctx.beginPath();
        ctx.arc(pixelX, pixelY, squareSize/6, 0, Math.PI * 2, false);
        ctx.closePath();
        ctx.fill();
      }
    })
  })
},
```

So after we check whether the cell contains a wall, then we check whether it contains a pellet. If it does, we draw a circle using code that looks remarkably like that we use to draw the PAC (but more on that later).



Say what you do, do what you say

Of course, now `drawWalls` is doing more than just drawing walls. That's confusing for future developers (including future you). We could expand it to `drawWallsAndPellets`, but I think a better name is `drawGrid`. That leaves us free to add more stuff that just walls and pellets without changing the name again.

So we change the name of the drawWalls function to `drawGrid`, and then change it in the two places it's called. If you're not sure exactly what I mean, check out the change in the git history[9].

Now `drawGrid` is named correctly, but we can still do better. There are two masses of code, one for drawing a wall and one for drawing a pellet. Let's separate those out and give them names:

```
drawWall: function(x, y){
  let ctx = this.get('ctx');
  let squareSize = this.get('squareSize');

  ctx.fillStyle = '#000';
  ctx.fillRect(x * squareSize,
               y * squareSize,
               squareSize,
               squareSize)
},

drawPellet(x, y){
  let ctx = this.get('ctx')
  let squareSize = this.get('squareSize');
```

[9] https://github.com/jeffreybiles/chapter-by-chapter-game/commit/244f311bc0e04add2c36be49783a740bffd887f3

```
15
16    let pixelX = (x+1/2) * squareSize;
17    let pixelY = (y+1/2) * squareSize;
18
19    ctx.fillStyle = '#000';
20    ctx.beginPath();
21    ctx.arc(pixelX, pixelY, squareSize/6, 0, Math.PI * 2, false);
22    ctx.closePath();
23    ctx.fill();
24  },
25
26  drawGrid: function(){
27    let grid = this.get('grid');
28    grid.forEach(function(row, rowIndex){
29      row.forEach(function(cell, columnIndex){
30        if(cell == 1){
31          this.drawWall(columnIndex, rowIndex);
32        }
33        if(cell == 2){
34          this.drawPellet(columnIndex, rowIndex);
35        }
36      })
37    })
38  },
```

This is so much better! drawGrid takes care of looping through the grid, and then it calls out to `drawWall` and `drawPellet` to draw each entity. We've renamed the long `columnIndex` and `rowIndex` to `x` and `y`, which will be intuitively familiar to anyone who's worked with graphs. Even better, it's now clear from the context that '1' means wall and '2' means pellet.

Scope Troubles

But it's not all sunshine and roses (yet). If you try to run this code, you'll run into the following error: `Uncaught TypeError: Cannot read property 'drawPellet' of undefined`.

This happens because of how javascript handles scope.

Each time before, when we called `this`, we've been referring to the `PacMan` component. So `this.drawGrid()` means 'go to the component, find the method called `drawGrid`, and call it'.

However, in the above code the scope of `this` has changed. It changes because of the functions we feed into the `forEach` function. We haven't given those functions a name, so we'll call them 'anonymous functions'. When you create an anonymous function, you create a new scope, so the meaning of `this` changes. And on this new scope, there is no `drawPellet` function.

Going back to an old analogy, it's the difference between saying "find the couch in this house" and "find the couch in this kitchen". You're in the wrong scope!

Fat Arrows

This scope change is terrible and is the cause of lots of bugs. People who write javascript have come up with lots of clever ways to get around it, but by far the best is a new language feature nicknamed 'fat arrows'. It's basically a way of declaring anonymous functions that is shorter and doesn't change the scope.

```
// old anonymous function declaration
console.log(this) // foo
myArray.forEach(function(item){
  console.log(this) // bar
})

// new fat arrow function declaration
myArray.forEach((item)=>{
  console.log(this) // foo
})
```

So using the fat arrow (`()=>`) instead of the function keyword (`function()`) preserves the scope instead of creating a new one.

This is how the drawGrid function looks now:

```
drawGrid: function(){
  let grid = this.get('grid');
  grid.forEach((row, rowIndex)=>{
    row.forEach((cell, columnIndex)=>{
      if(cell == 1){
        this.drawWall(columnIndex, rowIndex);
      }
      if(cell == 2){
        this.drawPellet(columnIndex, rowIndex);
      }
    })
  })
})
```

Now we're back to seeing beautiful walls and pellets everywhere. Thanks, fat arrow!

Why a fat arrow? What's a skinny arrow?

Great questions!

One upon a time there was a language called 'coffeescript'. It was a language that looked a lot like javascript, but it was far more beautiful. It kept many of the core concepts while sanding down the rough edges.

One of the rough edges was function declaration. It had a skinny arrow '->' that worked just like the function keyword, but it was shorter. It also had a fat arrow '⇒' that preserved the scope.

When it came time to update javascript, the creators realized how wonderful coffeescript was and stole as many of its best ideas as they could while still preserving backwards compatibility with old versions of javascript. One of those ideas was the fat arrow, although it is now sadly bereft of its sibling the skinny arrow.

Still having troubles? Check to make sure that you've changed both of calls to `drawWalls` to say `drawGrid`.

Share your circle

While we're on the subject of refactoring our code to make it more clear, you may notice that `drawCircle` and `drawPellet` look almost exactly the same:

```
1  drawPellet(x, y){
2    let ctx = this.get('ctx')
3    let squareSize = this.get('squareSize');
4
5    let pixelX = (x+1/2) * squareSize;
6    let pixelY = (y+1/2) * squareSize;
7    let radius = squareSize / 6;
8
9    ctx.fillStyle = '#000';
10   ctx.beginPath();
11   ctx.arc(pixelX, pixelY, radius, 0, Math.PI * 2, false);
12   ctx.closePath();
13   ctx.fill();
```

```
},

drawCircle: function() {
  let ctx = this.get('ctx');
  let squareSize = this.get('squareSize');
  let x = this.get('x');
  let y = this.get('y');

  let pixelX = (x+1/2) * squareSize;
  let pixelY = (y+1/2) * squareSize;
  let radius = squareSize / 2;

  ctx.fillStyle = '#000';
  ctx.beginPath();
  ctx.arc(pixelX, pixelY, radius, 0, Math.PI * 2, false);
  ctx.closePath();
  ctx.fill();
}
```

Not only that, but when we say `drawCircle`, what we really mean is `drawPac`. So let's rename that, then make it so these two functions can share some code.

```
drawPac(){
  let x = this.get('x');
  let y = this.get('y');
  let radiusDivisor = 2;
  this.drawCircle(x, y, radiusDivisor)
},

drawPellet(x, y){
  let radiusDivisor = 6;
  this.drawCircle(x, y, radiusDivisor)
},

drawCircle(x, y, radiusDivisor) {
  let ctx = this.get('ctx')
  let squareSize = this.get('squareSize');

  let pixelX = (x+1/2) * squareSize;
  let pixelY = (y+1/2) * squareSize;

  ctx.fillStyle = '#000';
```

```
21    ctx.beginPath();
22    ctx.arc(pixelX, pixelY, squareSize/radiusDivisor, 0, Math.PI * 2, false);
23    ctx.closePath();
24    ctx.fill();
25  },
```

As you can see, it was only a few lines difference between `drawPac` and `drawPellet`. Now we just need to go back to where we were drawing the PAC with `drawCircle` and call `drawPac` instead.

Processing the pellets

So we've spent a lot of time drawing the pellets and then making our code better, but the PAC is going to have to process the pellets. We'll start that here, and then finish in the next chapter. Our goal right now is to make the pellets disappear when the PAC visits their cell.

We'll start with figuring out where we want to place this code. We should check for any pellets every time the PAC moves to a new square, and then process the pellet if it exists. So let's put it in the `movePacMan` method.

```
1   movePacMan(direction, amount){
2     this.incrementProperty(direction, amount);
3
4     if(this.collidedWithBorder() || this.collidedWithWall()) {
5       this.decrementProperty(direction, amount)
6     }
7
8     this.processAnyPellets()
9
10    this.clearScreen();
11    this.drawGrid();
12    this.drawPac();
13  },
```

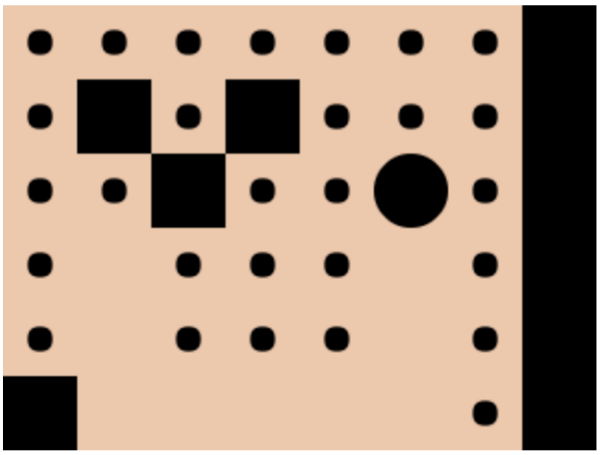
So we put the call to `processAnyPellets` after moving and checking for collisions with walls, but before redrawing everything.

Here's what the `processAnyPellets` function looks like:

```
processAnyPellets: function(){
  let x = this.get('x');
  let y = this.get('y');
  let grid = this.get('grid');

  if(grid[y][x] == 2){
    grid[y][x] = 0;
  }
},
```

It checks to see if the cell is equal to '2' (to see if it's a pellet), and if it is then it sets the cell to '0' (a blank space). This will replace the '2' that was in that cell of the grid with a '0'. It's an assignment, just like we could assign to the variable 'x', 'y', or 'grid', but it's reaching into the nested arrays to make the assignment.

Now run the code and see the pellets disappear underneath your PAC!



But before we move on, let's notice two things about this code. The first is that '==' and '=' look very similar, even though they do very different things. This is something you'll have to stay on the lookout for. A common error is to accidentally mix them up, and to assign to a value to a variable when you wanted to check to see if it was equal to the variable, or to return a boolean when you wanted to make an assignment.

The second thing you may notice is the similarities with the `collidedWithWall` function:

```
collidedWithWall: function(){
  let x = this.get('x');
  let y = this.get('y');
  let grid = this.get('grid');

  return grid[y][x] == 1
},
```

But in this case we won't refactor, because although they are superficially similar they are serving quite different purposes and combining them would make the code more difficult to read, not less. Although it's generally a good idea to refactor away repeated code, doing it too zealously can also lead to trouble.

Summary

In this chapter we added pellets to our grid, drew them, refactored our code so it is named better and repeats itself less, and started processing the pellets (by making them disappear). Along the way we learned about fat arrows, scope, assignments into nested arrays, and we practiced our refactoring skills.

In the next chapter we finish processing the pellets, giving us a score. We also decide what to do when we've finished processing all the pellets.

Gains and Losses

You're still shaking a bit from the revelations, but now you have to focus- can't go smashing into walls now just because you're important to some master plan to save humanity.

Matteo does his part to keep things normal, lecturing you on the economics of pellet collection.

"Each pellet represents a certain amount of energy- you're seeing that on your score counter. That energy is used to feed our people, build new tunnels, and in general keep the Caliphate running." It's review, this is literally what you used to spend all day studying, but you let him go on. It's keeping your mind from everything else.

But then he goes into territory you're not as familiar with. "Your PAC takes energy as well. Generally, the farther down you go, the more energy the PAC takes- both because it has farther down to go, and also because the atmosphere messes with our internal pellet processing."

That explanation doesn't sound quite right, but it's obviously enough for the PAC-men. You make a mental note to learn more about the phenomenon once this is over. See! You're already normalizing things. Matteo is still talking. "The training PAC you're in is one of the least efficient models, but that's okay- you only use it on the upper levels, and only during training. As it is, running it on this level is about breaking even in terms of energy usage."

Astoundingly, you haven't hit any walls yet- you're getting better! -and then you pick up the final pellet. Your screen darkens and you hear a new mechanical noise, one distinct from the noises of the PAC. Larger machinery.

"What's happening?"

"Ah yes, that's… we should have turned that feature off. If no nearby pellets are detected, your PAC automatically goes to the next level. It's fantastically efficient, in terms of pellets per minute. Putting in the elevators was expensive, but they've more than paid for themselves."

You absentmindedly collect pellets while he talks, and you realize: this could be fun. This could definitely be fun.

9: Scores and Levels

Keeping Score

In the last chapter we started processing the pellets, but really all we did was make them disappear. In this chapter, we're going to give those pellets a higher purpose. And that higher purpose is keeping score.

Let's start be adding a score property to the PacMan component:

```
export default Ember.Component.extend(KeyboardShortcuts, {
  score: 0,
  ...
});
```

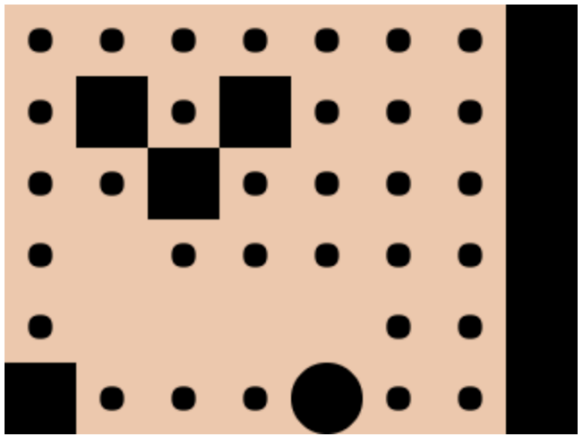
Then we'll show that score in the handlebars template for that component, right below the playing field:

```
<!-- templates/components/pac-man.hbs -->
<canvas id="myCanvas" width={{pixelWidth}} height={{pixelHeight}}></canvas>
<br>
Score: {{score}}
```

Remember, `{{score}}` is tracking the score property on the pac-man component. So right now it shows a score of 0, and it will keep showing that score, no matter where our PAC goes, until we write code that makes it change. Luckily, that's relatively easy:

```
processAnyPellets: function(){
  let x = this.get('x');
  let y = this.get('y');
  let grid = this.get('grid');

  if(grid[y][x] == 2){
    grid[y][x] = 0;
    this.incrementProperty('score')
  }
},
```

We just added the line that increments the score after the line where we removed the pellet. Start playing and the score will go up! One of the awesome things about Ember is that the properties in the template automatically change whenever they change elsewhere. That's called 'binding', and it's used heavily in traditional Ember applications.



Score: 6

So that was relatively easy. Let's do something more difficult.

Level Up

The initial part of creating the level system is very similar.

```
let PacMan = Ember.Component.extend(KeyboardShortcuts, {
  levelNumber: 1,
  ...
});
```

```
<canvas id="myCanvas" width={{pixelWidth}} height={{pixelHeight}}></canvas>
<br>
Score: {{score}}       Level: {{levelNumber}}
```

We start on level 1, and the level is shown after the score. ` ` is html code for 'non-breaking space'. It's a way to get some space between 'Score' and 'Level', since multiple plain spaces in html collapse down to one space.

Next we need to decide where to increment the level. Incrementing the score was easy- we were already doing something every time we ran across a pellet, and so we could attach the score increment action to that piece of code. For dealing with levels we'll have to create code that detects when the level is complete.

Our criteria for a level being done is that the player has picked up all the pellets. Here's the code:

```
levelComplete: function(){
  let hasPelletsLeft = false;
  let grid = this.get('grid');

  grid.forEach((row)=>{
    row.forEach((cell)=>{
      if(cell == 2){
        hasPelletsLeft = true
      }
    })
  })
  return !hasPelletsLeft;
},
```

Here's what's happening in this function:

1. We first set the `hasPelletsLeft` to false.
2. We loop through every cell in the grid. If any of them are equal to '2' (if they have a pellet) then we set `hasPelletsLeft` to true.
3. We return the opposite of `hasPelletsLeft` (`!` means 'not', so it would turn a true into a false, and vice versa). So if it detected a pellet and `hasPelletsLeft` was true, `levelComplete` would return false. If no pelleted were detected and `hasPelletsLeft` stayed false, `levelComplete` would return true.

So that's what we use to determine whether a level is complete. We can check that criteria every time the PAC picks up a pellet:

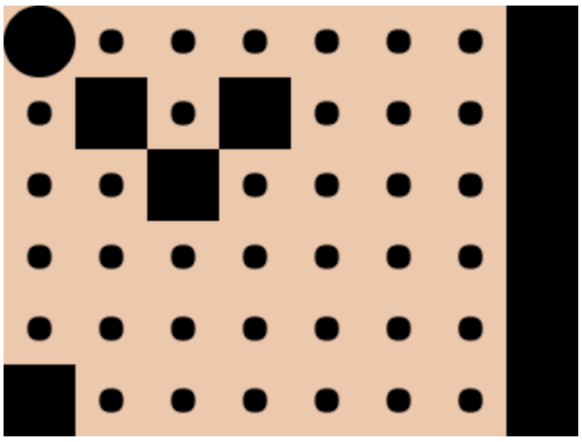
```
processAnyPellets: function(){
  let x = this.get('x');
  let y = this.get('y');
  let grid = this.get('grid');

  if(grid[y][x] == 2){
    grid[y][x] = 0;
    this.incrementProperty('score')

    if(this.levelComplete()){
      this.incrementProperty('levelNumber')
      this.restartLevel()
    }
  }
},
```

After a pellet is picked up and the score is incremented, we then check to see if it was the last one (if the level is complete). If it is, then we increment the `levelNumber` property and then restart the level.

Here's what restarting the level looks like:

```
restartLevel: function(){
  this.set('x', 0);
  this.set('y', 0);

  let grid = this.get('grid');
  grid.forEach((row, rowIndex)=>{
    row.forEach((cell, columnIndex)=>{
      if(cell == 0){
        grid[rowIndex][columnIndex] = 2
      }
    })
  })
},
```

First we move the PAC to `[0,0]` (the far upper left), then we go through each empty cell in the grid and put a pellet in it.



New Syntax

Remember when we said there was a better way to declare functions? We're introducing it now, because I want to start writing with it. The good news is that it's fairly simple and it will make our code look nicer.

So let's say you had a function like this:

```
addTwo: function(num){
  return num + 2;
}
```

Pretty cool... but we can make it more concise:

```
addTwo(num){
  return num + 2
}
```

Now that you know that `name(){}` is a valid way to declare a function, and is in fact the same as `name: function(){}`, you can see that the colon and the word `function` don't really add much information. They were nice for a while, to remind you that it's a function that we're creating, but now we don't need them.

We're going to make this syntax change everywhere that we declare a function. Recording all of those here in the book would be silly, so we're just going to link to the github diff where we made all those changes[10].

> If you're using a text editor with a nice search-and-replace feature, try searching for ': function(' and replacing all instances with '('.

Summary

In this chapter we reviewed a lot of concepts, especially the 'cycle through the grid' pattern. We learned a few minor things, like the boolean `!`, handlebars bindings, and a new function declaration, but mostly this was a respite to solidify what we already know and get some cool features into the game.

Now it's time to buckle up, because the next few chapters are going to be a doozy.

[10] https://github.com/jeffreybiles/chapter-by-chapter-game/commit/deac876c39833845bcb7ef905f913e9d6307b07f

Smooth, Efficient

"I've heard rumors that the ghosts are getting closer."

Jerome looks up from his plate.

"They say the resistance in Asia is gone."

He shrugs. "That isn't really our problem." But it was a tense, practiced shrug. Everything about him was tense. "Let's talk about your training."

"I've heard the chatter since coming here. After they finish with Asia, there's nothing stopping them from coming to California."

His eyes narrowed while he tried to look relaxed. It wasn't working. "There was never anything stopping them from coming to California. They're aliens. They have spaceships. If they wanted to be here, they'd be here." Was that supposed to be comforting? "Now, let's talk about your training."

You sigh. "Fine, let's talk about my training."

"How are you stress levels?"

You laugh. His jaw is clenched and his fist is gripping his fork like it's going to run away and start a Jihad. "How are *your* stress levels?"

He pauses, then laughs too abruptly. His manner changes completely. "Sorry, there's been some stuff at the lab, I just… I wanted to make sure that you were doing well. That your body was holding up under the new stress."

"I'm fine. No new bruises."

"Good. I was worried because they're about to start you on the next level of training PAC."

"Really?"

He pauses. "Or… maybe not. That's what they usually do. I mean, the sequence is, about this time in training, they start you on a new one."

"They say my training is going faster than normal."

"Yes, um, it… anyways, I'm glad you're feeling better. Maybe they slowed it down due to the bruises you had before."

Why is Jerome being so weird today?

"Tell me about your stresses at work," you say.

"Ah, it's… you know, end of the world and all." He's back to his normal self, joking his way through disaster. "We're working on a new variation of the PAC, one designed to handle science equipment. I recognize some of the slots."

You think about telling him that it's for you, but you refrain. You weren't sure what passed for a military secret around here, and they obviously hadn't told him.

"Anyways, I gotta go. And you gotta train."

You nod and take your leave. The entire walk to the training facilities you're haunted by some weird sense that something was up. Something more than the conspiracies you're already privy to.

Matteo greets you, and you see Terrance waving from a glassed-in room filled with monitors.

"We're moving you up today," says Matteo. "The next level of training PAC."

His explanation is rapid-fire- the new PAC moves smoothly from square to square instead of jumping, and while that makes control more difficult, it also makes it much more efficient energy-wise.

Jerome had been right- they were moving you up to a different PAC. How had he been so precise, so accurate, in his prediction? Maybe it was just coincidence.

You jump in, start collecting pellets, and let your mind go blank.

10: Animations

So, this is the chapter where I discovered that Pac-Man *might* not be the best game to use for teaching beginners... But hey, real life programming will often have you jumping out of your depth and then trying to swim to shore before you drown. So this is great prep.

> In real life programming, there are also sharks.

The first part of this chapter will be another "rearrange stuff and then everything plays the same" section ("Refactoring"). The worst part is that *why* we're doing the changes won't make much sense until we get to the second part of this chapter. The good news is that you'll learn a bit of new modern Javascript syntax, and get to see hashes used in more complex ways.

By the end of this chapter our PAC will be gliding smoothly between squares, instead of teleporting from square to square and taking up all the pellet energy.

Coordinating Directions

Let's start with the mysterious centerpiece of the refactor:

```
1  directions: {
2    'up': {x: 0, y: -1},
3    'down': {x: 0, y: 1},
4    'left': {x: -1, y: 0},
5    'right': {x: 1, y: 0},
6    'stopped': {x: 0, y: 0}
7  },
```

This is a hash that contains 5 other hashes- one each for `up`, `down`, `left`, `right`, and `stopped`. Each of those hashes contains exactly two keys: `x` and `y`. They codify what each direction means in terms of coordinates. It's pretty basic stuff, but the way the information is stored may look a little different.

Next we change what we're passing to `movePacMan` in `keyboardShortcuts`.

