Infinite Jumper in Phaser 3 with Modern JavaScript

Tommy Leung

Contents

Introduction

Thanks for downloading this book!

We'll be taking you through 10 chapters of game development using the Phaser 3 HTML5 game framework in modern JavaScript.

The game we'll be making is an infinite jumper like Doodle Jump.

The objective is to get as high up as you can by jumping on platforms that disappear as they leave the bottom of the screen.

Falling is a death sentence. Well, virtual death sentence. You can always respawn.

Why Web and not Unity?

Phaser 3 is a fantastic game framework that is simple enough for beginners to learn yet powerful enough to make nearly any 2D game you can think of.

The web and modern JavaScript is the ideal platform and language for beginners.

Unity is not. I've tried teaching Unity to beginners. The software is as intimidating as an airplane dashboard. You'd think an integrated development environment would just work but it doesn't.

All you need in this book is Visual Studio Code, the Chrome web browser, and an Internet connection.

Why Modern JavaScript?

The syntax and style of JavaScript that we are calling "modern" will soon be just JavaScript.

It is also easier to learn and has less quirks that make JavaScript so hated by developers who use–in their opinion–*real* programming languages.

Modern JavaScript is plenty real.

Not too long ago, the only way to use modern JavaScript involved configuring complex development tools.

This just made it infeasible for beginners. The graveyard of beginners is filled with people who were overwhelmed by tooling.

There will be no complicated tooling or development environment to set up in this book.

How to Use this Book

This book was intended for you to read and follow along from start to finish.

It is not a reference book that you can easily jump around in.

Each chapter builds on the previous and the level of complexity ramps up as you progress.

This is not a book that will teach you the complete in's and out's of modern JavaScript.

We expect that you know basic JavaScript from a Codecademy, Udemy, or similar course.

Any code or game development concepts that we think may be confusing will be explained.

Our goal is to leave you feeling comfortable with experimenting and modifying the game to add your own changes.

Example Source Code

You can get the complete example source code for Bunny Jump on Github at:

https://github.com/ourcade/infinite-jumper-template-phaser3

This repository using `git-lfs` so be sure to have that installed if you plan to use `git clone` to ensure the images and audio assets are downloaded.

Alternatively, just download a `zip` from the Releases section.

Follow Along Video

If you run into trouble then we have a 10 part video series on YouTube that goes over each chapter from the perspective of a beginner.

We go through the book just like you would and troubleshoot problems as they come up.

You can find the playlist on our YouTube channel at http://youtube.com/ourcadehq or by clicking here.

We have every intention of improving this book with updates. Feel free to let me know of any errors, unclear instructions, or any other problems by sending an email to tommy@ourcade.co or letting us know on Twitter @ourcadehq.

Now, let's get down to game making! Shall we?

Ourcade is a playful game development community for open-minded and optimistic learners and developers. Visit us at http://ourcade.co

Getting Started!

We will keep the set up as simple as possible.

Web development has great tooling to make things easier and simpler… if you already know what you are doing.

In reality, all the tooling makes it *harder* for the true beginner.

This book will not be using `node` or `npm` and we will not assume you are comfortable using the command line.

All you'll need is a modern web browser like Google Chrome and Visual Studio Code–often referred to as VS Code.

Go download and install both if you haven't already.

Create a New Folder

Our game code will be contained in a folder. Make one called `bunny-jump` somewhere on your computer. The desktop is fine.

Then make a folder called `src` inside the `bunny-jump` folder.

All our JavaScript code files will go there.

Now open the `bunny-jump` folder in VS Code.

We are going to do one more piece of set up before creating any files.

Development Server

The term "development server" sure sounds fancy but it is just a program that runs on your computer.

We will simply use the Live Server extension for VS Code.

To get the Live Server extension, go to the Extensions tab of VS Code by clicking on the icon with 4 blocks on the left side toolbar.

Search for "Live Server" and pick the one by Ritwick Dey. Then just press the Install button and VS Code will handle it.

Once installed, you'll get a "Go Live" button in the blue footer bar of VS Code.

Alternatively, you can bring up the VS Code Command Palette by pressing `Shift` + `Cmd`/`Ctrl` + `P` and then typing in "Live Server" to see all the Live Server commands.

We will be using "Open with Live Server" option or the "Go Live" button to start a development server that runs our game code.

Create an index.html

Every web application needs an `index.html` and a Phaser game is no different.

Right click in the Explorer panel that shows your project folders and select "New File" to make an `index.html` file.

You want this file at the base of `bunny-jump` on the same level as `src`.

Select `index.html` and insert the HTML below:

```
 1  <html>
 2      <head>
 3          <title>Bunny Jump!</title>
 4          <script
 5              src="//cdn.jsdelivr.net/npm/phaser@3.24.1/dist/phaser.js">
 6          </script>
 7      </head>
 8      <body>
 9          <script type="module" src="src/main.js"></script>
10      </body>
11  </html>
```

Line 5 is where we are including the Phaser 3 framework from the Internet.

Then on line 9 we are including a `main.js` JavaScript file that we will have to make.