```
keyboardShortcuts: {
  up() { this.movePacMan('up');},
  down()  { this.movePacMan('down');},
  left() { this.movePacMan('left');},
  right() { this.movePacMan('right');},
},
```

Previously, we were passing it a coordinate (like `x`) and then a value (either `1` or `-1`). Now we just pass them a string representing a direction. You can probably guess where this is going.

Let's take a high-level overview of the new `movePacMan` function, and then we'll go deeper into the new functions that we're using within it.

```
movePacMan(direction){
  if(!this.pathBlockedInDirection(direction)){
    this.set('x', this.nextCoordinate('x', direction));
    this.set('y', this.nextCoordinate('y', direction));

    this.processAnyPellets();
  }

  this.clearScreen();
  this.drawGrid();
  this.drawPac();
}
```

The last three commands are familiar, but the initial `if` statement is all new. You'll notice that we no longer check for `collidedWithBorder` or `collidedWithWall`... we can safely delete those functions. Replacing both of those is `pathBlockedInDirection`.

If the path *isn't* blocked (notice the `!` within the `if` statement), we set the `x` and `y` coordinates to their 'next' values, based on the `nextCoordinate` function. Then we process the pellets there.

We have two new functions to deal with. We'll tackle `nextCoordinate` first, because it's simpler and will give us a good first look at how we're using the `directions` hash.

nextCoordinate and Interpolated Strings

```
nextCoordinate(coordinate, direction){
  return this.get(coordinate) + this.get(`directions.${direction}.${coordinate}`\
);
},
```

We're being passed a `coordinate` ('x' or 'y') and a `direction` ('up', 'down', etc.). In the body of the function we're adding the results of two expressions. The first is the current `x` or `y` value. The second is digging deep into the `directions` hash.

This is some new syntax- the interpolated string. If the `direction` is 'up' and the `coordinate` is 'y', then our second expression will end up as `this.get('directions.up.y')` and resolve to `-1`. If it was 'right' and 'y', then our second expression would in up as `this.get('directions.right.y')` and resolve to `0`.

> Having trouble? Make sure to use the 'back-tick' character for string interpolation, not the single quote character. On most keyboards it's right below the escape key.

If you don't understand how the different variants on the hash resolve as they do, look back to chapter 7 when we defined `screenWidth` and `screenHeight`. Here, like there, we dig several layers in with one `.get` by having a string with multiple `.`s. The different is now we're defining some of the links on the fly.

If it helps, we could also have expressed `nextCoordinate` as:

```
nextCoordinate(coordinate, direction){
  return this.get(coordinate) + this.get('directions').get(direction).get(coordi\
nate);
},
```

In words, this function grabs the current value of the `x` or `y` coordinate, then uses the `directions` hash and the stated direction to figure out if and how much that value needs to change. It returns the new value.

In our usages within `movePacMan`, we set the `x` or `y` coordinate to the returned value.

Looking ahead with the Grid

Now we need to figure out whether our path is blocked or not. We'll do that using our `directions` hash and our `grid` array of arrays.

You'll recall our grid, shown in a very simplified form here:

```
grid: [
  [1, 2, 2],
  [2, 2, 1],
  [1, 2, 1]
]
```

`1` represents a wall, while `2` represents a pellet (`0` represents a blank space). What we're going to do, much like we did before with `collidedWithWall` and `collidedWithBorder`, is to project into the future and see if we're in a good spot or a bad spot.

```
pathBlockedInDirection(direction) {
  let cellTypeInDirection = this.cellTypeInDirection(direction);
  return Ember.isEmpty(cellTypeInDirection) || cellTypeInDirection === 1;
},

cellTypeInDirection(direction) {
  let nextX = this.nextCoordinate('x', direction);
  let nextY = this.nextCoordinate('y', direction);

  return this.get(`grid.${nextY}.${nextX}`);
},
```

So first we run `cellTypeInDirection`. That method uses `nextCoordinate`, which we just explained in the previous section, to get the next `x` and `y` values. Then, we're using string interpolation to find that position on the grid.

One interesting thing to note is that it's possible to give `x` and `y` values that end up being *off* the grid. That's fine- it'll just return `undefined`. Then that `undefined` will be caught by `Ember.isEmpty(cellTypeInDirection)` before we check that `cellTypeInDirection === 1`.

The end result is that if the next cell is out of bounds (`undefined`) or a wall (`1`), then we return `true` from `pathBlockedInDirection`.

Smooth Animations

The changes in this chapter so far have merely been setup; they end up doing exactly the same thing as before. In this second half of the chapter we're going to finally make the switch from choppy jumping to a smoooooth animated roll.

More specifically, when we hit a direction, the PAC will start rolling in that direction. Once it has reached the new square, it will stop. While it's rolling, it' won't take any input, but once it's stopped you can move it again (this particular will change in the next chapter).

Guarding our movement

Previously, we've been able to update state immediately after the action is triggered. Now we need to stretch it out over time, and stop taking input while we're animating. This is the first part of that, showing how we block off input while the animation is taking place.

```
isMoving: false,
direction: 'stopped',
movePacMan(direction){
  if(this.get('isMoving') || this.pathBlockedInDirection(direction)){
    // do nothing, just wait it out
  } else {
    this.set('direction', direction)
    this.set('isMoving', true)
    this.movementLoop()
  }
},
```

The purpose of the if statement is very simple: if we're already moving, or if the path is blocked... don't do anything! Otherwise, start the movement loop.

The way I constructed the if statement may be a bit controversial... it's usually not great to have a blank first clause in an if/else statement. Another option is as follows:

```
movePacMan(direction) {
  let inputBlocked = this.get('isMoving') || this.pathBlockedInDirection(directi\
on)
  if(!inputBlocked){
    this.set('direction', direction)
    this.set('isMoving', true)
    this.movementLoop()
  }
},
```

I like the first version because it's clear that `isMoving` leads to a dead end, while with the second version we have to parse it a bit more. However, like I said above, it's generally bad form to have a blank first clause in in if/else statement. These are just some of the tradeoffs you'll have to make when writing code.

> It also helps that I know what the code will eventually look like, and I know we'll eventually do something in the first branch of the if/else statement.

So if you're not moving, and the path isn't blocked, we set up the direction, set `isMoving` to `true`, then run `movementLoop`.

The Movement Loop

Here's what the movement loop looks like:

```
frameCycle: 1,
framesPerMovement: 30,
movementLoop(){
  if(this.get('frameCycle') == this.get('framesPerMovement')){
    let direction = this.get('direction')
    this.set('x', this.nextCoordinate('x', direction));
    this.set('y', this.nextCoordinate('y', direction));

    this.set('isMoving', false);
    this.set('frameCycle', 1);

    this.processAnyPellets();
  } else {
    this.incrementProperty('frameCycle');
    Ember.run.later(this, this.movementLoop, 1000/60);
  }

  this.clearScreen();
  this.drawGrid();
  this.drawPac();
},
```

There's quite a bit going on here, so we'll parse it piece by piece.

First, we've set up `frameCycle` and `framesPerMovement`. `framesPerMovement` is meant to be a constant, and `frameCycle` is meant to change. Specifically, we'll be incrementing `frameCycle` by 1 every time we go through the loop. Here's the simplified loop, replacing lots of stuff with comments and leaving only the skeleton of the loop.

```
movementLoop(){
  if(this.get('frameCycle') == this.get('framesPerMovement')){
    //... reset direction
    this.set('isMoving', false);
    this.set('frameCycle', 1);
    //...process pellets
  } else {
    this.incrementProperty('frameCycle');
    Ember.run.later(this, this.movementLoop, 1000/60);
```

```
    }
    //...drawing stuff
},
```

You'll see that the core is an if/else statement. If `frameCycle` is less than `framesPerMovement`, then increment it by 1, then rerun everything 33 milliseconds later (1000/60 implies 60 frames per second). Once `frameCycle` is equal to `framesPerMovement`, we reset everything- setting `isMoving` to false, setting `frameCycle` to 1, moving to the next `x` and `y` coordinates, and eating the pellet with `processAnyPellets`. We also stop the loop, since we don't call it again. Of course, after each run through the loop, we redraw everything.

Ember.run.later

`Ember.run.later` is a construct we haven't seen yet. It does pretty much exactly what it says on the tin- it says "Ember, run this later". The details are that the first argument is the scope to run the function in, the second argument is the function to run, and the third argument is how many milliseconds later to run it. Notice that `this.movementLoop` doesn't have `()` at the end- that's because we're not running it right now, just putting the *function itself* in as an argument so it can be run later.

> Old-school Javascripters may notice than `Ember.run.later` fills a similar role as `setTimeout`. This is true. With Ember, you'll never have to use `setTimeout` ever again.

Drawing with FrameCycles

So our code is properly looping, taking its time- you can see that by the delay between pressing a button and seeing the PAC move- but it's not animating yet. To make that happen, we'll have to make our `drawCircle` function aware of directionality and `frameCycle`.

We'll start by passing in a direction to the `drawCircle` function:

```
drawPac(){
  let x = this.get('x');
  let y = this.get('y');
  let radiusDivisor = 2;
  this.drawCircle(x, y, radiusDivisor, this.get('direction'));
},

drawPellet(x, y){
  let radiusDivisor = 6;
  this.drawCircle(x, y, radiusDivisor, 'stopped');
```

```
11  },
12
13  drawCircle(x, y, radiusDivisor, direction) { /*...*/}
```

The `drawPellet` function will always pass in `stopped`, since the pellets never move. The `drawPac` function, on the other hand, will pass in the direction that the PAC is currently moving.

Then, within the `drawCircle` function, we change how `pixelX` and `pixelY` are calculated:

```
1   drawCircle(x, y, radiusDivisor, direction) {
2     let ctx = this.get('ctx')
3     let squareSize = this.get('squareSize');
4
5     let pixelX = (x + 1/2 + this.offsetFor('x', direction)) * squareSize;
6     let pixelY = (y + 1/2 + this.offsetFor('y', direction)) * squareSize;
7
8     ctx.fillStyle = '#000';
9     ctx.beginPath();
10    ctx.arc(pixelX, pixelY, squareSize/radiusDivisor, 0, Math.PI * 2, false);
11    ctx.closePath();
12    ctx.fill();
13  },
```

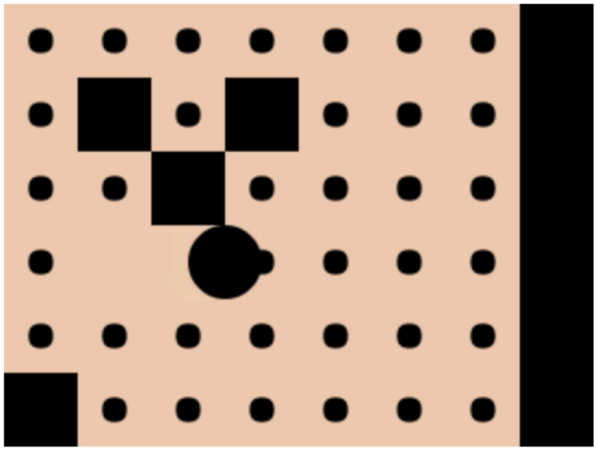
In addition to adding `1/2` to `x` and `y`, we're also adding in `offsetFor`. That's how far the circle is offset from the center, which is calculated using- you guessed it- the `frameCycle`.

```
1   offsetFor(coordinate, direction){
2     let frameRatio = this.get('frameCycle') / this.get('framesPerMovement');
3     return this.get(`directions.${direction}.${coordinate}`) * frameRatio;
4   },
```

So we get the `frameRatio`, which is basically the percentage of the movement that has been completed. It will always be between 0 and 1.

Then we multiply the `frameRatio` by the value we get from digging in to `directions` once again. So if we're on frame 10 (out of 30), and our `direction` is 'right', then we'll be 1/3 of the way between the square where we started and the square to the right of it. By incrementing the `frameCycle`, we're moving the PAC slowing to the right, towards its destination square.

And now we have an animating circle!



Summary

Wow, so that was the longest chapter by far, and I asked you to do some pretty intense thinking. Congratulations on making it through!

This chapter introduced string interpolation, `Ember.run.later`, and the concept of an animation loop. In addition, we asked you to reason deeply about hashes and do some fairly complex refactors.

There are still plenty of challenges ahead, but you've proven that you can handle them.

No Stopping

Adjusting to the new controls took some time, but you were able to handle it well enough after about an hour. Matteo kept giving you new challenges, each slightly more complex than the last, and by three hours in it was as if your PAC had always moved like this. When you finally step out of the PAC, your muscles twitch to anticipate jolts and shakes that are no longer coming.

Dinner is strange- a quiet tension has settled over the group. The once-jovial atmosphere had excluded you, and it felt oppressive at the time, but this is even worse. There are whispers, then a too-loud, too-short laugh. The sound of metal scraping against plastic.

Jerome sits down across from you. "How's the training?"

"It was exactly like you said. We started the next level of training."

"Good," he says. He looks like he's about to say something else, but then he just nods. "Good."

"Why's everyone so quiet?"

He looks around, leans in closer, whispers even quieter. "The ghosts disappeared from Asia."

"So the resistance really is gone."

He nods. "I couldn't tell you today at lunch, but now it seems like everyone knows. No use keeping you even more in the dark."

"But you didn't just say the resistance is gone… you said that the ghosts disappeared as well."

"Yea. Our tracking equipment had been picking them up en masse, but now there's only a few wandering the pellet tunnels."

"Looking for stragglers."

"Exactly. We didn't know their intentions before- maybe it was resources, maybe colonization- but after conquering Asia they just picked up and left. Kill and run."

"And where are they now?"

"That's the thing… we don't know. The ones that left Asia, we can't pick them up anymore. No one can."

You both quietly finish your meal, your thoughts filled with Armageddon. Were the ghosts the armies of Gog and Magog? Surely. But then again, Jesus wasn't coming to lead your armies. Both faiths agreed that He would do that. You guess that you'll find out which one is really true after the final battle, when you see whether He takes everyone to paradise or kills all the pigs and breaks all the crosses. Whether He kills you.

You're tapped on the shoulder, and he says that you've been called for additional training. You find that you don't mind.

Matteo nods at the PAC. It's subtly different than the one you used last time, and you find that it doesn't stop at regular intervals.

"Efficiency", he says when you ask.

The training continues.

11: The Game Loop

In the last chapter we smoothed out how the PAC moves... but it still stops every square! That's not optimally efficient! In this chapter we'll make it so it keeps on rolling without pressing a direction every square. Then we'll introduce the 'intent' system to make it manageable.

Rollin rollin rollin

The reason that the PAC currently stops is because the loop stops... once you reach the portion of the code where you move the PAC coordinates and collect the pellets, it just stops and then waits for instructions from the player. We can keep stuff going by just introducing an `Ember.run.later` into the other path within the if/else block.

> `movePacMan` requires an argument for the direction. In `Ember.run.later`, we can provide arguments to the passed function after we pass in the function but before we pass in the number of milliseconds to wait.

```
movementLoop(){
  if(this.get('frameCycle') == this.get('framesPerMovement')){
    let direction = this.get('direction')
    this.set('x', this.nextCoordinate('x', direction));
    this.set('y', this.nextCoordinate('y', direction));

    this.set('frameCycle', 1);
    this.set('isMoving', false)

    this.processAnyPellets();

    Ember.run.later(this, this.movePacMan, direction, 1000/60)
  } else {
    this.incrementProperty('frameCycle');
    Ember.run.later(this, this.movementLoop, 1000/60);
  }

  this.clearScreen();
  this.drawGrid();
  this.drawPac();
},
```

`movePacMan`, if you recall, calls `movementLoop` *if* `isMoving` is false and the next step in the current direction isn't blocked. So as long as those two conditions are met, the loop keeps going.

The net effect of this is that the PAC is out of control, keeping on going like it's on ice. This could make for some interesting puzzles, but it's not what we want.

But before we fix this, let's make it even more out of control:

```
didInsertElement() {
  this.movementLoop();
},
direction: 'down',
```

This starts the loop automatically instead of waiting for input, and starts it in the direction 'down'.

The Intent system

The easiest way of giving the player control is to respond instantly to their demands... but within our grid system, that would be a disaster. Instead we'll implement the 'intent' system, which stores the next desired direction of the user (their intent) and then acts on that when given a chance (when the PAC fully reaches the next square).

We'll deal with setting the intent first:

```
intent: 'down',
keyboardShortcuts: {
  up() { this.set('intent', 'up');},
  down()  { this.set('intent', 'down');},
  left() { this.set('intent', 'left');},
  right() { this.set('intent', 'right');},
},
```

So the intent starts 'down', and then we've changed the arrow keys so they change the intent instead of directly calling `movePacMan`.

Now we'll completely revamp `movePacMan`, even renaming it `changePacDirection` to better reflect its new responsibilities.

```
  changePacDirection(){
    let intent = this.get('intent')
    if(this.pathBlockedInDirection(intent)){
      this.set('direction', 'stopped');
    } else {
      this.set('direction', intent);
    }
  },
```

Up top you'll notice that we've change the name of `movePacMan` to `changePacDirection`. That's to reflect the fact that it's now `movementLoop` doing the moving.

Next you'll see that we're grabbing the `intent` from the component scope. That's the `intent` that's being set when we hit an arrow key. The most recently hit key is what's used. So we check that the `intent` won't run us against a wall, and then we change the `direction` to the `intent`. If it will run us against a wall, we change `direction` to stopped.

Finally, at the end, we've removed the call to `movementLoop`. You'll see when we get to `movementLoop` why that's possible.

> You may notice that we've removed `isMoving` completely, and the 'stopped' direction is taking over many of the jobs `isMoving` used to handle. `isMoving` was simply an intermediate solution introduced in order to keep the learning curve manageable while we built up a robust system. I hope it worked!

Now we're going to make our changes to `movementLoop`:

```
movementLoop(){
  if(this.get('frameCycle') == this.get('framesPerMovement')){
    let direction = this.get('direction')
    this.set('x', this.nextCoordinate('x', direction));
    this.set('y', this.nextCoordinate('y', direction));

    this.set('frameCycle', 1);
    this.processAnyPellets();
    this.changePacDirection();
  } else if(this.get('direction') == 'stopped'){
    this.changePacDirection();
  } else {
    this.incrementProperty('frameCycle');
  }

  this.clearScreen();
```

```
17    this.drawGrid();
18    this.drawPac();
19
20    Ember.run.later(this, this.movementLoop, 1000/60);
21  },
```

The first change is that we removed the reference to `isMoving`.

The second is that we added a new branch to the if/else statement. This block is entered when the `direction` is 'stopped'. This logic is taking the place of the `isMoving` check that used to be in `movePacMan`. The big difference is that now the loop keeps on rolling, drawing stuff and waiting for the the `intent` to change.

The last is that we've moved the recursive call to `movementLoop` to the end of the function. That means it gets called *every* time- there is no exiting this loop.

The effect of these changes is that when you press a button, it registers your intent, and then the PAC starts moving in that direction when it reaches the next square.

> It's worth noting that this is *very different* than the "Run Loop" in most Ember programs. That's because programming a game is very different in some ways than programming a typical web app. So even though they're both run loops, be sure not to mix up the game run loop and the Ember Run Loop (aside from `Ember.run.later`, we won't directly encounter the Ember Run Loop in this book).

Summary

In this chapter we completed our game loop, then attached it the intent system to provide for a smooth, advanced control system.

We've accomplished a lot in these last 11 chapters, but stuff has gotten a bit unwieldy. Our `pac-man.js` file is over 200 lines! In the next 3 chapters we're going to combine Ember's classes and mixins with modern Javascript's module system in order to separate out some of the logic and make the code easier to understand.

Objectifying

When you get back you collapse into your bunk. Another three hours of intense training have worn you thin, and you drop off to sleep almost immediately.

But it doesn't last. At 2am you awake with a start, half-remembering a dream with flames. A half hour later you realize that sleep will not come again, at least not soon, so you begin wandering the hallways.

They're different at night. The lights on the walls are dimmed, about a tenth of the illumination that they usually are. Humans may live underground, but the realities of their rhythms hadn't changed. That was the blessing of tradition, the wisdom of Islam. Not recognizing the value of tradition had been the downfall of the infidels.

But your thoughts tend to focus more on your own downfall- the one that had already started, that was now spreading beyond Asia. You walk and contemplate, tracing your hand along the wall, until you hear voices.

At first they're quiet, an echo in the distance. You begin to recognize the voices before you recognize words, and you can hear Jerome and Terrance, along with a third voice you don't recognize, and a fourth coming from a radio.

"We're still here," says the voice on the radio. "There aren't many of us left- a thousand, maybe, in Muharrad, and probably a few more in some other cities."

"Tell me more about the end," says the third, unfamiliar voice. "What happened, *exactly*?

"They just… left. They knew we were here, they *had* to know we were here, but they just up and left."

"Our sensors are still picking up a few Ghosts here and there."

"Well yes, there are some, but they're not vicious like they were earlier. No more slaughter since they pulled out. They're chasing us around underground, but only when we go after the pellets."

"So civilians are safe?"

"It would seem so, sir. But it hasn't been long. They may start again."

"That's a relief, as far as it goes. We'll call again at eleven hundred- have the standard reports ready by then. The Caliphate thanks you for your service."

"Thank you, General."

The hum of the radio cuts out.

"Jerome, Terrance," says the general, "How's your… project coming along."

"He's progressing faster than expected," says Terrance. "Even with our adjusted expectations."

“I told you,” says Jerome.

“I did have to divulge some classified info. I felt it was the best mitigation strategy at the time.”

“Understood”, says the general. “Jerome, what about on your end?”

“Nothing classified that wasn’t already in the rumor mill. And I don’t think he suspects.”

“Excellent, but I meant on his mental health.”

“Yes sir. He’s had a tight schedule, and some bumps and bruises, but I think he’s pushing through. He’s not fragile.”

“He’s had some PAC tremors,” interjects Terrance.

You tune them out as you come to the realization: Jerome is not your friend. Maybe he was, maybe he still is, but mostly now you are his test subject, his observed specimen. An act.

“His mental progression is also going well. He’s picked up a lot of new ways of thinking, as you’ve seen in the reports, but he still hasn’t really refactored his brain to operate like that of a PAC operator.”

“Fix that. Enough for him to survive, at least.”

“Yes sir.”

Eventually the conversation turns to other, less interesting topics. As you sneak back to your room, you decide not to let out how much you know. If they’re hiding something, so can you.

Sleeping was, somehow, not easier.

12: Separation of Concerns

We noticed at the end of the last chapter that the file `components/pac-man.js` was over 200 lines long. That's really long, and worse, it was getting hard to keep track of where we had put various functions. Maybe you stuck `offsetFor` in between `keyboardShortcuts` and `movementLoop` instead of with the drawing methods where it makes the most logical sense. Or maybe you have everything in a logical order, but there's just so much that it's difficult to remember what that logical order was.

In the next two chapters we're going to split up all this code into several logical sections based in different files and accessed in different ways. This adds a bit of overhead, but ultimately makes our job easier- especially when we later want to increase the complexity of the game (such as when we add ghosts).

This chapter will be a review of of the concepts we'll need to know in order to perform this separation, and the next chapter will be the actual separation.

Defining new files

We'll start by defining a Pac as a separate Ember Object. The following code is enough to get you started: "'js //models/pac.js import Ember from 'ember';

export default Ember.Object.extend({}) "'

This is pretty similar to what we had when we first created our component:

```
1  //components/pac-man.js
2  import Ember from 'ember';
3
4  export default Ember.Component.extend({})
```

That's no accident! There are two levels of similarity here.

> From now on, code samples will have a filename attached. Previously, they were assumed to be in components/pac-man.js unless otherwise specified.

Return of the ES2015 Modules

The first level of similarity is the basic form of a file within ember-cli. You import stuff, then you export stuff- and for many things, that export is done via extending an object.

Before we get to the second level of similarity, let's go ahead and get the Pac object within our pac-man component. We do thatÂ through the import keyword:

```
//components/pac-man.js
import Ember from 'ember';
import Pac from '../models/pac';
```

The Pac that we're importing is the *default* export from `../models/pac`. The file path has a '..' in it because we're using relative file paths, and '..' means 'go up one folder'. You could also use an absolute file path: `appName/models/pac`.

So the net result of using ES2015 modules (import/export) is that we can share code between files. That import line is the same as saying:

```
let Pac = Ember.Object.extend({})
```

As we fill up the Pac object with functionality (not to mention use it in several different contexts) then having it in a separate file becomes more and more useful.

Classy Systems

The second similarity between our Object declaration and our Component declaration is that they are both extended through a hash. Right now the Pac extension hash is empty, but it will be filled with values (mostly functions) attached to various keys, just like the Component extension hash is right now.

This similarity of form is because `Ember.Component` is actually just an extension of `Ember.Object`. `Ember.Component` is defined as `Ember.View.extend({...})`, which is defined as `Ember.CoreView.extend({...})`, which is defined by `Ember.Object.extend({...})`. So an Ember Component is just an Ember Object with a bunch of stuff added to it.

In our application thus far, the only portions of that added stuff we've taken advantage of so far are 1) the fact that we can stick a component into a handlebars, and 2) that we have an attached template with access to every variable within the scope. Everything else we've learned about the Component applies to the Object.

> Click here for a video on Ember's class system and the inheritance chain[11]
>
> If you've never used Ember before but some of this is looking familiar, it's probably because Ember has purposefully reconstructed something very similar to the Ruby object system- and in doing so, they've created a system that is far more familiar to most developers (including from other OO languages like Java or Python) than Javascript's default Prototype system.

[11] https://www.emberscreencasts.com/posts/40-class-inheritance

Mixins

There are some things we'll need access to in both the pac-man component and the pac object. For those, we'll use a Mixin. Mixins work by merging their hash with that of the object where they're mixed in. For example, here's a mixin being mixed in to three different Ember objects:

```
let PelletFiend = Ember.Mixin.extend({
  lovesPellets: true
})
let Pac1 = Ember.Object.extend(PelletFiend, {});
let Pac2 = Ember.Object.extend(PelletFiend, {});
let SickPac = Ember.Object.extend(PelletFiend, {
  lovesPellets: false
})
```

That's the exact same as defining them all as follows:

```
let Pac1 = Ember.Object.extend({
  lovesPellets: true
});
let Pac2 = Ember.Object.extend({
  lovesPellets: true
});
let SickPac = Ember.Object.extend({
  lovesPellets: false
})
```

As you can see, `Pac1` and `Pac2` take on the `lovesPellets: true` value from the `PelletFiend` mixin, while `SickPac` already has a value for the `lovesPellets` property, which takes precedence over the value given in the Mixin. Of course, since our example Mixin only has one property defined, it's a bit silly to mix it in to an Object that has already assigned a value to that property... but the vast majority of Mixins have multiple properties, some of which should not be overridden, and some of which specifically marked as changeable defaults as a way of changing how the Mixin is used.

Anyways, in our case we'll just be using the Mixin as a place to store properties and functions that are needed in both the Pac model and the pac-man component. Here's how we define our bare mixin:

```
//mixins/shared-stuff.js
import Ember from 'ember';

export default Ember.Mixin.extend({})
```

And here's how we import it into the pac-man component:

```
import Ember from 'ember';
import KeyboardShortcuts from 'ember-keyboard-shortcuts/mixins/component';
import SharedStuff from '../mixins/shared-stuff';
import Pac from '../models/pac';

export default Ember.Component.extend(KeyboardShortcuts, SharedStuff, {...})
```

If you're already a professional software engineer, you probably died a little inside at the Mixin name `SharedStuff`, but I assure you that the name is temporary. After the separation is complete, and as we add more complexity to the game, we'll create a more nuanced (and better-named) code-sharing story.

Naming aside, the important thing I wanted to show was us mixing our Mixin into the Component. It takes the same form as when we mixed in KeyboardShortcuts, but now instead of mixing in someone else's code from an addon we're mixing in our own code from within the project.

Classes and Instances

So far in this chapter we've covered how to split up the code and then bring it back together- split it up into Objects and Mixins, then bring it back together with the ES2015 module system (import/export statements). That's *almost* enough to be able to tackle the next chapter's refactor, but there's one more aspect of the object system that we'll have to tackle.

Let's say that we move the variable `x` from the pac-man component to the Pac model. We want to do that because `x` represents the x value of where the Pac is, so it makes sense to have it within the Pac object. However, that will cause us a problem- we need access to `x`! We need to figure out how the pac-man component will be able to access the variable `x` on the Pac.

We've imported `Pac` using the module system, but if we try to access the x value using `Pac.get('x')` (or any other direct method you can think of) it won't work. That's because the `Pac` that we imported is the Class definition. It is not a Pac object itself- it just tells us how to *create* Pac objects.

Here's how we can get access to that `x` value:

```
import Pac from '../models/pac';

export default Ember.Component.extend(KeyboardShortcuts, SharedStuff, {
  didInsertElement() {
    let pac = Pac.create()
    pac.get('x') //1
    this.set('pac', pac)
    this.movementLoop();
  },
  anotherFunction(){
    this.get('pac.x') //1
  }
})
```

Here's what we've done:

1. Created a Pac object, which is an *instance* of the Pac class.
2. Called `get` on that instance, which gave us the value of any property on the object. In this case, `x` (and the value returned is 1)
3. Put the instance onto the component scope so we can access from anywhere within the component
4. Accessed it from another part of the component (and used deep getting to directly reach the `x` value)

For those who prefer analogies, imagine that the `Pac` class definition we imported is a cookie cutter. When we call `create` with the class definition, it's like putting the cookie cutter down onto some flat-rolled cookie dough and creating a new cookie. The cookie is the instance, and the cookie cutter is the class definition.

Summary

In this chapter we learned about Ember's Object System, which bears a striking similarity to Ruby's Object System. Kind of crazy that a web framework is recreating parts of other languages, but hey-it's Javascript!

In the next chapter we'll use our new-found knowledge of Objects, Mixins, Classes, and ES2015 Modules in order to refactor our code into something much more manageable.

Loyalties

Your goal is to act the same, as if you hadn't heard anything last night… but even so, you still can't help but be more reticent when talking with Jerome. The feeling just isn't the same.

He's not the enemy, you tell yourself. He's not the enemy… just a spy.

"How was the training session last night," he asks.

In response, you shovel food into your mouth. It doesn't help.

"This is a tricky time during training. It may not feel like much is changing, but you're absorbing a lot of new patterns. You need to make sure you give it time to gel, have patience, even if it's tricky."

You recognize the words from last night. This talk been pre-rehearsed. The other ones, the ones that inspired you so much before- also pre-rehearsed. You still hold the idea that everything before this assignment had been genuine, but looking at Jerome talking he looks just as he did back then.

You get up and walk towards another table. On the way, you throw up.

You never make it to the other table- you end up retreating to the bathroom instead.

Jerome follows, and you feel like throwing up again.

"Fuck off," you say.

He backs up. "Bad food?" he asks nervously. You're slumped against the tile wall, but he's the one backing up and nervous.

"I know what you're doing," you say.

"I'm helping a friend, that's what I'm doing."

"Bullshit. I heard you talking to them last night. You're spying on me."

"It was me or someone else. I convinced them I would be more effective, since I already knew you."

"Congratulations. You're effective. Do you want me to be grateful?"

"I'm protecting you, Salam. Do you want someone else rifling through your books? Do you want someone else watching your lips when you pray? Getting you on video, trying to psychoanalyze?"

"What are you implying?"

"You're an apostate, Salam. You're an infidel."

"I'm observant."

“So am I, and I observed you in the lab, back when you were more careless. You’re not ‘Ahl Al-Kitab. You don’t get to be anything but what you were born.”

“They don’t care. I observe the forms, I don’t advertise. Besides, I’m too important.” You say it hopefully. With determination.

“Maybe now, but what about when they get their weapon? Do you think they will treat you as a dhimmi, when you were not born dhimmi? What about your parents? Your sisters? It won’t end well, Salam.”

Your heart sinks. You know he’s right.

Last year, JosÃ© had been found with an ancient tome from one of the natives- something Harris- that questioned the existence of Allah. He claimed it was a keepsake from his grandmother. But still, he didn’t come back to the lab. Your only message from him since then was a strangely worded video, a farewell, saying he was transferring to a secret lab.

You realize, with slowly growing horror, that there was no secret lab.

It’s time to compartmentalize. Make your ideas separate- completely private to outsiders.

You look Jerome in the eye and nod. “I’ll burn it.”

13: Separation of Concerns

In this chapter we're going to separate out a `Pac` object from the `pac-man` component.

We're doing this for several reasons. The first is that the `pac-man` component, currently at 200 lines, is just too damn long. The second is that having an object on which to store all the pac-related variables and methods just makes sense. We'll really be glad of this separation when we add in the complexity of ghosts (and extra lives).

Separation: Pac Object

The game plan is to separate out the `Pac`, put what needs to be shared into the `SharedStuff` mixin, and then edit various functions in the `pac-man` component so that they correctly reference values as being in the `Pac` instance.

We'll start with the `Pac`.

```
1  import Ember from 'ember';
2  import SharedStuff from '../mixins/shared-stuff';
3
4  export default Ember.Object.extend(SharedStuff, {
5    direction: 'down',
6    intent: 'down',
7
8    x: 1,
9    y: 2,
10
11   draw(){
12     let x = this.get('x');
13     let y = this.get('y');
14     let radiusDivisor = 2;
15     this.drawCircle(x, y, radiusDivisor, this.get('direction'));
16   },
17
18   changeDirection(){
19     let intent = this.get("intent")
20     if(this.pathBlockedInDirection(intent)){
21       this.set('direction', 'stopped');
22     } else {
```

```
23        this.set('direction', intent);
24      }
25    },
26
27    pathBlockedInDirection(direction) {
28      let cellTypeInDirection = this.cellTypeInDirection(direction);
29      return Ember.isEmpty(cellTypeInDirection) || cellTypeInDirection === 1;
30    },
31
32    cellTypeInDirection(direction) {
33      let nextX = this.nextCoordinate('x', direction);
34      let nextY = this.nextCoordinate('y', direction);
35
36      return this.get(`grid.${nextY}.${nextX}`);
37    },
38
39    nextCoordinate(coordinate, direction){
40      return this.get(coordinate) + this.get(`directions.${direction}.${coordinate\
41  }`);
42    },
43  })
```

This is our `Pac` object. Notice the variables that we've brought along: `direction`, `intent`, `x`, and `y`. Those are all variables that pertain directly to the state of the Pac (rather than the state of the rest of the world).

We've also brought along some functions. Take note of two of those: `draw` and `changeDirection`. The astute will notice that they used to be called `drawPac` and `changePacDirection`- having the class name in the function name is a dead giveaway that you should transfer it over to the object.

The bottom three functions were brought along because they were needed for `changeDirection`- `changeDirection` calls `pathBlockedInDirection`, which calls `cellTypeInDirection`, which calls `nextCoordinate`. If they had stayed in the pac-man component we wouldn't be able to easily access them from within the Pac.

The readers who are double-checking my explanations may have noticed a discrepancy in that last paragraph- `changeDirection` needs `pathBlockedInDirection` to be within the `Pac` scope, but then why is `draw` allowed to call `drawCircle`, which we haven't included in the `Pac` definition? For that matter, what about the `directions` hash and the `grid`, both of which are accessed in `changeDirection` dependencies? The answer lies in the `ShareStuff` mixin.

Separation: SharedStuff Mixin

The `SharedStuff` mixin is where we're putting all the stuff that's needed by both the `pac-man` component and the `Pac` object.

> Normally `SharedStuff` is a horrible generic name for a mixin, but in this case it fits, because the only thing holding these things together thematically is that they're stuff that is shared with both the component and the object. As the sharing gets refined, so will the name.

Here we'll find the `grid` property, the `directions` property, and the `drawCircle` function that we were 'missing' in the `Pac` definition, as well as the properties and functions required to support them.

```
1  import Ember from 'ember';
2
3  export default Ember.Mixin.create({
4    frameCycle: 1,
5    framesPerMovement: 30,
6
7    // 0 is a blank space
8    // 1 is a wall
9    // 2 is a pellet
10   grid: [
11     [2, 2, 2, 2, 2, 2, 2, 1],
12     [2, 1, 2, 1, 2, 2, 2, 1],
13     [2, 2, 1, 2, 2, 2, 2, 1],
14     [2, 2, 2, 2, 2, 2, 2, 1],
15     [2, 2, 2, 2, 2, 2, 2, 1],
16     [1, 2, 2, 2, 2, 2, 2, 1],
17   ],
18   squareSize: 40,
19
20   ctx: Ember.computed(function(){
21     let canvas = document.getElementById("myCanvas");
22     let ctx = canvas.getContext("2d");
23     return ctx;
24   }),
25
26   drawCircle(x, y, radiusDivisor, direction) {
27     let ctx = this.get('ctx')
28     let squareSize = this.get('squareSize');
29
```

```
30      let pixelX = (x + 1/2 + this.offsetFor('x', direction)) * squareSize;
31      let pixelY = (y + 1/2 + this.offsetFor('y', direction)) * squareSize;
32
33      ctx.fillStyle = '#000';
34      ctx.beginPath();
35      ctx.arc(pixelX, pixelY, squareSize/radiusDivisor, 0, Math.PI * 2, false);
36      ctx.closePath();
37      ctx.fill();
38    },
39    offsetFor(coordinate, direction){
40      let frameRatio = this.get('frameCycle') / this.get('framesPerMovement');
41      return this.get(`directions.${direction}.${coordinate}`) * frameRatio;
42    },
43
44    directions: {
45      'up': {x: 0, y: -1},
46      'down': {x: 0, y: 1},
47      'left': {x: -1, y: 0},
48      'right': {x: 1, y: 0},
49      'stopped': {x: 0, y: 0}
50    },
51  });
```

> Although Pac only directly calls two properties and a function, we've got six properties (one is a computed property) and two functions in the Mixin. If we were designing an API we might label the two properties and the function we call as Public, and the others as Private.

Separation: pac-man Component

Now we're going to dump the entire `pac-man` component in here. So sorry, but it's got to be done. I did show *some* mercy by compacting the functions that aren't changed with a `....`

Explanation to follow.

```
import Ember from 'ember';
import KeyboardShortcuts from 'ember-keyboard-shortcuts/mixins/component';
import SharedStuff from '../mixins/shared-stuff';
import Pac from '../models/pac';

export default Ember.Component.extend(KeyboardShortcuts, SharedStuff, {
  didInsertElement() {
    this.set('pac', Pac.create())
    this.movementLoop();
  },

  score: 0,
  levelNumber: 1,

  screenWidth: Ember.computed(function(){...}),
  screenHeight: Ember.computed(function(){...}),
  screenPixelWidth: Ember.computed(function(){...}),
  screenPixelHeight: Ember.computed(function() {...}),

  drawWall(x, y){...},
  drawGrid(){...},
  drawPellet(x, y){
    let radiusDivisor = 6;
    this.drawCircle(x, y, radiusDivisor, 'stopped');
  },

  clearScreen(){...},

  movementLoop(){
    if(this.get('pac.frameCycle') == this.get('pac.framesPerMovement')){
      let direction = this.get('pac.direction')
      this.set('pac.x', this.get('pac').nextCoordinate('x', direction));
      this.set('pac.y', this.get('pac').nextCoordinate('y', direction));

      this.set('pac.frameCycle', 1);

      this.processAnyPellets();

      this.get('pac').changeDirection();
    } else if(this.get('pac.direction') == 'stopped'){
      this.get('pac').changeDirection();
    } else {
```

```
43        this.incrementProperty('pac.frameCycle');
44      }
45
46      this.clearScreen();
47      this.drawGrid();
48      this.get('pac').draw();
49
50      Ember.run.later(this, this.movementLoop, 1000/60);
51    },
52
53    processAnyPellets(){
54      let x = this.get('pac.x');
55      let y = this.get('pac.y');
56      let grid = this.get('grid');
57
58      if(grid[y][x] == 2){
59        grid[y][x] = 0;
60        this.incrementProperty('score')
61
62        if(this.levelComplete()){
63          this.incrementProperty('levelNumber')
64          this.restartLevel()
65        }
66      }
67    },
68
69    levelComplete(){...},
70
71    restartLevel(){
72      this.set('pac.x', 0);
73      this.set('pac.y', 0);
74
75      let grid = this.get('grid');
76      grid.forEach((row, rowIndex)=>{
77        row.forEach((cell, columnIndex)=>{
78          if(cell == 0){
79            grid[rowIndex][columnIndex] = 2
80          }
81        })
82      })
83    },
84
```

```
85    keyboardShortcuts: {
86      up() { this.set('pac.intent', 'up');},
87      down()  { this.set('pac.intent', 'down');},
88      left() { this.set('pac.intent', 'left');},
89      right() { this.set('pac.intent', 'right');},
90    },
91  });
```

We'll start from the top.

First, in `didInsertElement`, we're creating an instance of `Pac` and setting it to `pac` on the component scope.

Next, in `drawPellet` we're accessing the `drawCircle` function, which has been moved to the `SharedStuff` mixin. Not a change, but just showing that we can still access. Same story for the `ctx` computed property that is in the `drawWall`, `drawGrid`, and `clearScreen` functions (functions definitions minimized to save room).

We'll skip `movementLoop` for now and go straight to `processAnyPellets`. Here we're accessing `pac.x` and `pac.y`- as you can see, `pac-man` can easily access properties that are in the `Pac` object when it needs to. In `restartLevel` and the various `keyboardShortcuts` methods we see that we can also *set* properties on the Pac object.

Now for `movementLoop`. First, we can see from `this.get('pac').nextCoordinate`, `this.get('pac').changeDirect` and `this.get('pac').draw()` that we can call functions that are on the pac instance. Second, we notice that the function is calling `pac` a lot- it seems as if the pac instance is referenced every other line.

In fact, if you count, you'll see that the `pac` instance is referenced *far more* than every other line. Like the word `Pac` in `movePac` and `changePacDirection`, this is a sign that the function may belong on the `Pac` class.

However, there is more nuance here- `drawGrid`, `clearScreen`, and `processAnyPellets` clearly don't belong on the `Pac`. So what do we do?

Separation: Partial method extraction

The answer is to separate out part of the loop and put it into a method on the Pac.

Here's what we left in the `pac-man` component:

```
loop(){
  this.get('pac').move();

  this.processAnyPellets();

  this.clearScreen();
  this.drawGrid();
  this.get('pac').draw();

  Ember.run.later(this, this.loop, 1000/60);
},
```

This code is much clearer!

There are two jobs be completed here: processing pellets and drawing stuff. The rest is left to the loop in the `Pac` object.

Most of our functionality is in the newly-created `move` function, but I've gone ahead and created two other named functions in order to clarify what we're doing in the loop:

```
//models/pac.js
move(){
  if(this.animationCompleted()){
    this.finalizeMove();
    this.changeDirection();
  } else if(this.get('direction') == 'stopped'){
    this.changeDirection();
  } else {
    this.incrementProperty('frameCycle');
  }
},

animationCompleted(){
  return this.get('frameCycle') == this.get('framesPerMovement');
},

finalizeMove(){
  let direction = this.get('direction');
  this.set('x', this.nextCoordinate('x', direction));
  this.set('y', this.nextCoordinate('y', direction));

  this.set('frameCycle', 1);
},
```

The addition of `animationCompleted` and `finalizeMove` make it clearer how `move` accomplishes the job of moving the `Pac`. There are several lower-level parts to that job, but they've been mostly farmed out to other functions: deciding if the animation is completed yet, finalizing the move, changing the `Pac`'s direction, and continuing the animation if in the middle of a move.

This is much clearer than our old version! It also demonstrates an important nuance in our refactoring practices- when you're separating out an object, the function isn't the smallest unit of separation. You may end up pulling apart a poorly-put-together function into several functions -across more than one object- that end up being much more cohesive and understandable.

> If you're following along with the github repository, I regret to inform you that they have now diverged. This is because I previously had a separate and independently-running `loop` on the Pac which was not only confusing, but was poorly-performing (especially after multiple game restarts). If there is a bug in the upcoming chapters, it's probably from the change. Please email those bugs to me at bilesjeffrey@gmail.com.

However, there is a problem arising from all this.

Resyncing on Restart

The problem surfaces when you restart the level while going left or up. Sometimes when you do that, the Pac will go outside the play area one square.

It does this because the animation cycle and the pellet-counting loops aren't synced up. Previously, pellet-counting happened in between finalizing the move and deciding upon the next direction. Now, the move is decide on in the `Pacs` `move` function, it will decide on the next direction *before* the pellets can be processed (and thus the level restarted). So it's already decided to keep going left or up when the level restarts (placing it at `[0, 0]`), and it keeps on moving left or up until the animation cycle is complete.

One way to prevent this is to move the `processAnyPellets` call into the `Pac`'s `move` function, but that would mean carrying the `processAnyPellets` function and all of its dependencies into the `Pac`. An argument could be made either way (it is the `Pac` that is processing the pellets), but the dependencies (which rely heavily on the level and current state of the grid) make the transfer shaky right now.

The way which we'll go about it is to simply add some further instructions to the `restartLevel` method.

```
restartLevel(){
  this.set('pac.x', 0);
  this.set('pac.y', 0);
  this.set('pac.frameCycle', 0);
  this.set('pac.direction', 'stopped')

  let grid = this.get('grid');
  grid.forEach((row, rowIndex)=>{
    row.forEach((cell, columnIndex)=>{
      if(cell == 0){
        grid[rowIndex][columnIndex] = 2
      }
    })
  })
},
```

The further instructions are to set `pac.frameCycle` to `0` and `pac.direction` to 'stopped'. That essentially resets the loop, forcing it to decide on its direction by using the new state of the level.

Summary

That was a lot of work, but I hope you'll agree that the cleaner, more understandable code makes it worth it. In the next chapter we'll continue refactoring the code by pulling out a `Level` class, and then we'll finally begin to see the fruits of our labor as we discover a way to create multiple levels and switch them in and out.

Separation

You don't actually burn it.

That would be messy. Incriminating. There's just no good way to discreetly light a manuscript on fire while in an underground bunker.

Instead, you ferry it to the latrine and let the water slowly soak away your fingerprints. You flush, to hurry the process along, then titrate in an acid. Someone will find it, but by then the words will be gone from the pages, and it will be untraceable to you.

As you walk away, you repeat key passages in your head, making sure that they hadn't been truly lost.

Still, when you get back to your bunk, you cry.

At breakfast the next day you avoid Jerome. Even with his reasons, you get a sick feeling when you see him.

You walk past him, and walk aimlessly among the tables.

"You, kid." You turn and see one of the beefy PAC men, one that had particularly frightened you when you first got here. He didn't look quite as intimidating now, since you had bulked up a bit from your training. He was also smiling. "Take a seat."

Another man laughs. "Someone's taking a step out his ivory tower."

"But he's still so serious," says another. "I don't think it wears off that quick."

You sit down slowly, carefully, watching their expressions as you do. Their eyes are encouraging, even if their words are mocking.

"Maybe he knows something about the aliens. Maybe that's why he hasn't joined us."

"What, like he knows which ones of us are gonna die, so he doesn't want to get to know us and get all sad when it happens?"

"No, I mean like… serious shit. Maybe instead of clowning with us, he's off thinking about strategy and saving the world and shit."

He gives you a significant look, obviously expecting you to say something. You're dumbfounded both by how they're talking about you when you're right there, making jokes at your expense, and by how it seems they've been wanting to talk to you his whole time. Was it really you who had been closed off?

“Maybe he’s a mute,” says another.

Yes, you had definitely been closed off. You stammer out something about the aliens, trying not to reveal too much. “The aliens are… yea, the aliens are a problem. And I’ve been missing my friends at the lab.”

They get serious all of a sudden. “I miss my wives and kids every day I’m out here,” says the big man. “Only thing that gets me through is my boys here.” He motions to the rest of the table.

Another man nods. “I got a wife and four kids at home. Last time I went back, my oldest was telling me the history of the PAC corps, asking when he could join me here.”

“I ain’t married like these guys,” says one of the younger men, “but I got people at the mosque that I miss.”

“Hey, you know what? We’re collecting pellets today, but join us at Isha’a tonight. A man should not pray alone.” The others agree, a chorus of encouragement.

The breakfast goes on, them going back and forth and you occasionally joining in. They’re not who you’re used to being around- they’re crude, loud, and nothing special in terms of IQ- but for the first time in weeks you can almost forget everything that’s happened.

You smile.

It doesn’t last.

Matteo is all business- clearly something is happening on the alien front, but he won’t tell you what.

“That’s not your concern. What is your concern is being able to move between levels, to adapt when given a new layout. Are you ready?”

You nod. Grit your teeth.

No trial shall overtake you beyond what you can bear.

14: Level Out

The last chapter started our quest to clean up the pac-man component by separating out coherent sections of code and making them their own objects. We pulled out the `Pac` object, as well as the `SharedStuff` mixin.

This chapter continues that quest, pulling out the `Level` object, making some more sense of what `SharedStuff` is, and then using our newly-separated Level objects to create easily-swappable alternate levels.

The Level Object

We'll start by showing the Level object that we've pulled out, then explain what went into the decisions made on what to pull and what not to pull, and then show what changes must be made elsewhere in order to keep things working.

```
1  //models/level.js
2  import Ember from 'ember';
3
4  export default Ember.Object.extend({
5    // 0 is a blank space
6    // 1 is a wall
7    // 2 is a pellet
8    grid: [
9      [2, 2, 2, 2, 2, 2, 2, 1],
10     [2, 1, 2, 1, 2, 2, 2, 1],
11     [2, 2, 1, 2, 2, 2, 2, 1],
12     [2, 2, 2, 2, 2, 2, 2, 1],
13     [2, 2, 2, 2, 2, 2, 2, 1],
14     [1, 2, 2, 2, 2, 2, 2, 1],
15   ],
16
17   squareSize: 40,
18   width: Ember.computed(function(){
19     return this.get('grid.firstObject.length')
20   }),
21   height: Ember.computed(function(){
22     return this.get('grid.length');
```

```
23    }),
24    pixelWidth: Ember.computed(function(){
25      return this.get('width') * this.get('squareSize');
26    }),
27    pixelHeight: Ember.computed(function() {
28      return this.get('height') * this.get('squareSize');
29    }),
30
31    isComplete(){
32      let hasPelletsLeft = false;
33      let grid = this.get('grid');
34
35      grid.forEach((row)=>{
36        row.forEach((cell)=>{
37          if(cell == 2){
38            hasPelletsLeft = true
39          }
40        })
41      })
42      return !hasPelletsLeft;
43    },
44  })
```

The first property we put into the level is the grid.

This is the most obvious choice; the grid is representation of the level's layout. If we changed it from `grid` to `layout` (so we'd be accessing `level.layout` instead of `level.grid`) it would make just as much sense. In fact, if were still storing the grid on the `pac-man` component, we wouldn't be remiss to simply name it `level`. So obviously that has to come with us.

Then we bring in the group of functions declaring the height, the width, and the pixel versions thereof. Coming with them is the `squareSize` property, which helps translate from the grid versions to the pixel versions of width and height. This transfer was also pretty obvious: we prefixed all of them with `screen`, but it would have worked just as well to prefix them with `level`. As it is now, when accessing one of these properties in the `pac-man` handlebars, it will be calling `level.pixelHeight` instead of `levelPixelHeight`.

Now, if I had had a master plan for all of this, I would have named stuff as `levelPixelHeight` and `layout` instead of the semi-synonymous `screenPixelHeight` and `grid`. But luck *did* strike with the name `levelComplete`. It obviously belongs on the `level` object (another sign: the only other thing it accesses is `grid`, which has now been moved to the `level` object). We've taken it over and renamed it `isComplete`.

Restarting It All

Going off that heuristic, there's one other function that you might think should be moved over: `restartLevel`. However, we can see that although the method name has "Level" in it, often references `pac`, which the `level` object does not have access to. In fact, if you look at the `restartLevel` function you'll notice it actually has two jobs:

```
//components/pac-man.js
restartLevel(){
  // restart Pac
  this.set('pac.x', 0);
  this.set('pac.y', 0);
  this.set('pac.frameCycle', 0);
  this.set('pac.direction', 'stopped');

  // restart Level
  let grid = this.get('level.grid');
  grid.forEach((row, rowIndex)=>{
    row.forEach((cell, columnIndex)=>{
      if(cell == 0){
        grid[rowIndex][columnIndex] = 2;
      }
    });
  });
},
```

The comments were added for this code sample, and represent the two jobs that `restartLevel` is doing.

When we first created this function it only had the second job- that of restarting the level. The first job, of restarting the Pac, was added on later.

Let's go ahead and refactor this by renaming it and then taking advantage of the fact that we have Level and Pac objects now.

```
//components/pac-man.js
restart(){
  this.get('pac').restart();
  this.get('level').restart();
}
```

```
//models/pac.js
restart(){
  this.set('x', 0);
  this.set('y', 0);
  this.set('frameCycle', 0);
  this.set('direction', 'stopped');
}
```

```
//models/level.js
restart(){
  let grid = this.get('grid');
  grid.forEach((row, rowIndex)=>{
    row.forEach((cell, columnIndex)=>{
      if(cell == 0){
        grid[rowIndex][columnIndex] = 2;
      }
    });
  });
}
```

There we go! Everything is properly descriptive, and each object has a way to restart itself.

Making it all fit

Of course, just separating it all out and naming things well won't be enough to make our app work again. We'll also have to edit some portions of the code so it's referencing the correct variables.

Start by replacing every referencing to `grid` with a reference to `level.grid` (except those that are within `level`, of course). Then do the similar replacements with `screenPixelWidth`, `screenPixelHeight`, and `levelComplete`.

When you're doing this, you may notice that the `Pac` accesses the grid... but doesn't have access to the level. We'll need to give it access to that.

```
//components/pac-man.js
didInsertElement() {
  let level = Level.create()
  this.set('level', level)
  let pac = Pac.create({level: level})
  this.set('pac', pac)
  this.loop();
},
```

Here we're creating a level in `didInsertElement` and setting it on the component scope. So far so good. Then, when creating the `Pac`, we add a hash that has only one key-value pair: the key being `level`, and the value being the level that we just created.

This adds that property to the `Pac` object- but only to *that instance* of the object that is currently being created. If you create another `Pac` object, it will be completely new and not know anything about the level that we're assigning here.

Another interesting property of the hash we're passing in is that it overrides any value that is defined on the class. To test that, try instantiating the level with `Level.create({squareSize: 20})`.

So now, if we've made all our moves correctly, the game should be back up and running again- just with much better code quality.

LessSharedStuff

Part of that code quality is that `SharedStuff` now has a tighter focus. If you look at it, there are two groups of code: drawing-related stuff and the directions hash.

It's tempting to want to separate the two groups, but then we'd just end up mixing back in the one with `directions` into the one with the drawing stuff, since `offsetFor` relies on those directions.

Here we're going to make a daring move and say "fuck it".

Yea, I know, programming books are supposed to downplay the shitty parts of their code unless there's an easy fix ready, poised to demonstrate some point, but that's not how real-life programming works. I mean, there's plenty of downplaying of shitty code in real life, but there's also a huge amount of code which is bad, and acknowledged to be bad, and is yet never fixed because fixing it isn't easy.

So, in recognition of reality, we're going to leave an ill-defined mixin with a vague name in our code.

> I considered renaming the mixin `mixins/shitty-name.js` or `mixins/fuck-it.js`, but no reason to keep reminding people of this.

Now, back on the the improvements!

Another Level

Because we've separated out our level into its own separate object, it's trivial to create a second distinct level on top of the first:

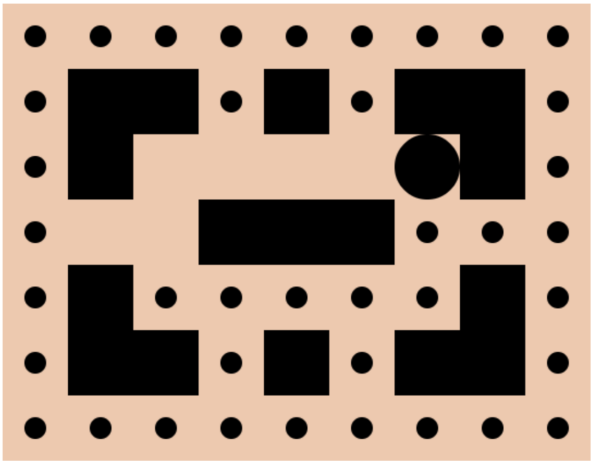
```
//models/level2.js
import Level from './level';

export default Level.extend({
  squareSize: 60,
  grid: [
    [2, 2, 2, 2, 2, 2, 2, 2, 2],
    [2, 1, 1, 2, 1, 2, 1, 1, 2],
    [2, 1, 2, 2, 2, 2, 2, 1, 2],
    [2, 2, 2, 1, 1, 1, 2, 2, 2],
    [2, 1, 2, 2 ,2, 2, 2, 1, 2],
    [2, 1, 1, 2, 1, 2, 1, 1, 2],
    [2, 2, 2, 2, 2, 2, 2, 2, 2],
  ]
})
```

We're importing the level, and then overriding the grid with a new grid of our choosing (and changing the `squareSize` too, just for kicks).

Then in the `pac-man` component we can just replace `Level` with `Level2`:

```
//components/pac-man.js
import Level2 from '../models/level2'

export default Ember.Component.extend(KeyboardShortcuts, SharedStuff, {
  didInsertElement() {
    let level = Level2.create()
    this.set('level', level)
    let pac = Pac.create({level: level})
    this.set('pac', pac)
    this.loop();
  },
  ...
})
```

Just import the new level and use it as the instantiating class and boom, the level is switched out. So awesome!



But there's one problem: our new arrangement has the Pac starting in a wall. Luckily it's already moving into an open square, but we might not always be so lucky. We need to specify the starting position of the Pac within the level.

The Level Gains More Regulatory Power

We'll start by creating a hash in the level called `startingPac`:

```
1  //models/level2.js
2  startingPac: {
3    x: 0,
4    y: 3
5  },
```

We'll then use those when instantiating the `Pac`, making sure it starts in the desired place:

```
1  //components/pac-man.js
2  didInsertElement() {
3    let level = Level2.create()
4    this.set('level', level)
5    let pac = Pac.create({
6      level: level,
7      x: level.get('startingPac.x'),
8      y: level.get('startingPac.y')
9    })
10   this.set('pac', pac)
11   this.loop();
12 },
```

Since we're overriding the starting values of `x` and `y` on the `Pac`, we might as well set them to `null`. This signifies to the user of the class that these values are meant to be passed in (we'll go ahead and add a null starting value for `level` while we're at it).

We'll also make sure to set the `x` and `y` values properly upon restart.

```
//models/pac.js
level: null,
x: null,
y: null,
restart(){
  this.set('x', this.get('level.startingPac.x'));
  this.set('y', this.get('level.startingPac.y'));
  this.set('frameCycle', 0);
  this.set('direction', 'stopped')
},
```

And that gets us to always having the correct starting position, both when the game first starts and when the game restarts after a win. If you plan on going back to the original level, be sure to give it a `startingPac` hash. I used the following:

```
//models/level.js
startingPac: {
  x: 2,
  y: 1
},
```

Summary

In this chapter we continued separating out our various concerns- this time by separating out our Level. In addition to our code being better organized and easier to read, we finally started seeing the other benefits of separating out different classes when we were able to quickly create a second level. We also got a horrifying dose of reality that it's best to forget.

In the next chapter we'll finally add in our arch-enemies the ghosts- and then we'll really be glad we've separated stuff out!

First Haunting

It was a hard day's training- Matteo had switched out the layout of the maze, shaking up your mental map and making you generalize your reflexes. Some things you had learned as useful on the last course were harmful if done unthinking on this one, so a situational awareness began to take hold. In the middle of the day he switched it again, and by the end you had a rough idea of the tradeoffs of the different techniques you had learned in different situations.

But despite the difficulty, despite still feeling the sting of the situation with Jerome, despite having to flush one of your most precious possessions, you were able to get through it knowing that you had new friends to see tonight.

You enter the prayer room and slowly pick out your friends from breakfast. You probably wouldn't be able to recognize any of them individually except for the big man, but together they were distinctive.

The group is unusually serious- whispering, huddled. You approach cautiously, but when they see you their entire demeanor changes. Not quite the boisterousness of their cafeteria behavior, but still jovial and welcoming.

"Good day?" you ask.

"Long day," says one of them.

"Weird day," says another.

"No need to worry our new friend about that, eh?" says the big man. "Plenty of other stuff to talk about."

The others look around nervously, unsure. Something had shaken them. "You see a Ghost or something?" you ask.

"We don't know exactly… it was… something. Something like a PAC, but with these protrusions. These…"

"Tentacles. It was like tentacles."

"Tentacles?"

"Yea, these wiggly limbs that some ancient water animals had."

"Anyways, we got a glimpse of it. We didn't get too much video, because we were focused on getting the PAC out of harm's way, but Headquarters is analyzing what we have."

The Imam called the prayer meeting to order, bringing the conversation to a halt. You go through the prayers, feeling good to have brothers beside you once again. In the silences, the gaps, you think your own prayers- more personal ones.

It's all going well… but then there's a rumbling. The Imam talks louder, trying to think it out of existence, but the rumbling increases in volume even more, becoming difficult to ignore.

If the rumbling is difficult to ignore, the screaming is impossible.

Jerome bursts through the door to the prayer room, something that in any other situation would be a near-unforgivable breaking of etiquette. "Salam!" By the time you turn around he's grabbed you and is rushing you away.

You're stuffed into your PAC, this time loaded with scientific instruments. "It's imperative that these objects stay functional," says Jerome. "You're the only one who understands the science, but I know you'll need this equipment to succeed."

Jerome gets into a nearby PAC, and you hear him over the radio. "Let's go."

"Where?"

"Down."

15: Ghostly

The ghosts have finally arrived! They're here to stop your pellet-gathering activities once and for all! Also to make the game more challenging.

The first step is to create the ghost itself.

The Ghost Itself

We'll create a `Ghost` class, then include all the drawing stuff needed with the `SharedStuff` mixin.

```
//models/ghost.js
import Ember from 'ember';
import SharedStuff from '../mixins/shared-stuff';

export default Ember.Object.extend(SharedStuff, {
  draw(){
    let x = this.get('x');
    let y = this.get('y');
    let radiusDivisor = 2;
    this.drawCircle(x, y, radiusDivisor, this.get('direction'));
  },
})
```

The only thing we're defining right now is the `draw` function, which is just like the `Pac` draw function.

Next we'll need to get a `Ghost` into our app. We'll start by adding to the mass of instantiations in `didInsertElement`:

```
//components/pac-man.js
import Ghost from '../models/ghost';

didInsertElement() {
  //...
  let ghost = Ghost.create({
    level: level,
    x: 0,
    y: 0
```

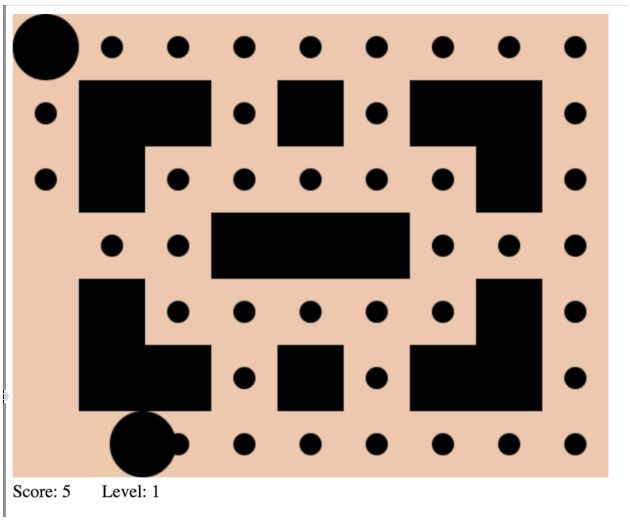
```
10    });
11    this.set('ghost', ghost)
12    //...
13  },
```

We're setting the `x` and `y` values to `0` by default right now, although that will change later.

Then during the loop we'll add a call to the ghost's `draw` function:

```
1   //components/pac-man.js
2   loop(){
3     this.get('pac').move();
4
5     this.processAnyPellets();
6
7     this.clearScreen();
8     this.drawGrid();
9     this.get('pac').draw();
10    this.get('ghost').draw();
11
12    Ember.run.later(this, this.loop, 1000/60);
13  },
```

All this code will result in a "`Ghost`" showing up in the upper left hand corner of the screen.



However, right now the "Ghost" just sits there. In order to make it recognizable as a ghost we'll need to add:

- Color Differentiation

- Movement
- Chasing Behavior

That's what the rest of this chapter will be about.

True Colors

First up will be color differentiation. The first change will be to our shared `drawCircle` function, addig an optional 5th argument in order to accommodate further flexibility:

```
1  //mixins/shared-stuff.js
2  drawCircle(x, y, radiusDivisor, direction, color = '#000') {
3    //...
4    ctx.fillStyle = color;
5    //...
6  },
```

So the *optional* 5th argument has a *default* value of `#000` (black). If no 5th argument is passed in, it will be black. If a 5th argument is passed in, then it will have that as the color (overriding the default).

Here's our revamped draw function for the `ghost`, with the new 5th argument added in.

```
1  //models/ghost.js
2  draw(){
3    let x = this.get('x');
4    let y = this.get('y');
5    let radiusDivisor = 2;
6    this.drawCircle(x, y, radiusDivisor, this.get('direction'), '#F55');
7  },
```

If you want to, this would be a good time to change our `Pac` to the yellower color of `#FE0`.

Notice that the pellets still call `drawCircle` with only `4` arguments, and they are still black. That means everything is working as expected.

Movement: It's alive!

We need to make our `Ghost` move. Luckily, we already have a circle that moves- the `Pac`. We're going to extract a bunch of stuff from the `Pac` into a `Movement` mixin:

```
//mixins/movement.js
import Ember from 'ember';

export default Ember.Mixin.create({
  x: null,
  y: null,
  level: null,
  direction: 'down',

  move(){...},
  animationCompleted(){...},
  finalizeMove(){...},
  pathBlockedInDirection(direction) {...},
  cellTypeInDirection(direction) {...},
  nextCoordinate(coordinate, direction){...},
})
```

Everything we've copied over has been copied over verbatim. Instead of going over these one by one, it may be more instructive to look over what we *haven't* copied over:

```
//models/pac.js
import Ember from 'ember';
import SharedStuff from '../mixins/shared-stuff';
import Movement from '../mixins/movement';

export default Ember.Object.extend(SharedStuff, Movement, {
  direction: 'down',
  intent: 'down',

  restart(){...},
  draw(){...},
  changeDirection(){
    let intent = this.get("intent")
    if(this.pathBlockedInDirection(intent)){
      this.set('direction', 'stopped');
    } else {
      this.set('direction', intent);
    }
  },
})
```

We've left `intent`, `restart`, `draw`, and `changeDirection`. We also have `direction`, but that's overriding the `direction` we put into `Movement`.

I left `changeDirection` expanded so that you could be reminded of the decision-making algorithm used for the `Pac`. That's going to be the main difference between the `Pac` and the `Ghosts`.

So we'll mix in `Movement` to the `Ghost`, and add in a blank `changeDirection` so that our `move` function won't throw an error:

```
//models/ghost.js
import Ember from 'ember';
import SharedStuff from '../mixins/shared-stuff';
import Movement from '../mixins/movement';

export default Ember.Object.extend(SharedStuff, Movement, {
  draw(){...}
  changeDirection(){
    // We'll decide later
  }
})
```

The only thing left to do now is to add the movement to the loop.

```
//components/pac-man.js
import Ghost from '../models/ghost';

loop(){
  this.get('pac').move();
  this.get('ghost').move();

  this.processAnyPellets();

  this.clearScreen();
  this.drawGrid();
  this.get('pac').draw();
  this.get('ghost').draw();

  Ember.run.later(this, this.loop, 1000/60);
},
```

It's moving! Hurrah!

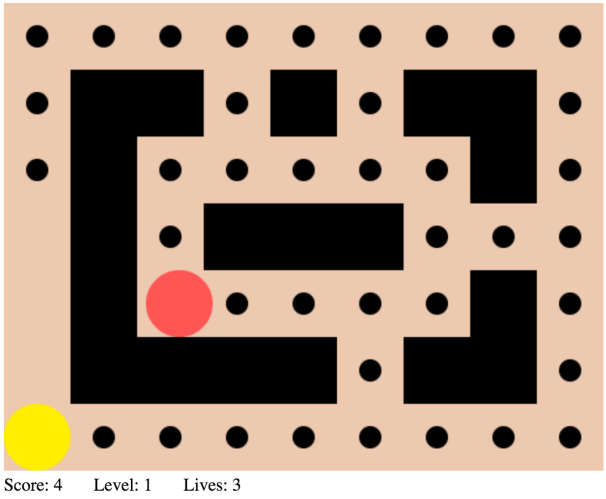
Unfortunately, it just moves straight down then falls off the screen. Clearly our Ghost is going to need some better decision-making skills if it wants to catch our heroes.

Heat-seeking Ghosts

This is where we define `changeDirection` for our Ghost. Unfortunately, it's not as easy as just taking directions from the user. We'll program a *very* primitive AI, one which combines heat-seeking capabilities with a bit of randomness.

The heat-seeking simply looks to where the Pac is and tries to make the Ghost move in that direction- either by moving closer horizontally or vertically, whichever is more convenient.

The randomness is there for two reasons. The first is so it's harder to get a ghost stuck in a corner, as it often would in a purely deterministic heat-seeking algorithm.



In the above situation, a purely deterministic ghost will never reach the PAC. While that may make for some interesting puzzle situations (see the eventual sequel to this book, EDIT PACMAN (2017?)), it's not so great for gameplay. Hence, the randomness.

The second reason for the randomness is so that when we add multiple ghosts, they don't converge into one stack- once they get on the same square once, they would by definition always move in the same "best" direction and be, for all intents and purposes, one ghost. So, once again, not great for gameplay. Hence, the randomness.

Alright, so let's build this thing:

```
changeDirection(){
  let directions = ['left', 'right', 'up', 'down']
  let directionWeights = directions.map((direction)=>{
    return this.chanceOfPacmanIfInDirection(direction);
  })

  let bestDirection = this.getRandomItem(directions, directionWeights);
  this.set('direction', bestDirection)
},
```

As we go over `changeDirection`, there are three things here that you likely don't recognize:

1. The `map` method (built into Javascript).
2. The `chanceOfPacmanIfInDirection` function.
3. The `getRandomItem` function.

Don't worry, we'll explain them all- starting with the common and incredibly useful Javascript function `map`.

Map

`map` is a sibling method to `forEach`, which we've used before. They both work on arrays, and do something with each member of the array. The difference is that while `forEach` is meant to perform an action on each item in an array (such as drawing a wall forEach cell that equals 1), `map` is meant to *return* a new *transformed* array, and the function inside `map` is what does the mapping of the source array to the new array.

Let's take a simple example:

```
let fruits = ['cherry', 'strawberry', 'orange']
let yummyFruits = fruits.map((fruit)=>{
  return `yummy ${fruit}`;
});
console.log(yummyFruits) //['yummy cherry', 'yummy strawberry', 'yummy orange']
```

`map` is called on the `fruits` array and takes a function. The first argument of that function is an item in the array (the second item is the index, but we're not using that here). We then prepend the word "yummy" to the fruit, and *return* that new value.

We'll see that usage paralleled in our `changeDirection` method:

```
let directions = ['left', 'right', 'up', 'down']
let directionWeights = directions.map((direction)=>{
  return this.chanceOfPacmanIfInDirection(direction);
})
```

In this case, instead of just prepending the word 'yummy', we're calling out to a method that calculates the chance that the Ghost will find the Pac by going in that direction.

Let's look at that method next.

Where's the Pac?

As we'll see soon, the name `chanceOfPacmanIfInDirection` isn't *precisely* correct, but it's close enough. Let's dig in.

```
chanceOfPacmanIfInDirection(direction) {
  if(this.pathBlockedInDirection(direction)){
    return 0;
  } else {
    let chances = ((this.get('pac.y') - this.get('y')) * this.get(`directions.${\
direction}.y`)) +
                  ((this.get('pac.x') - this.get('x')) * this.get(`directions.${\
direction}.x`))
    return Math.max(chances, 0) + 0.2
  }
},
```

First, we'll check to see if the path is blocked in that direction. If it is, we return a `0`. Otherwise, we do a more complicated calculation.

The first line of the `chances` calculation is as follows:

```
(this.get('pac.y') - this.get('y')) * this.get(`directions.${direction}.y`)
```

The first part is getting the difference between the `y` value of the ghost and the `y` value of the Pac- that can be either positive or negative (or `0`, if they're on the same row). Then we multiply it by that direction's `y` value. So 'up' would end up being `-1` and 'right' would end up being `0`.

So 'right' and 'left' automatically end up as a big `0`, no matter the different in `y` values. Then for 'up' and 'down' it's -1 or 1. That sign is extremely important- if it's the same sign as the difference between the pac and the ghost's `y` values, then `chances` will end up positive.

Or, put another way, if the Pac is in that direction then the result will be positive- and the farther away it is in that direction, the larger the positive result it will be. If the Pac is in the opposite direction, the result will be negative.

But it's not just the `y` value- we add to that result a similar operation for `x` values:

```
let chances = ((this.get('pac.y') - this.get('y')) * this.get(`directions.${dire\
ction}.y`)) +
              ((this.get('pac.x') - this.get('x')) * this.get(`directions.${dire\
ction}.x`))
```

That second line, with the `x` values, does for 'left' and 'right' what the first line did for 'up' and 'down'.

So the `chances` will end up with some sort of value, usually positive or negative. Then we round up all negative value and add a bit of extra randomness:

```
return Math.max(chances, 0) + 0.2
```

The `Math.max` call finds the larger of `chances` and `0`, essentially rounding up all negative values to `0`. The addition of `0.2` makes it possible that the Ghost will sometimes go the opposite direction of the Pac... although not very often. Changing this variable can change how the game plays.

So that's how we calculate the "chances" of the Pac being in any one direction.

The (Weighted Random) Chase

Next we'll go back and reorient ourselves in our larger `changeDirection` function:

```
changeDirection(){
  let directions = ['left', 'right', 'up', 'down']
  let directionWeights = directions.map((direction)=>{
    return this.chanceOfPacmanIfInDirection(direction);
  })

  let bestDirection = this.getRandomItem(directions, directionWeights);
  this.set('direction', bestDirection)
},
```

We just got `directionWeights` from the `map`, and it's an array such as `[0, 3.2, 0.2, 0.2]`. That means that there's no chance of going left (probably a wall there), a large chance of going up, and a small chance of going down or right. This is called "weighted randomness".

Now we just need a function that will take this array and choose a direction randomly, but preferencing the directions with larger weights. That function is `getRandomItem`:

> Note that we somewhat disingenuously send the result of `getRandomItem` to `bestDirection`. Is it really the "best" direction, if we selected it semi-randomly? Consider it our `probablyPrettyGoodDirection`.

```
getRandomItem(list, weight) {
  var total_weight = weight.reduce(function (prev, cur, i, arr) {
      return prev + cur;
  });

  var random_num = Math.random() * total_weight;
  var weight_sum = 0;

  for (var i = 0; i < list.length; i++) {
      weight_sum += weight[i];
      weight_sum = Number(weight_sum.toFixed(2));

      if (random_num < weight_sum) {
          return list[i];
      }
  }
},
```

Holy mother of Pac, that is a lot of things we haven't seen before. And some of them (like `reduce`) are advanced topics. I'm declaring that understanding this function is out of scope of this book- just copy and paste it into your code and know that it does weighted randomness.

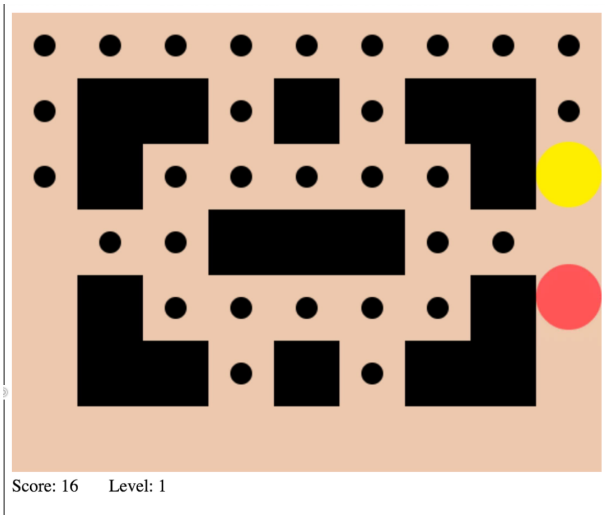
So the function takes in the directions and the array of weights, and it spits out a string of a direction. That string is set as the new direction.

And there we go- we have an algorithm for going after the Pac. But before it can do that, we need to do one last thing: feed it a version of the Pac.

```
//components/pac-man.js
import Ghost from '../models/ghost';
import Pac from '../models/pac';

didInsertElement() {
  //...
  let pac = Pac.create({...})
  //...
  let ghost = Ghost.create({
    level: level,
    x: 0,
    y: 0,
    pac: pac
  });
  this.set('ghost', ghost)
```

```
16    //...
17  },
```

This completes our work on Ghost AI, but there's plenty of room for improvement here. In the original PacMan, each of the 4 ghosts had its own AI "personality" that made it behave slightly different than the other ghosts. That would be a cool PR to make to the addon[12]!



Summary

In this chapter we created a `Ghost` class and instantiated it into our component. To do this, we pulled out a bunch of code from the `Pac` class and put it into the `Movement` mixin, which we then include in both. We then created an algorithm to make our `Ghost` seek out the `Pac`, providing for some much more exciting gameplay.

Along the way we learned about the `map` function, which is vital for working effectively with arrays.

In the next chapter, we'll add in consequences for running into a ghost- collision detection to figure out whether we're hit, a restart function for when we get hit, and a lives counter to remind us of our mortality. In the chapter after that we'll increase the number of ghosts and make it easy to set their starting places in a level.

[12] https://github.com/jeffreybiles/pac-man

Sacrifice

The Ghost bears down on you, a ball of malevolent fury. Tentacles slapping against the walls, catapulting it forward and violently steering it through corners. You stand aghast for a moment until your instincts kick in. There's no playing dead, no fighting it. Flight is the only thing left.

Your PAC moves with equal agility, thankfully not registering the tremor in your hands. Your entire body.

"Left," says Jerome over your headset. You turn left at the next opportunity.

"Straight. Now right. Straight again. Straight. Right. Straight. Left. Shit, no, I mean straight. Oh shit."

It's too late. You've already turned left, and soon you join him in his cursing- it's a completely dead end. You look behind and see the ghost coming around the corner, it's violent whipping coming to a stop. It knows it has you. Tentacles wave, adrift, as gears turn. There's talk over your headset, but you can't hear it over the low moan coming from the Ghost. The moaning stops, and the gears begin to turn with greater speed.

"I've got it," says a voice. It's the big man. You just met him yesterday, but you already consider him one of your closest friends here.

The gears of the Ghost open, revealing something that looks… human.

"Allahu Akbar!" yells the big man as his PAC crashes into the Ghost. The momentum shoves both machines out of your way, and a frantic Jerome is yelling at you to take advantage of this and move.

The human figure from the Ghost is slumped against the wall, mouthing something. You think you hear your name from somewhere, but that can't be right.

The Ghost itself has gone into self-defense mode. Two tentacles are crushed, sparking uselessly. The ones that still work are tearing apart the big man's PAC.

"Move!" yells Jerome. "Come on!"

You know enough about PAC architecture to know how close the tentacles are to flesh.

They're close.

They're there.

You wonder how long you'll remember this image.

16: Contact

In our last chapter we created a ghost that chased the Pac, but when it caught the Pac, nothing happened. That's not a fun game! Failure must have consequences!

And in this chapter, consequences will prevail! But first, let's up the challenge by increasing the number of ghosts.

Multiple Ghosts

Adding the second ghost is remarkably easy. We just copy the code that creates our first ghost, and then change the value we assign it to:

```
// in didInsertElement in /components/pac-man.js
let ghost1 = Ghost.create({
  level: level,
  x: 0,
  y: 0,
  pac: pac
});
let ghost2 = Ghost.create({
  level: level,
  x: 5,
  y: 0,
  pac: pac
})
```

We have put it 5 spaces to the right so that we can see it as a separate Ghost right at the starting gate.

The we need to make sure that the loop makes *both* Ghosts moves. We'll do that by first assigning them to the `ghosts` array, then looping over that array.

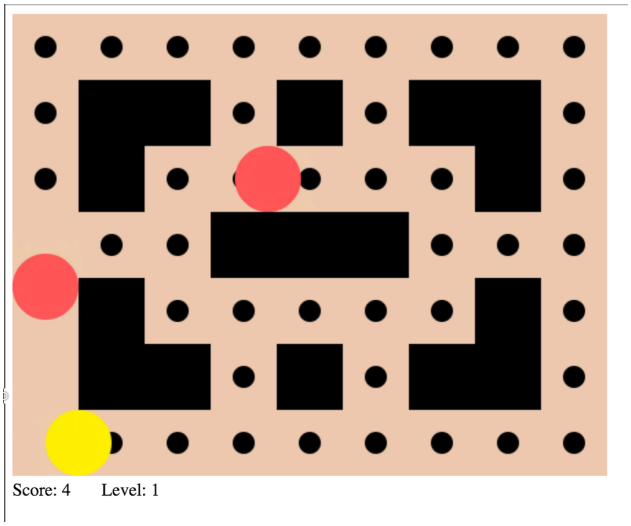
```
// in /components/pac-man.js

didInsertElement(){
  //...
  let ghosts = [ghost1, ghost2]
  this.set('ghosts', ghosts)
  //...
}

loop(){
  this.get('ghosts').forEach((ghost) => { ghost.move() } );
  //...
}
```

This may seem like extra work right now- why not just set them separately and then call `move` on both of them by hand?- but it will pay off later when we have more ghosts and more methods we want to call on all of them.

Later on, we'll add in the following code in the `draw` method:

```
// in didInsertElement in /components/pac-man.js
this.get('ghosts').forEach((ghost)=> { ghost.draw() } )
```

With this, we'll be looping and drawing both ghosts. If we wanted to add a third, we'd only need to create a third Ghost in `didInsertElement` and then include that in the `ghosts` array. All the other code would stay the same.



Before we move on, I wanted to introduce some nuances and shortcuts to our arrow function syntax.

Better Arrow Functions

In the last two code samples, we've been using the arrow function as follows:

```
(ghost)=> { ghost.draw() }
```

However, there are some shortcuts we can take. First, if there is only one argument, then the parentheses are not needed. Second, if there is only one expression in the function, the curly braces are not needed. The previous code could look as follows:

```
ghost => ghost.draw()
```

Notice how much cleaner it is! And no relevant information was lost. Here it is within the context of the code:

```
//in `loop`
this.get('ghosts').forEach( ghost => ghost.move() );
//...
this.get('ghosts').forEach( ghost => ghost.draw() );
```

One thing to keep in mind when you're using this is that if you add a second expression (let's say you want to call `ghost.draw()` and the hypothetical method `ghost.sound()`), then you need to add back the curly braces. And if you add a second argument, you need to add back the parentheses. But for situations like this, this shortcut makes our code better!

Now, back to building the game! Next we'll put in the code that can tell whether we've run into a Ghost.

Collision Detection

We're first going to create a function that can tell whether a Ghost has run into our Pac. We'll do that by checking to see if they share the same `x` and `y` values (i.e., that they're in the same square).

```
collidedWithGhost(){
  return this.get('ghosts').any((ghost)=>{
    return this.get('pac.x') == ghost.get('x') &&
           this.get('pac.y') == ghost.get('y')
  })
},
```

So it's getting the `ghosts` array, and then it calls the *any* method, which we've used before. It runs a function for each item in the array, and if any of them return true, then the `any` function returns true. In this case, we're just checking whether the `x` and `y` values of the ghost in question matches with the `x` and `y` values our Pac.

Next, we'll call that within our loop, and restart everything if true.

```
loop(){
  this.get('pac').move();
  this.get('ghosts').forEach( ghost => ghost.move() );

  this.processAnyPellets();

  this.clearScreen();
  this.drawGrid();
  this.get('pac').draw();
  this.get('ghosts').forEach( ghost => ghost.draw() );

  if(this.collidedWithGhost()){
    this.restart();
  }

  Ember.run.later(this, this.loop, 1000/60);
},
```

Then we'll need to edit our `restart` function to include our Ghosts.

```
restart(){
  this.get('pac').restart();
  this.get('level').restart();
  this.get('ghosts').forEach( ghost => ghost.restart() );
},
```

If you run the code now, you'll see that it's all working great- the pellets get put back into position, the Pac goes to where it originally was placed, and the Ghosts… well, they both go into the upper-left corner, on top of one another.

That's because although we set the positions of the Ghosts differently when we create them, resetting them puts them at `[0, 0]`. What we need is a default starting position for the ghosts defined on the level, just like we have defined the starting place of the Pac on the level.

Places, Please

We'll start off with making just a small change that accomplishes our goal for this specific level. Then we'll generify the fix so that levels are once again interchangeable.

The initial change takes place within the `Ghost` class:

```
1  //models/ghost.js
2  init() {
3    this.set('startingX', this.get('x'));
4    this.set('startingY', this.get('y'));
5    return this._super(...arguments);
6  },
7  restart(){
8    this.set('x', this.get('startingX'));
9    this.set('y', this.get('startingY'));
10   this.set('frameCycle', 0);
11   this.set('direction', 'stopped')
12 },
```

First we have our `init` function. It's similar to `didInsertElement` in that it's run at the beginning of an object's lifecycle, but while `didInsertElement` can only be used on Components and other things that inherit from View, `init` can be used on anything that inherits from Ember Object (that is, most things).

> While `didInsertElement` is run when the element is inserted into the DOM, `init` is run when the object is initialized. Components are inserted into the DOM, but most objects are not- hence them not being able to use `didInsertElement`. Although our Objects have a `draw` function, that's painting a representation on the Canvas, not inserting an element into the DOM.

So we set new values `startingX` and `startingY` within the `init` function, using the values that we give it for `x` and `y`. When we call `restart`, it uses those values to reset `x` and `y`.

The third line in `init` may puzzle you at first- we'll explain why we do it in this optional next section.

Super Spread(optional)

> This next part is more advanced, and not necessary to understanding the rest of the book

When you instantiate an object, `init` is run. Any object in the object chain can define `init`. This can be a problem because a definition that is lower on the inheritance chain will override a definition that is higher up. If we defined `init`, and something above is also defined `init` and did important stuff there, we're going to break stuff with our `init` definition.

That's where `_super()` comes in. It sends the command up the chain, saying "Even though I defined `init` down here, I want to run any `init`s that are above me in the inheritance chain". Then it feeds it in `...arguments`.

We've previously used `...` to signal "we're leaving out part of the code", like it's used in normal English. However, here it has a very specific meaning. It's the "spread" operator, which takes an array and spreads it out.

```
let nums = [1, 2, 3];
console.log([nums, 4, 5]) //[[1, 2, 3], 4, 5]
console.log([...nums, 4, 5]) //[1, 2, 3, 4, 5]

let arguments = ['firstArg', 'secondArg']
this._super(arguments) //this._super(['firstArg', 'secondArg'])
this._super(...arguments) //this._super('firstArg', 'secondArg')
```

So instead of using the array, it using each separate item of the array that is being spread.

In this case, `arguments` is a javascript Array that is automatically created for each function that contains every single argument given to the function when it is being called.

> That's it for the optional part. It was short, advanced, and maybe you didn't understand it all. That's okay! Now back to our regularly scheduled programming.

Level Knows Best

Now we want to make it so that we can have different Ghost starting positions on different levels.

Here's how we define it in level2:

```
//models/level2
startingPac: {
  x: 0,
  y: 3
},
startingGhosts: [{
  x: 0,
  y: 0
}, {
  x: 5,
  y: 0
}],
```

I've left the `startingPac` in there for reference, so you can see that we're basically doing the same thing, just sticking multiple of them in an array.

Then we'll want to create those Ghosts in `didInsertElement`:

```
// components/pac-man.js
didInsertElement(){
  //...
  let ghosts = level.get('startingGhosts').map((startingPosition)=>{
    return Ghost.create({
      level: level,
      x: startingPosition.x,
      y: startingPosition.y,
      pac: pac
    })
  })
  this.set('ghosts', ghosts)
  //...
}
```

Notice that because we're using `map` on the array, we can create as many Ghosts as the level specifies.

And then... that's it! Before we move on to giving our Pac multiple lives, make sure to add Ghost starting positions to level1 and any other levels you've made.

Extra Lives

The final part of this chapter is the simplest; a nice victory lap for all your hard work.

We'll start off by adding the `lives` property to the `pac-man` component:

```
//components/pac-man.js
lives: 3
```

Then we'll show it in the handlebars for that component:

```
<!-- templates/components/pac-man.hbs -->
<canvas id="myCanvas" width={{level.pixelWidth}} height={{level.pixelHeight}}></\
canvas>
<br>
Score: {{score}}       Level: {{levelNumber}}      \
; Lives: {{lives}}
```

That will get the levels showing up below our level. Now we need for the Pac to lose a life when it hits a ghost:

```
//components/pac-man.js
loop(){
  //...
  if(this.collidedWithGhost()){
    this.decrementProperty('lives');
    this.restart();
  }

  Ember.run.later(this, this.loop, 1000/60);
},
```

So every time we collide with a Ghost, we lose a life before restarting the level.

What happens when we get to 0 lives?

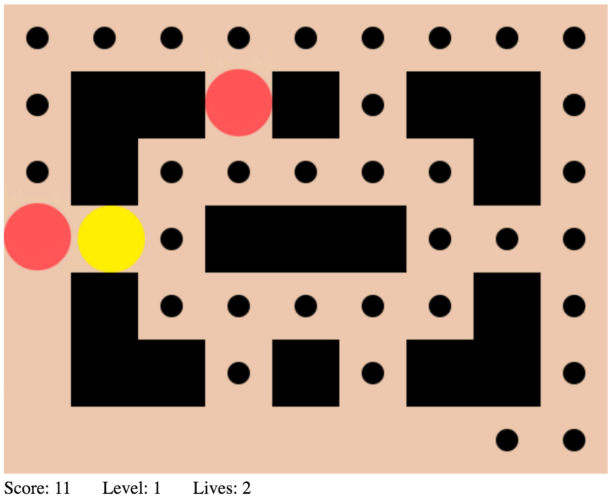
```
//components/pac-man.js
restart(){
  if(this.get('lives') <= 0) {
    this.set('score', 0)
    this.set('lives', 3)
    this.get('level').restart();
  }
  this.get('pac').restart();
  this.get('ghosts').forEach( ghost => ghost.restart() );
},
```

We added the `if` statement which only triggers when the player is out of lives, and then we reset the score and number of lives. We also moved the level restart call in there, so that it won't restart the level when you die *unless* you're out of lives.

This does mean that we'll have to specifically reset the level when we want to restart stuff due to the level being completed:

```
//components/pac-man.js
processAnyPellets(){
  let x = this.get('pac.x');
  let y = this.get('pac.y');
  let grid = this.get('level.grid');

  if(grid[y][x] == 2){
    grid[y][x] = 0;
    this.incrementProperty('score')

    if(this.get('level').isComplete()){
      this.incrementProperty('levelNumber')
      this.get('level').restart();
      this.restart()
    }
  }
},
```

So that finishes out our extra lives feature.



Summary

In this chapter we accomplished a lot: we added multiple ghosts, made the ghosts dangerous, gave our Pac multiple lives, and allowed each level to determine how many Ghosts to have and where they should start. We also learned a new shorter version of the arrow function syntax, as well as some optional stuff about `_super` and the spread operator.

In the next chapter we're going to make it so that winning a level moves you to a different level. Your players are gonna be drowning in variety!

Ruins

It's only hours after the escape that Jerome lets you rest. It's past midnight, and the adrenaline is starting to wear off.

"Thirty minutes. That's all. Then we move."

You get out of your PAC, grab a pellet from its stores, and put it in the processor. It whirs, chemical reactions and carefully calibrated heat and pressure making the pellet into something edible.

You take a moment to look around and notice strange patterns on the tunnels.

"What is this place?"

"A strange city from the old world. It was named after an ʿAhl al-Kitab holy man who fought for the poor, but was taken over by a series of cults who practiced wild and licentious living. They were useful in their time, but in the end they would not accept the Caliphate, so we had to destroy them."

"Killed them all?"

Jerome shook his head. "No. There were some that would submit. And the ʿAhl al-Kitab, rest assured, were allowed to live as dhimmi. But the cults were pig-headed, vile- some were dedicated to gaining wealth and power without measure, making themselves gods on earth while the common people starved. One of their sects flew away into the sky before we could stop them, but we destroyed the rest. Then there was another dedicated to the abominations of Sodom and Gomorrah. It was pleasure to rain down on them brimstones hard as baked clay, spread, layer on layer."

You had known vague outlines of this story, but hearing the history so bluntly makes you sick to your stomach. Clearly your ancestors were one of those who converted and were spared (it was only later that they became apostate), but that didn't make it any easier to take. This had once been your civilization.

But, as your grandfather had said, every great empire was built upon atrocities. The Caliphate had destroyed this civilization but left over half of its people alive. This dead civilization had nearly wiped out the entire populace of those who had been there before them. And before that, even at the beginning, a species called the neanderthals.

The processor dings, and Jerome hands you your portion of the pellet. You're not sure you're hungry anymore, but you eat anyways. The food takes the edge off some of your irritableness, turns some of it into curiosity.

"This brimstone… is that why it's so deep underground? We literally buried it?" You notice that you're still using "we", even though Jerome knows what you are. Old habits die hard, and who knows who else might be listening?

"The brimstone is figurative, a reference to a history farther back. We used rockets, bombs, guns, knives. The burying came later. When the food ran out and we started digging, the dirt from the

tunnels had to go somewhere. Eventually all the great cities- even those that submitted- had to be buried so we could feed our people."

You look at the walls, trying to uncrush them in your mind and recreate what had been there before. It's maddeningly difficult, having only seen a scant few pictures of these ancient above-ground cities.

"You've studied this a lot, haven't you."

"It's fascinating," he said. "It's good to know your past. I'm sorry that so much of it is my people being dicks to your people."

You shrug. Your belly is getting full, and all this seems distant now, with the Ghost threat and all. "From what I hear, we were dicks to you sometimes as well."

"Yea. But it's the final dicking that everyone remembers. Most of the victors remember it as glorious, but those who study the history are somewhat more torn."

"I'm surprised the elders allowed all this investigation."

Jerome shrugs. "The highly useful like us have certain freedoms that others are not granted. Not enough freedom to become apostate, but enough freedom to investigate subject that are Haram."

Jerome gets up, and you follow suit.

"Farther down?"

"Farther down. To the power pellets."

17: Moving On Down

The chase is on- we've got Pacs running away from Ghosts, getting caught, running through their life reserves. Things look pretty grim for the Pacs! But in this and the next few chapters we're going to give them a few tools to turn the tide.

The first is to let them move on to another level. Making is to they no longer just repeat the same level over and over makes their task seem a bit less Sisyphean. Of course, they'll still loop through all the available levels (no winning screen for you!), but it's less obvious.

The Level Array

We'll start by defining the level array:

```
1  // components/pac-man.js
2  import Level from '../models/level'
3  import Level2 from '../models/level2'
4
5  export default Ember.Component.extend(/*mixins*/, {
6    levels: [Level, Level2],
7    //...
8  })
```

We're importing the two levels we've defined (you can define and import more if you like) and putting them in an array. Now we'll use that array to select which one to start with, instead of hard-coding one or the other like we were before.

Let's define the method that we'll use to do that (levels array included for reference):

```
1  // components/pac-man.js
2  levels: [Level, Level2],
3  loadNewLevel(){
4    let levelIndex = (this.get('levelNumber') - 1) % this.get('levels.length')
5    let levelClass = this.get('levels')[levelIndex]
6    return levelClass.create()
7  }
```

This method ends up replacing the `Level.create()` or `Level2.create()` call we were using before.

The first line is taking the `levelNumber` (which starts at 1 and then goes up by 1 every time we beat a level) and turning it into a tool for finding the correct `Level`. First we subtract 1, since arrays (like the `levels` array) are indexed starting at 0 instead of starting at 1. Then we use the remainder operator (%) in order to make sure that we're hitting an actual level. So if we have 2 levels, and `levelNumber` is 3, it would go back to the first one ((3 - 1) % 2 == 0).

The second line takes that `levelIndex` and uses it to look up the level class from the `levels` array.

In the third line we'll create an instance of the class. Remember, in the levels hash we're not storing strings, we're storing a reference to the actual class itself, so it's live and we can instantiate it.

We return the result of that creation.

How is that used elsewhere? Well, we could use it to simply replace `Level2.create()` within `didInsertElement`.

```
1  // components/pac-man.js
2  didInsertElement(){
3    let level = this.loadNewLevel();
4    //...
5  }
```

That's a great start, but we have bigger plans.

Changing Levels

Currently, there is exactly one place in the code where we want to load a new level and its attendant Pacs, Pellets, and Ghosts. When the game is over, whether we lose or win, we simply restart. Level changing complicates that.

The whole of the apparatus that we have in `didInsertElement` (aside from starting the loop) will now be called both at the beginning of the game AND when we change levels. Therefore, we need to decouple that code from `didInsertElement`.

```
// components/pac-man.js
didInsertElement() {
  this.startNewLevel();
  this.loop();
},

startNewLevel(){
  let level = this.loadNewLevel();
  level.restart()
  this.set('level', level)

  let pac = Pac.create({
    level: level,
    x: level.get('startingPac.x'),
    y: level.get('startingPac.y')
  });
  this.set('pac', pac);

  let ghosts = level.get('startingGhosts').map((startingPosition)=>{
    return Ghost.create({
      level: level,
      x: startingPosition.x,
      y: startingPosition.y,
      pac: pac
    })
  })
  this.set('ghosts', ghosts)
},
```

This is a lot of code, but it's mostly just taking the code from `didInsertElement` (aside from starting the loop) and sticking it into `startNewLevel`. Then we call `startNewLevel` from within `didInsertElement`, achieving the same effect but now allowing the abstracted code to be called from elsewhere.

Note that no matter how many times we start a new level, we're only starting the loop once. That's because starting it multiple times is a recipe for bad performance and/or weird behavior- better to keep the same loop and just carry on with new versions of the Pac, the Level, and the Ghosts.

> There is one other change, and that is calling `level.restart()`. You may think: "Why are we calling restart on a newly created level? Surely we don't need to do that!". And for leveling up that is indeed true... but for some reason, when I beat the first level but then run out of lives on the next level, the game restarts me on a level 1 that is completely devoid of pellets. I don't know why this happens, but this call fixes it.

> This interlude goes to show you that it is often easier to fix a problem than to understand it. Use your best judgement about when that's enough.

So now we can just call `startNewLevel` from elsewhere.

```
processAnyPellets(){
  let x = this.get('pac.x');
  let y = this.get('pac.y');
  let grid = this.get('level.grid');

  if(grid[y][x] == 2){
    grid[y][x] = 0;
    this.incrementProperty('score')

    if(this.get('level').isComplete()){
      this.incrementProperty('levelNumber')
      this.startNewLevel()
    }
  }
},
restart(){
  if(this.get('lives') <= 0) {
    this.set('score', 0)
    this.set('lives', 3)
    this.set('levelNumber', 1)
    this.startNewLevel()
  }
  this.get('pac').restart();
  this.get('ghosts').forEach( ghost => ghost.restart() );
},
```

Awesome stuff! From here on out, you can have fun creating new levels and putting them in different orders- it's almost like you have a complete game!

> Bonus: We're currently setting the Pac and Ghost starting positions from the level... you could figure out how to specify other variables, such as color or speed (framesPerMovement).

Summary

In this chapter we covered how to create a level progression. Pretty cool! And as a tribute to how powerful our Javascript and Ember knowledge has become, we only had to learn one new construct (the remainder operator `%`) in order to get that done.

> Of course, learning new things is always good, don't get me wrong. But being able to code something without looking up new constructs is like being able to have a conversation without learning a new word- super convenient.

Now we're in the home stretch, and the rest of the book will just be putting in cool features like teleporters and power pellets. Enjoy the ride!

Patrol

You're not sure how, but the Ghosts have caught up. Every level you go down seems to have them wandering, searching for you. As such, you and Jerome have built up a repertoire of tricks to detect and avoid them while still diving deeper and collecting the maximum number of pellets.

Night-time (now even less connected to the solar cycle than it was before) you take turns keeping watch. The hardest part isn't the reduced sleep, it's knowing that at any moment you could be pulled from your rest and have to make your escape. The scariest part, as exhaustion sets in, is wondering whether you'll be able to get up in time for the next attack.

But you continue, going down. Patterns repeat.

Occasionally Jerome will find a communications device that lets you talk with people up top. Terrance has survived, is in hiding, but Matteo was killed in the first wave. You don't ask about other individuals by name, but the casualty rate is astonishing. They insist that you must find the power pellets and make a weapon.

Eventually you reach a new piece of tech- a teleporter.

"The pellets down here are denser, allowing us to build these things. When you go into one, it takes you to another part of the map."

The teleporters are useful, but even better is the fact that the Ghosts don't seem to understand how they work. You grab some much-needed rest by camping out right on the edge of a teleporter and simply making the jump whenever a Ghost appears nearby. It's a risky strategy though, if the Ghosts ever figure out the teleport system, so after three days of doing this Jerome puts a halt to it.

And farther down you go. The days and nights blur.

The PAC display says it's been two weeks.

18: Teleport

Teleportation- in this case, the ability to jump from a square at one end of the level to a corresponding square at the other end of the level- sounds like a feature that will be difficult, but in actuality it's quite easy. Not only that, but it's easy to turn on and off based on the level- that means that if you designed a bunch of levels after the last chapter, there's no need to go back and rewrite them.

Finding What to Change

Let's take a look at the current state of our technology:

```
1  //mixins/movement.js
2  pathBlockedInDirection(direction) {
3    let cellTypeInDirection = this.cellTypeInDirection(direction);
4    return Ember.isEmpty(cellTypeInDirection) || cellTypeInDirection === 1;
5  },
6
7  cellTypeInDirection(direction) {
8    let nextX = this.nextCoordinate('x', direction);
9    let nextY = this.nextCoordinate('y', direction);
10
11   return this.get(`level.grid.${nextY}.${nextX}`);
12 },
13
14 nextCoordinate(coordinate, direction){
15   return this.get(coordinate) + this.get(`directions.${direction}.${coordinate}`\
16 );
17 },
```

These methods, which are mixed in to both the Pac and the Ghosts, are telling us which directions we're allowed to move in. `pathBlockedInDirection` takes a direction, and then finds the cell type in that direction (via `cellTypeInDirection`). If it's a wall or it's null, then it returns true- the path is indeed blocked.

That null value isn't there because of an error- it's there because it represents when `cellTypeInDirection` 'falls off the grid'. It looks on the grid and returns a null value for those coordinates because those coordinates don't exist. Those are the invisible walls around our levels that we've been taking for granted. When we add teleportation, we'll be returning something else instead of null.

To get the cell type, it reaches into the grid, but the x and y values that it reaches into the grid with are determined by `nextCoordinate`.

`nextCoordinate` is currently very simple: it calculates one coordinate (x or y) at a time, saying what that coordinate would be if we moved one square in the passed-in direction. It does this by grabbing the current value of that coordinate, then adding the change that would occur based on the passed-in direction and the directions hash (either 0, -1, or 1).

`finalizeMove`, which does the actual moving, also calls upon `nextCoordinate`. That means that changing `nextCoordinate` will not only change how we determine whether we can move or not, it will change how we actually do the moving, and it will do it in a manner that they're consistent with each other.

Where Do We Go From Here

So we've found the method we need to change- let's get to changing!

```
1  nextCoordinate(coordinate, direction){
2    let next = this.get(coordinate) + this.get(`directions.${direction}.${coordina\
3  te}`);
4    if(this.get('level.teleport')){
5      if(direction == 'up' || direction == 'down'){
6        return this.modulo(next, this.get('level.height'))
7      } else {
8        return this.modulo(next, this.get('level.width'))
9      }
10   } else {
11     return next;
12   }
13 },
14
15 modulo(num, mod){
16   return ((num + mod) % mod);
17 },
```

Let's start off with what's familiar: the first line of `nextCoordinate`. It's the same calculation, but instead of returning it directly it assigns it to `next`. Now if the value of `teleport` on the currently loaded level is false, it will return `next`- essentially giving us the same method as before. But if the value is true, we do something different.

First we'll split into two different branches based on the direction that's passed in. If it's up or down, we go to the first branch where we'll be using the level's height. If it's left or right, we go to the second branch where we'll be using the level's width.

Then, we'll be using our own specialized modulo function. So if we do a regular modulo (`%`), it works great for things that are higher than the divisor. `5 % 2 == 1`. But when you get to negative numbers, the modulo doesn't automatically make them positive. So `-1 % 2 == -1`. If we want the Pac to go off the top of the level and show up at the bottom, we'll have to fix that. Hence our own modulo function, which just adds in the divisor before running Javascript's built-in modulo.

Looking back at `nextCoordinate` with this knowledge, we see that we're taking the next coordinate value and modulo-ing it with the relevant level dimension.

As an example: if it's going off the right of the screen of a level of width `8`, then the `next` value will be `8`, and (remember: this is all base-zero, so `7` is the last available value for a level with width `8`) our modulo will bring it to a value of `0`. Then if you turn around, your `next` value will be `-1`, and our modulo will bring it to a value of `7`.

So that's how you make a teleporter.

> While both the Pac and the Ghosts will be able to teleport, it's not common at all for Ghosts to teleport. That's because their heat-seeking capabilities don't take teleportation into account, so they will almost always move in the opposite direction. Remember that when designing levels.

Turning It On

To use the teleport function, you have to set `teleport: true` in your level. Here's an example:

```
1  import Ember from 'ember';
2
3  export default Ember.Object.extend({
4    teleport: true,
5    //...
6  })
```

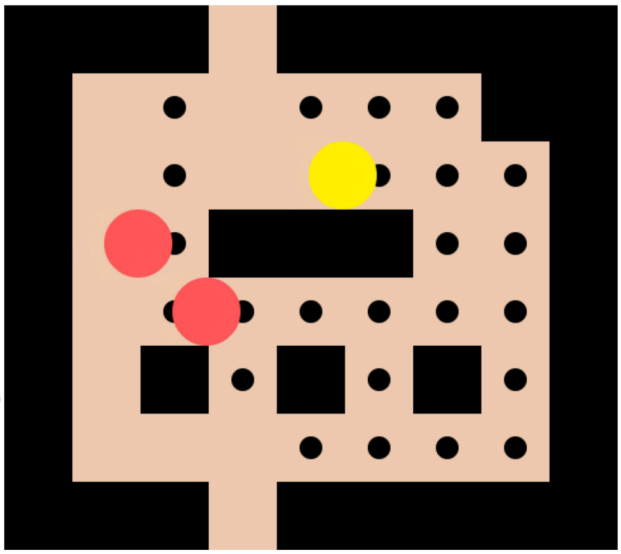
If you set it true in the base level, it will automatically be true in the child levels unless you turn it specifically turn it off in them. If you don't set it in a level at all, and it's not inheriting from a level that has it set, then it will automatically return `null`. `null` ends up acting the same as `false`.

A Teleport-First Level

Although we can turn on teleportation in any level, it feels wrong to have an entire line of teleportable squares. It feels more right for a teleportation level to be surrounded by walls except at certain points that are cut away to reveal a teleport.

Here I show a level that looks more like what you would normally see as a teleport level. Notice the near-solid `1`'s on the outside- those are the walls that frame the level and make the teleport squares feel special.

```
//models/teleport-level.js
import Level from './level';

export default Level.extend({
  squareSize: 60,
  grid: [
    [1, 1, 1, 2, 1, 1, 1, 1, 1],
    [1, 2, 2, 2, 2, 2, 2, 1, 1],
    [1, 2, 2, 2, 2, 2, 2, 2, 1],
    [1, 2, 2, 1, 1, 1, 2, 2, 1],
    [1, 2, 2, 2 ,2, 2, 2, 2, 1],
    [1, 2, 1, 2, 1, 2, 1, 2, 1],
    [1, 2, 2, 2, 2, 2, 2, 2, 1],
    [1, 1, 1, 2, 1, 1, 1, 1, 1],
  ],
  startingPac: {
    x: 1,
    y: 1
  },
  startingGhosts: [{
    x: 6,
    y: 6
  }, {
    x: 5,
    y: 1
  }],
  teleport: true
})
```



Summary

This chapter reviewed our movement code, analyzed the control structures within it, and edited it at a crucial point to allow for teleportation. We set it so that it would only teleport if the level specified that we should, and then we created a level that feels more fitting for teleportation.

Before we leave this chapter, I want to take a moment to point to something we didn't do: we didn't change anything about the movement algorithm outside of `nextCoordinate`. That's the power of abstraction- we just changed how we found the next coordinate, and everything else simply used the new values.

In the next few chapters we'll be turning the tables- creating a Power Pellet that makes the Ghosts run away.

A New Hope

The weeks stretch together- sleep, run, eat, run, sneak, explore, run. The ghosts are relentless.

Going downward did have its benefits, though. The pellets became more dense, allowing for a greater energy expenditure in food preparation. That is, more delicious and more nutritious food. When you weren't dizzy from sleep deprivation, you could take a small amount of pleasure in the delectable flavors that hadn't been available at lower energy densities.

"The PAC-men must love going down this far," you say. "I mean, when there aren't ghosts chasing after them."

Jerome nodded agreement. "That's one of the perks of the job. You get to eat of the old recipes."

"The old recipes?"

"The pellet density, it used to be like this everywhere- even up at the surface. I mean, at that time the density down here was probably even greater. But you harvest the pellets, and each time they grow back a little slower, a little smaller. It's the fossil fuel problem all over again."

You stare at him blankly. Maybe you've been down here too long for anything to make sense any more. "The what?"

"The fossil fuel problem. There used to be a liquid energy in the earth that we burned to make stuff go. Just like the pellets, but even more dense. It was wonderful, or so the legends say. But it kept on getting harder and more expensive to dig up, and the effects of burning it were turning the planet… well, you know the tree sanctuaries? They were everywhere before. Anyways, the caliphate tried to stop it, but eventually they succumbed to the same pressures and desires that led the infidels to burn so much liquid energy."

"And now we have a desert world where we get our food from pellets."

Jerome nodded. "Yea. But that's not quite the whole story. We could still grow food above-ground for a little while, in places where we weren't harvesting pellets. There were more of us then."

You don't get to ask him about those last few statements because you detect a Ghost in the vicinity. Jump in, maneuver away. Rinse and repeat.

Jerome finds another communication station, says simply: "We're close."

Meanwhile, you've started hearing things.

"We have a plan," the voices say. You don't want to know what this plan is, but the voices continue. "Join us, Salam. We have worked out the plan from first principles."

You ignore them the best you can.

Then, a day later, you finally find it. A power pellet.

“It’s bigger than a typical pellet,” you say. “Even the more dense ones down here.”

Jerome nods. “So much energy. Isn’t it beautiful?”

“So… how’s it going to fit?”

“We started making some modifications to fit this size of pellet. We’ve temporarily repurposed the thermal exhaust port- when you run into a power pellet, it’ll turn off the heat exhaust function for a couple seconds, long enough to force in the pellet. It’s kludgy, but it’ll do.”

“So I just run over the power pellet?”

“Yea. We’ll do a trial run, and then you can get to work experimenting on the next one we find.”

You move your PAC on top… and it’s electrifying.

19: Power

So we've found the power pellets... what to do with them? In this chapter we're going to place the power pellets on the map, turn our Pac green based on, then deal with a problem that arises from our power pellet placement.

On The Map

In our current grid, we use `0` to represent a blank square, `1` to represent a wall, and `2` to represent a pellet. Now we're going to add `3`, which will represent a power pellet.

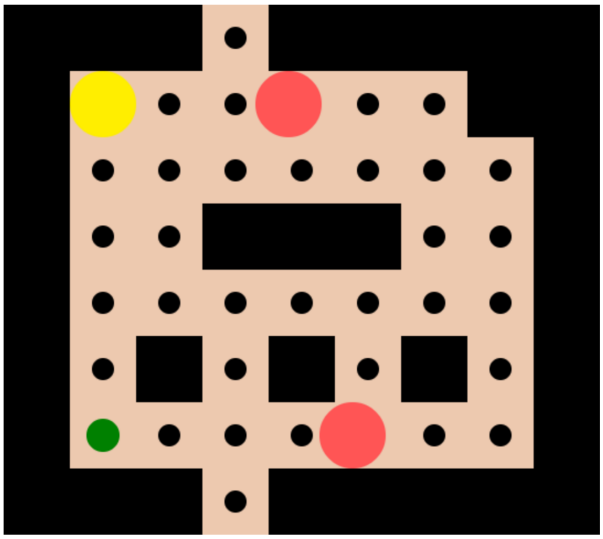
Adding a power pellet to a level is at easy as replacing a `2` with a `3`- go ahead and do that on the first level in your levels array.

Of course, the power pellet isn't *drawn* or *interacted with* yet, but that's what our next code changes will do.

```
1  //components/pac-man.js
2  drawGrid(){
3    let grid = this.get('level.grid');
4    grid.forEach((row, rowIndex)=>{
5      row.forEach((cell, columnIndex)=>{
6        if(cell == 1){
7          this.drawWall(columnIndex, rowIndex);
8        }
9        if(cell == 2){
10         this.drawPellet(columnIndex, rowIndex);
11       }
12       if(cell == 3){
13         this.drawPowerPellet(columnIndex, rowIndex)
14       }
15     })
16   })
17 },
18 drawPowerPellet(x, y){
19   let radiusDivisor = 4;
20   this.drawCircle(x, y, radiusDivisor, 'stopped', 'green')
21 },
```

This set of changes draws the power pellet. In `drawGrid` we're adding a check for `cell == 3`, which is of the same form as the checks for `1` or `2`. If the check matches, it calls the new method `drawPowerPellet`, which is very similar to `drawPellet` but with a different radius and color.

> You may ask: "why not use an if/else stack, instead of these three separate ifs?" That is a great question, which I will consider deeply when writing the second edition of this book.



So that draws it, but if you move the PAC over it, nothing will happen. We need to add code that makes them interact.

```
1  //components/pac-man.js
2  processAnyPellets(){
3    let x = this.get('pac.x');
4    let y = this.get('pac.y');
5    let grid = this.get('level.grid');
6
7    if(grid[y][x] == 2){
8      grid[y][x] = 0;
9      this.incrementProperty('score')
10
11     if(this.get('level').isComplete()){
12       this.incrementProperty('levelNumber')
13       this.startNewLevel()
14     }
15   } else if(grid[y][x] == 3){
```

```
16      grid[y][x] = 0;
17      this.set('pac.powerMode', true)
18    }
19  },
```

The code we've added is the last clause- the second branch in the if/else block. It checks to see if the grid value is 3, and if it is, it sets the value to 0. Then it sets `powerMode` on the `pac` to true.

Changing Colors, Ternary Operators

What does `powerMode` do? Well, eventually it's going to do a lot of things, but for now we'll satisfy ourselves with just one thing: changing the Pac's color.

```
1  //models/pac.js
2  powerMode: false,
3  draw(){
4    let x = this.get('x');
5    let y = this.get('y');
6    let radiusDivisor = 2;
7    let color = this.get('powerMode') ? '#AF0' : '#FE0';
8    this.drawCircle(x, y, radiusDivisor, this.get('direction'), color);
9  },
```

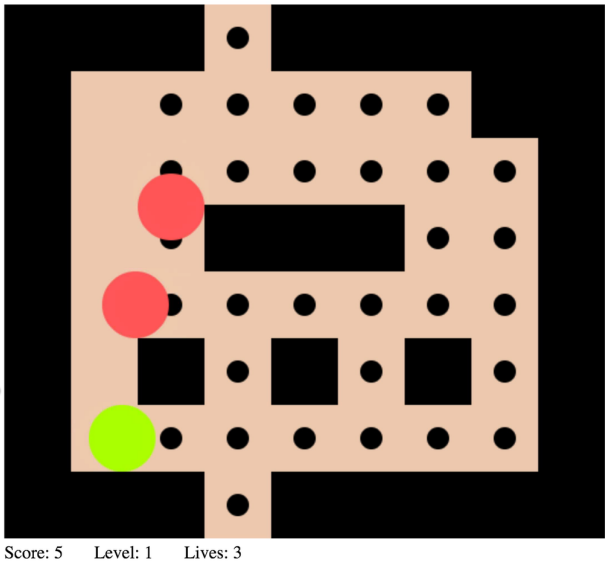
The main thing that we changed in `draw` is the following line:

```
1  let color = this.get('powerMode') ? '#AF0' : '#FE0';
```

It may look pretty unfamiliar- that's because it's a ternary operator. Fortunately, it can be expressed in terms of an if/else statement:

```
1  let color;
2  if(this.get('powerMode')){
3    color = '#AF0';
4  } else {
5    color = '#FE0';
6  }
```

As you can see, the ternary operator is way shorter. It also removes the need to awkwardly declare the variable before making the if statement.



Resetting the Level

We can now draw the power pellet, eat the power pellet, and change the Pac's color in response, but there is one bug that comes up. If you eat the power pellet then run out of lives, you'll notice that the power pellet doesn't regenerate- it's a regular pellet now.

The reason this happens is because our current `restart` function simply takes every blank space and fills it with a regular pellet. It's got no respect for the grid we've set out.

> "Wait a minute," you might ask, "We're creating a new level each time we call `startNewLevel`, right? And when we're making the new level, it should create it from the grid, right? And then that grid is on the object and shouldn't effect the class definition, which is used to instantiate later levels, right?"
>
> I think you're right in all of those things. But Ember's object system has some issues which, long story short, mean that when you define an array as a property on an object, different instantiations can end up sharing the same data. This is the same problem that we had with the pellets not showing up when we restarted, which is why we're calling `level.restart()` in `startNewLevel`.

The way we'll fix this is by separating out the grid that the level uses and the layout of the grid. So as a first step, rename all your `grids` into `layouts`. Then, we'll change the restart function:

```
//models/level.js
layout: [
  [2, 2, 2, 1, 1, 1, 1, 1],
  //...
],
restart(){
  var newGrid = jQuery.extend(true, [], this.get('layout'));
  this.set('grid', newGrid);
}
```

Instead of looping through every single cell and replacing stuff, we're just replacing the grid wholesale with a copy of our layout. Simple and effective.

The crux of this is our use of jQuery's `extend` method. It can be used for more, but here we're using it as a way to copy the layout.

The first argument to `jQuery.extend` is a boolean that says we'll be doing a "deep" extend. That means it won't just copy over the most shallow layer (the rows), but it will also go deeper (in this case, getting the cells). That's important, because it's the cells that we want!

The second argument is the target object. That's the object that we'll be putting the copied information onto. In this case it's a new array, which guarantees that it's a "fresh" javascript construct that is unrelated to the layout (and hence changes to it will not effect the layout).

The third argument is the object from which we'll be copying over properties.

So taken all together, we're copying over `layout` in its entirety to a new javascript array, then assigning it to `newGrid`. `grid` is then set to that value. Whew!

Summary

This level was a mixed bag in terms of difficulty. We started off with the simple addition of the power pellet to the map, and adding in the interaction. Then we moved on to introducing a new construct, the ternary operator, which is basically a really nice shortcut for some types of if statements. Finally we addressed a really thorny bug- while acknowledging that we don't understand *why* the underlying bug happens- and learning about jQuery's extend method.

In the next chapter we'll introduce the Power Pellet's offensive effects. Ghosts Beware!

The PACs Strike Back

The initial effects of the power pellet are impressive at first, but ultimately not that useful. You get a jolt of electricity, and the PAC turns green.

Fortunately, the instruments you brought down with you are still intact. You experiment, first getting the unique energy signatures and constituent elements from the pellet, then trying to recombine them into something weaponized. The elders were right that this had potential. Possibly catastrophic amounts of potential, but for today you're focused on something to protect the PACs from the Ghosts.

After a couple hours measuring and studying the pellet itself, you start adding different formulations of the pellet to the PAC to see how it changes things.

Your pellet runs out before you find anything interesting, but the deeper you go the more numerous the power pellets become. Each one leads to more experiments, each one getting closer to an answer, either confirming a theory or letting you discard one.

Finally, it happens. You hop in the PAC and you can feel the power. You write the conversion program- very quick, quick enough to be used in battle- and load it into your PAC's permanent memory.

Jerome sends the conversion program to command. It won't do them much good up there, without power pellets, but it will be useful if they come down here, or if you figure out how to preserve power pellets long enough to bring them up there.

But for now, it's time for payback.

20: Turning the Tide

In this chapter, we finally give the Pac some offensive capabilities- the ability to chase away the ghosts and win against them when in powerMode. While we don't introduce any big new programming concepts, we'll exercise our current knowledge and learn a couple new bits of syntax.

Ghost Run

The ghosts are usually chasing you, but that's not necessarily the best move for them after you've swallowed a Power Pellet. It's much wiser for them to run away.

We'll do this by changing the `chanceOfPacmanIfInDirection` method in `ghost`:

```
//in models/ghost.js
chanceOfPacmanIfInDirection(direction) {
  if(this.pathBlockedInDirection(direction)){
    return 0;
  } else {
    let desirabilityOfDirection = ((this.get('pac.y') - this.get('y')) * this.ge\
t(`directions.${direction}.y`)) +
                  ((this.get('pac.x') - this.get('x')) * this.get(`directions.${\
direction}.x`))
    if(this.get('pac.powerMode')){
      desirabilityOfDirection *= -1;
    }
    return Math.max(desirabilityOfDirection, 0) + 0.2
  }
},
```

First, we've renamed what used to be called `chance` to `desirabilityOfDirection`, a much more descriptive name. That's because we're no longer just calculating whether going in that direction will increase a ghost's chances of getting to the `Pac`, we're calculating whether it's a good idea to go in that direction. And when our Pac is in powerMode the best move is to run away- making the desirability of a specific direction the exact opposite of what it normally would be.

We accomplish this switch in the code by checking to see if the `Pac` is in powerMode, and then multiplying the `desirabilityOfDirection` variable by `-1` if it is.

Shorthand Operators

Here we're seeing a new piece of syntax, '*=':

```
let x = 5
x *= 2 //=> 10
x *= 3 //=> 30
let y = 5
y = y * 2 //=>10
```

We can see that '*=' is equivalent to multiplying the value of the left-hand-side variable by the right-hand-side value, then assigning the result to itself. It's a *shorthand operator*.

Similar shorthand operators can be found for other operators within Javascript.

```
let x = 5
x += 2 //=> 7
x -= 3 //=> 4
x /= 2 //=> 2
```

Shorthand operators can make your code both easier to write and easier to read.

More Specific Collisions

The Ghosts are running, and they're doing that because they know that if they collide with the Pac while it's in powerMode, they'll have to completely retreat from the battlefield.

In order to represent those consequences in-game, we'll first have to change our `collidedWithGhost` method so that instead of returning just `true` or `false`, it returns details on which Ghost collided with our Pac. Before it didn't matter, since the only consequences were for the Pac, but now the Ghosts themselves have consequences.

We'll also rename it `detectGhostCollisions`, since `collidedWithGhost` implies that the function returns a true/false answer.

```
detectGhostCollisions(){
  return this.get('ghosts').filter((ghost)=>{
    return (this.get('pac.x') == ghost.get('x') &&
            this.get('pac.y') == ghost.get('y'))
  })
},
```

As you can see if you compare it to your current code, it's almost exactly the same but with big difference- instead of using the any method, we're using the `filter` method. So instead of returning a boolean, we're returning an array of the items whose filter function returned true.

If one Ghost matched the function we fed to `filter`, then the returned array would have that one Ghost in it. If two Ghosts matched, the returned array would have two Ghosts in it. Most often it will return an empty array, since most of the time we're not running into any Ghosts.

When we're using `detectGhostCollisions` within the loop, we'll have to change up our technique a bit when we're checking to see if any collisions exist.

```
loop(){
  //...

  let ghostCollisions = this.detectGhostCollisions();
  if(ghostCollisions.length > 0){
    if(this.get('pac.powerMode')){
      ghostCollisions.forEach( ghost => ghost.retreat() )
    } else {
      this.decrementProperty('lives');
      this.restart();
    }
  }

  Ember.run.later(this, this.loop, 1000/60);
},
```

Now instead of just passing in the boolean to the if statement, we're checking the length and seeing if it's above zero. We also assign it to a variable so that we can reuse the array later on without running `detectGhostCollisions` twice.

After we've determined whether there are ghost collisions we need to check whether the Pac is in powerMode. If it's not, we'll do the same thing we've always done- decrement lives, then restart everything. If it is in powerMode, we'll loop through all the collided ghosts and tell them all to retreat.

Full Retreat

When we hit a Ghost, we want that Ghost removed from the playing field. That removal will be made temporary in the next chapter (and many of these details will change), but in this chapter we're just differentiating between removed and present.

We'll start by giving our Ghosts a boolean on `removed` and changing how it acts based on the state of that boolean

```
1  //models/ghost.js
2  removed: false,
3  retreat(){
4    this.set('removed', true)
5    this.set('x', -1)
6    this.set('y', -1)
7  },
8
9  //mixins/movement.js
10 move(){
11   if(this.get('removed')){
12     // do nothing, it's gone
13   } else if(this.animationCompleted()){
14   //...
15 }
```

`removed` is false by default (it's also falsy by default everywhere, but this makes it explicit). Then when we call `retreat` on a ghost, it sets `removed` to `true`. That has the effect of making the removed Ghost immobile, accomplished by creating a null branch at the top of a big if/else chain in `move`.

Also in the `retreat` function we're setting the position of the ghost to `[-1, -1]`... safely out of the board, so that the Ghost doesn't get stuck on a square the Pac needs.

Regroup

In the commercial version of pac-man, the Ghosts don't just disappear when eaten- they go to a specific place on the board while waiting for their chance to come back out. We're going to replicate that here by specifying those places in the level document, then using that in the `retreat` function.

```
//in models/teleport-level.js
import Level from './level';

export default Level.extend({
  layout: [
    [1, 1, 1, 2, 1, 1, 1, 1, 1],
    [1, 2, 2, 2, 2, 2, 2, 1, 1],
    [1, 2, 2, 2, 2, 2, 2, 2, 1],
    [1, 2, 2, 1, 1, 1, 2, 2, 1],
    [1, 2, 2, 2 ,2, 2, 2, 2, 1],
    [1, 2, 1, 2, 1, 2, 1, 2, 1],
    [1, 3, 2, 2, 2, 2, 2, 2, 1],
    [1, 1, 1, 2, 1, 1, 1, 1, 1],
  ],
  //...
  ghostRetreat: {
    x: 4,
    y: 3
  }
})
```

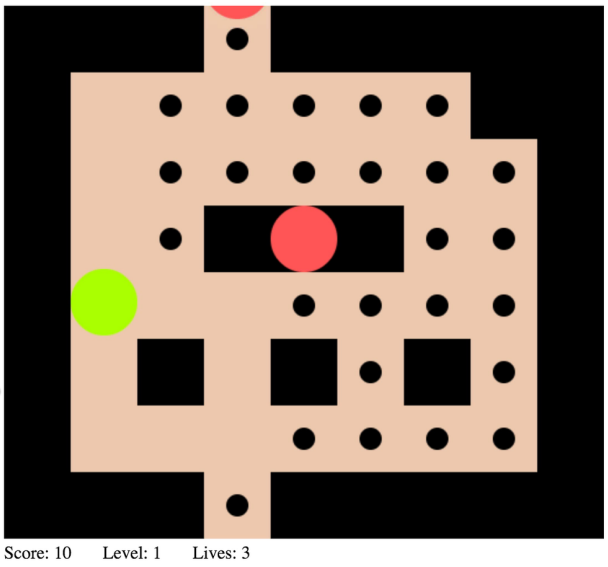
First we define the `ghostRetreat` hash, giving it an x and a y value. This coordinate is where we will put the retreating ghosts. Note that in this level, it's in the middle of a wall- somewhere that the Pac won't have to go in order to complete the level. That wall is also right next to an open space, which is important for when the Ghost eventually decides to come out from its retreat (next chapter).

Now we'll use the `ghostRetreat` when retreating.

```
//models/ghost.js
retreat(){
  this.set('removed', true)
  this.set('frameCycle', 0)
  this.set('x', this.get('level.ghostRetreat.x'))
  this.set('y', this.get('level.ghostRetreat.y'))
},
```

We're setting the `x` and `y` value to the values on the `ghostRetreat` hash. That will put them in the designated `ghostRetreat` square.

We're also setting the `frameCycle` to `0`- otherwise it will be as if the Ghost is in suspended animation halfway out of the retreat square (try it to see the effect).



Be sure to specify a `ghostRetreat` square on any level that has ghosts and a power pellet.

Summary

In this level we gained the power of the Power Pellet… but none of its limitations. Those limitations will catch up to us next chapter when we discover that the Power Pellet can't last forever.

But in the meantime, we can bask in our temporary Pac invincibility, our improved skill with complex programs, and our knowledge of the new syntax `*=` and `filter`.

The Return of the Jihadi

The chase continues, smashing one Ghost after another. You try to turn your eyes away from the wreckage of each tentacled machine, but fragments of images stick in your mind. Sometimes you see legs sticking out. Arms. It's hard to destroy the Ghost without killing whatever's inside.

Sometimes, you can hear screaming.

"We forgive," the voices say. "Come to us."

Too late you see what should have been obvious- that an individual power pellet can't last forever. You barrel into a Ghost, but this time it doesn't buckle and bend under your attack. The tentacles catch you and hold the PAC still, while others rip off layer after layer of the PAC.

"Salam," says Jerome, "I'm sorry. I'm sorry I couldn't protect you. Please, believe. Before it's too late. I want to see you in heaven. I want you to reap the rewards of our Jihad."

The radio cuts out and you're in the open, grabbed and swung by one of the tentacles.

"Cooperate," says a voice. "Cooperate, and we will let your friend go."

You look around, at least as much as you can while dangling in the air. Your PAC is destroyed, along with the scientific instruments. Jerome is boxed in but still making suicidal runs at the Ghosts that block his path. They rebuff him now, but they could easily destroy him.

There's nothing to say but 'yes'.

21: Time's Up

Good things can't last forever. That goes both for this book and for our Pac's powerMode invincibility.

Don't worry, we'll end the invincibility *before* ending the book.

Timers

This chapter will mostly consist of several fairly complex discussions while we set up the Pac's powerMode timer and the visual indicators that help our user figure it out. Then at the end we'll echo those discussions for our Ghost's retreat timer.

First, let's go over the general timer architecture, which we'll put into our `movement` mixin.

```
1  //mixins/movement.js
2  timers: [],
3  move(){
4    //...
5
6    this.tickTimers();
7  },
8  tickTimers(){
9    this.get('timers').forEach((timerName)=>{
10     if(this.get(timerName) > 0){
11       this.decrementProperty(timerName)
12     }
13   })
14 },
```

We have an array of timers- empty by default, but we'll be filling them up soon enough. Then at the end of the `move` function (called once every time through the loop) we call `tickTimers`.

`tickTimers` goes through every timer and, if it's greater than 1, decrements the property by 1. Because `get` and `decrementProperty` both take strings, the same loop can decrement any arbitrary property whose name we give it, which is a fantastically efficient way to do things. That way, the same `tickTimers` function can work for both the Pac timers and the Ghost timers.

So let's create a timer for our Pac- this one counting down how long we'll be in powerMode.

Power Waning

First we'll define some variables.

```
1  powerMode: Ember.computed.gt('powerModeTime', 0),
2  powerModeTime: 0,
3  maxPowerModeTime: 400,
4  timers: ['powerModeTime'],
```

We're setting four properties here. The first is redefining `powerMode`, which now instead of just being a `true` or `false` value, it's a computed property that returns a `true` or `false` value depending on several factors. In this case, if the `powerModeTime` property is greater than ('gt' for short) `0`, then `powerMode` is `true`.

> Want to know more about Computed Properties? Click Here to watch my videos on them[13].

The second property we're setting is `powerModeTime`, which is how much time we have left in powerMode. It starts at `0` and, as we'll see soon, gets set higher when we pick up a power pellet.

The third property is `maxPowerModeTime`, which here is `400`. Since we're getting approximately `60` ticks through the loop per second, this will last us `6` to `7` seconds.

The last property is the timers array. The default from `movement` is a blank array, but here we're adding in `powerModeTime`. That's telling it to tick that property down by one every time through the loop during our `tickTimers` call.

There's one other important piece of code that runs whenever we eat the power pellet:

```
1  // in components/pac-man.js
2  // run when eating the power pellet, replacing `this.set('pac.powerMode', true)`
3  this.set('pac.powerModeTime', this.get('pac.maxPowerModeTime'));
```

Here's where the whole thing starts. Instead of turning `powerMode` to true *directly*, we're doing it indirectly by setting `powerModeTime` to a value higher than `0`.

If you run the game now, you should be able to eat a power pellet, wait a couple seconds, and be normal again. However, that can be frustrating to the player (and quite devastating to our main character) when they're chasing a ghost and boom, they're normal again and they die.

Let's add some indicators that the player can see.

A New If

The first indicator will be the easier one.

[13] https://www.emberscreencasts.com/tags/computed-properties

```
<canvas id="myCanvas" width={{level.pixelWidth}} height={{level.pixelHeight}}></\
canvas>
<br>
Score: {{score}}       Level: {{levelNumber}}      \
; Lives: {{lives}}      
{{#if pac.powerMode}}
  PowerMode: {{pac.powerModeTime}}
{{/if}}
```

Here we've added some code to the end of our handlebars file which displays the `powerModeTime` variable on the Pac... but it only does so if `pac.powerMode` is true.

The construct we're using to do that is a Handlebars if block. The hash (#) right at the beginning tells us that it's a Handlebars block, so the `{{#if argument}}` section is what tells us it's an if block. The if block ends at `{{/if}}`. Anything in between is the block, and it is show if and only if the argument given in the if block declaration is true.

It's kind of like our Javascript if statement, but in handlebars form.

Changing Colors

The second indicator will be a bit more difficult, and will involve learning several new constructs.

What we're going to do is to have the Pac slowly fade from bright green to yellow, based on how much time is left in the timer. It's not that much code, but it's filled with new advanced stuff, so I'll just dump it here and then go over it concept by concept.

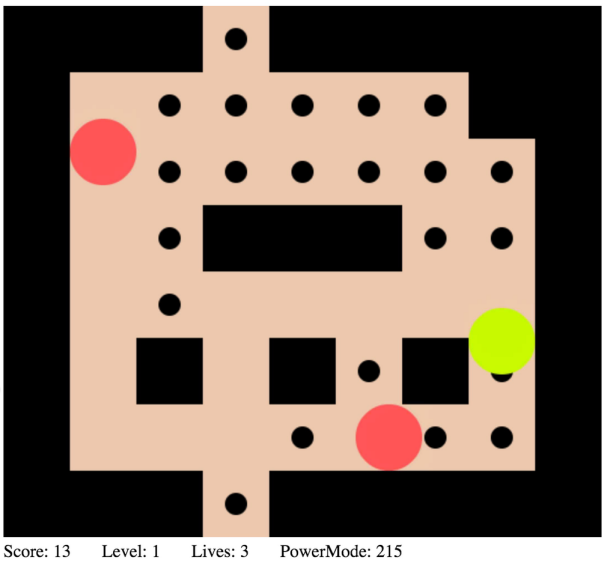
```
//models/pac.js
draw(){
  let x = this.get('x');
  let y = this.get('y');
  let radiusDivisor = 2;
  this.drawCircle(x, y, radiusDivisor, this.get('direction'), this.get('color'));
},
color: Ember.computed('powerModeTime', function(){
  let timerPercentage = this.get('powerModeTime') / this.get('maxPowerModeTime');
  let powered = {r: 60, g: 100, b: 0};
  let normal = {r: 100, g: 95, b: 0};
  let [r, g, b] = ['r', 'g', 'b'].map(function(rgbSelector){
    let color =  powered[rgbSelector] * timerPercentage +
                 normal[rgbSelector] * (1 - timerPercentage)
    return Math.round(color)
  })
```

```
17    return `rgb(${r}%,${g}%,${b}%)`
18  }),
```

The only change to `draw` is the fact that it's getting its color from the computed property `color` rather than defining it with a ternary operator (fancy if statement) like we were before.

Then we have the computed property `color`. It has a *dependent key* of `powerModeTime`, which means that if the property `powerModeTime` has changed since last time the `color` property has been called, then it recomputes. Otherwise it can use the same value as before.

The broad strokes of this are that we get a `timerPercentage`, a number between `0` and `1`, which corresponds to how much of the power we have remaining. Then we have two hashes (`powered` and `normal`) which specify the constituent amount of red, green, and blue that are required to make the powered and normal colors respectively (bright green and yellow). Then the next next big block is all about finding the current amounts of red, green, and blue to mix in, based on mixing together the powered and normal colors on a certain percentage based on how much power is left. Finally we put it in a form that CSS can understand.



Don't worry if you didn't get all that- it was just a summary, and we'll be going over several of the concepts in more detail.

RGB

In our previous dealings we'd used hexadecimal codes for our colors. They can be 6-digit codes comprised of the numbers 0-9 and A-F, but we'd been using the more simplistic 3-digit codes, like `#EF0` and `#FAA`. The hash symbol, in this context, indicates that we're using hexidecimal.

For our color computed property, we're going to be switching to RGB percentages. They're a bit more verbose, but they're much easier to reason about and to combine.

> Quick, what's the average of 40 and 60? Okay, now what's the average of 9 and C? The first one is much easier, because humans don't naturally think in hexadecimal. (The answers are '50' and 'A8')

Looking at these two hashes, we can take a decent guess at what they're going to look like when they're done:

```
let powered = {r: 0, g: 100, b: 0};
let normal = {r: 100, g: 95, b: 0};
```

We can see here at a glance that the powered state will be very green, and the normal state will be a mix of red and green (which ends up being yellow). The middle part then takes a weighted average (more on that soon) and feeds that to the rgb color syntax:

```
return `rgb(${r}%,${g}%,${b}%)`
```

A completely "normal" PAC (one with a `powerModeTime` of 0) would return `rgb(100%,95%,0%)`.

Mixing Colors

The middle part is where the magic happnes- we take a weighted average based on how much power there is. That's what the next part is all about.

```
let [r, g, b] = ['r', 'g', 'b'].map(function(rgbSelector){
  let color =  powered[rgbSelector] * timerPercentage +
               normal[rgbSelector] * (1 - timerPercentage)
  return Math.round(color)
})
```

There's two new parts here, as well as some not-super-intuitive algorithms going on. Let's take things one at a time.

The first new part is `Math.round`, but it's fairly self-explanatory.

The second new part is array destructuring. Here's a very simple example.

```
let [r, g, b] = ['red', 'gorilla', 'boxer'];
console.log(g)//=> 'gorilla'
```

Basically, we're assigning values to several variables based on their position in the receiving array. In our more complicated example, we're assigning those variables based on what's returned from the function within the `map`.

That function is where we get the individual weighted averages. The `rgbSelector`is either 'r', 'g', or 'b', and it can find the correct number from the `powered` and `normal` hashes. Then they're multiplied by either how much power is left (for the `powered` hash) or how much power has been drained (for the `normal` hash). Those are then added together. The entire effect is of it transitioning from one color to another as the timer ticks down to `0`.

Retreat, Redone

We can take all of the logic that we've just used to time-limit and color-code the Pac's powerMode and use it to time-limit and color-code the retreat of the Ghosts.

First we'll add in several properties on the Ghost:

```
//models/ghost.js
removed: Ember.computed.gt('retreatTime', 0),
retreatTime: 0,
maxRetreatTime: 500,
timers: ['retreatTime'],
```

Those might look familiar, because they're basically the same set of properties as `powerMode`, just with different names. In the same way, we'll change a line in `retreat` from setting the property to true to set the time to max, just like we changed the line when picking up a power pellet.

```
//models/ghost.js
retreat(){
  this.set('retreatTime', this.get('maxRetreatTime'))
  //...
}
```

The `retreatTime` variable then is ticked down to `0` by the `tickTimers` method, which is running automatically on every loop for every timer within `timers`.

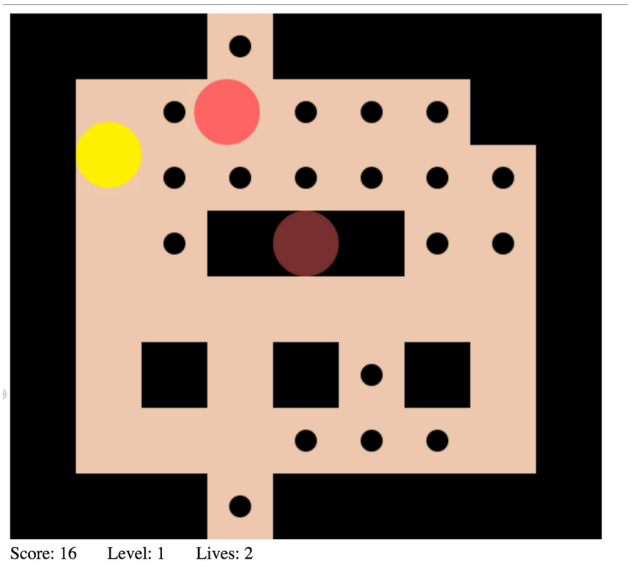
The color for the Pac is then set in a similar way:

```
//ghost.js
color: Ember.computed('retreatTime', function(){
  let timerPercentage = this.get('retreatTime') / this.get('maxRetreatTime');
  let retreated = {r: 0, g: 0, b: 0};
  let normal = {r: 100, g: 40, b: 40};
  let [r, g, b] = ['r', 'g', 'b'].map(function(rgbSelector){
    let color =  retreated[rgbSelector] * timerPercentage +
                 normal[rgbSelector] * (1 - timerPercentage);
    return Math.round(color);
  });
  return `rgb(${r}%,${g}%,${b}%)`;
}),
```

The numbers have changed, but most of the other stuff stays the same.

> Bonus assignment: abstract as much of the two `color` methods as you can to a common method that just takes in arguments for the two color extremes and the `timerPercentage`.

Now when the Ghosts retreat, they go to the retreat square, turn black, and slowly fade back in to red. It's a rather ominous effect. Then, when they're fully red again, they get back in the game.



Summary

We're finished with the game! Hurrah!

This last chapter was a big one, covering lots of new topics like the RGB color encoding scheme, destructured arrays, and handlebars block helpers. We've added in timers to change the state of the Pacs and the Ghosts, giving them effects that slowly wear off.

Even though this is the last chapter, there's still more that can be done. I'm going to make an addon so you can easily include it in your app. I'll include in that addon a set of levels that feel more like traditional PacMan levels.

Then, of course, there's plenty more that could be done to tweak the gameplay. Who said PacMan was the best possible version of PacMan? Maybe you can make it more fun! If you think you can, you now have the tools to do so.

> Did you improve PacMan? Send me your best shot. I want to make a list of cool PacMan variants created by readers.

Epilogue

“Why are you killing us?” you ask.

The Ghosts hmm and haw, shifting uncomfortably. They are both staring intently at computer screens. “One moment,” says the one called Elos. “We are trying to put this into words your culture can understand.”

The second one, Damian, nods in front of his screen. “This one is strange for his culture, more like us. That is why we have selected him, after all. He may be able to understand.” Damian says this with barely a change in expression.

“Very well,” says Elos. “The planet has been destroyed, just as the Prophet feared.”

“The prophet has worked it out from first principles.”

“So,” you say, “That doesn’t say why you’ve been killing us.”

“The prophet had a dream,” says Elos. “A dream of living sustainably from sunlight, humanity traveling between the stars.”

“A very strange dream at the time, but the holy book says He worked it out from first principles.”

They both look up from their screens, stare at you, then say in unison: “It has been fulfilled, though only in part.”

They look back down at their screens.

“Your civilization did not start this planetary destruction- that was the civilization before the one into which our Prophet was born- but your civilization has nearly finished it.”

“And that’s an excuse for killing us all?”

“Within three generations, you would all be dead anyways. The pellets would have dried up, the earth completely barren. Killing some of you now was necessary, so that later many may live.”

“That is why we sent the flood of… of what you call Ghosts.”

“So you kill half of us before we kill ourselves?” you ask. “Is that it?”

“Oh no, that’s certainly not all of it,” says Elos without expression. “We killed much more than half. And we’re not done.”

“Yes,” says Damian. “We have a bigger plan. A plan to restore the earth to its former glory- nay, to a glory far greater than it has ever seen. A society where men are peaceful and free.”

“You won’t like this plan, not at first. You will be reticent to help us, although your help will lead to much less… collateral. Think of it this way- we are separating the sheep from the goats.”

“And this plan,” adds Damian, “We have worked it out from first principles.”

What's Next?

We've built a great game of PacMan... what now?

First, if you want to continue your PacMan adventures, check out the PacMan addon[14] (based off the code in this book) and my screencast on how to customize levels in the PacMan addon[15] (it's changed a tiny bit, to make developing larger levels easier, and the default level set mirrors that of the original Ms PacMan). Using these, you can easily create your own levels and share them with the world.

If enough people purchase this book, I may write a sequel- it will tell the story of Salam's time with the Ghosts, and show how to create a PacMan level editor using Ember. So if you enjoyed this book, tell your friends!

Next, if you're wanting to know more about Ember, there are lots of great resources for you to check out.

- EmberScreencasts.com[16]. There are over 100 short videos teaching various aspects of Ember and related technologies (including some ES6), and I release two new videos every week.
- The Ember Guides[17]. The official guides. Well written, and pretty good for checking what the official way is to do things.
- Ember 101[18]. A free e-book that focuses on EmberJS.

And, of course, there's much more waiting to be discovered, and new stuff happening every day. I recommend Ember Weekly[19] for keeping up with the latest.

[14] https://github.com/jeffreybiles/pac-man
[15] https://www.emberscreencasts.com/posts/132-customizing-the-pacman-addon
[16] https://www.emberscreencasts.com/
[17] https://guides.emberjs.com
[18] https://leanpub.com/ember-cli-101
[19] http://emberweekly.com/