Our game code will start in this `main.js` file. This is also called the entry point of our game!

Create the Entry Point

Create a new `main.js` file in the `src` folder like we did for `index.html`. You'll have to right click on the `src` folder and then select "New File".

To start we will add a very simple line of code that prints a message to the browser's console.

Put this in your `main.js`:

```
console.log('Hello, World!')
```

Let's Go Live!

We don't have much but enough to test that our project set up is working.

Pro Tip: testing early and often will save you many headaches in the future.

Either hit that "Go Live" button near the bottom right of VS Code or select "Open with Live Server" from the Command Palette like we talked about in the Development Server section.

Live Server will most likely open a browser window for you at an address that looks like `localhost:5500`.

If it did not then a notification message will have popped up in the bottom right of VS Code telling you what port the server started at. Open a new Chrome window and go to `localhost:PORT_NUMBER` where `PORT_NUMBER` should be replaced by the number in the notification message.

A blank white page means your development server is running. The title of the tab should be `Bunny Jump` or whatever was between the `<title>` and `</title>` tags in your `index.html`.

Right click on the blank page and select "Inspect" to bring up the Chrome Developer Tools. You should see a `Hello, World!` message in the Console.

Troubleshooting

If you run into problems then make sure your project structure looks like this:

```
bunny-jump
    o-- src
       -- main.js
    -- index.html
```

You are also welcome to Tweet at us @ourcadehq with any questions!

Modern JavaScript

In a world not too long ago, you couldn't use modern JavaScript without a myriad of tools and libraries.

Beginners were stuck with learning traditional JavaScript but that is no longer the case.

Modern JavaScript is easier to understand and simpler to learn. It is also more inline with how professional developers write software.

That's why we will be using modern JavaScript in this book.

Phaser is included as a global meaning you can access `window.Phaser` in your code. This is convenient but not recommended in modern JavaScript.

It also doesn't let VS Code give us intelligent code completion. What good are these fancy computers if they can't help us?

We will do 2 things to make our use of Phaser 3 compliant with modern JavaScript and get full VS Code IntelliSense.

Using Modules

Modules are another fancy term. You can simply think of them as JavaScript files.

We will be importing various *bindings* (or exports) from modules using syntax like this:

```
import { banana } from './fruits.js'
```

The code above is importing the `banana` binding (or export) from a module called `fruits.js`.

Here's what `fruits.js` might look like:

```
const banana = {
    color: 'yellow',
    ripe: true
}

export {
    banana
}
```

See that we export `banana` with the `export` keyword. This is why we can import it in the earlier example.

We want to do this with Phaser but we don't have a file that exports Phaser. Remember that we are loading the distribution version of Phaser from the Internet with a `<script>` tag.

Distribution version just means that the Phaser source code was run through a bunch of tools to create a small, single package that works in both modern and legacy browsers.

So how do we solve this problem? Game development–and software engineering in general–is all about solving problems.

First, create a `lib` folder in the `src` folder. Then make a new file in `lib` called `phaser.js`.

The code in `phaser.js` will simply export `window.Phaser` from our `phaser.js` file.

```
export default window.Phaser
```

We can then import Phaser in the `main.js` file. Replace what was in `main.js` with this:

```
import Phaser from './lib/phaser.js'

console.dir(Phaser)
```

Save all your files and Live Server should detect that you made changes and automatically reload. If not then you may have to stop and restart Live Server or just manually reload the browser.

You should see an Object logged to the browser Console. Inspect it by clicking on it and it will expand to show various properties that belong to Phaser.

Make sure you include the `.js` extension in your **`import`** statement or you might get an error about MIME types.

VS Code IntelliSense

We will do one more thing before we start writing game code.

Visual Studio Code is a fantastic code editor that tens of thousands of professionals use to develop the apps we use every day.

And it is completely free!

One of the most useful features of VS Code is code completion or IntelliSense.

And in order to get that feature for external libraries like Phaser we need to provide some help in the form of type definitions.

Phaser includes an official type definition file that you can find on Github by going to the `types` folder in the official Phaser repository.

If the link doesn't work or look right then go to Github.com and search for "photonstorm/phaser".

Click on `phaser.d.ts` and then use Github's Download button to get the file or select "View Raw" and then copy and paste the contents into a file named `phaser.d.ts`.

Make a folder called `types` under the `src` folder and put the `phaser.d.ts` file there.

Next, create a new file called `jsconfig.json` at the base of the project on the same level as `index.html` and insert the following:

```json
{
    "compilerOptions": {
        "module": "es6",
        "target": "es6"
    }
}
```

This `jsconfig.json` file is to configure JavaScript handling for VS Code.

At the end of all this your project structure should look like this:

```
bunny-jump
    o-- src
        o-- types
            -- phaser.d.ts
        -- main.js
    -- index.html
    -- jsconfig.json
```

Go back to `main.js` and type out `Phaser.`. You should get an inline menu with different properties that belong to Phaser like `AUTO`.

Send us a tweet @ourcadehq if you can't get it to work. Attach a screenshot of VS Code or a short video of what you are seeing.

Ready, Player One

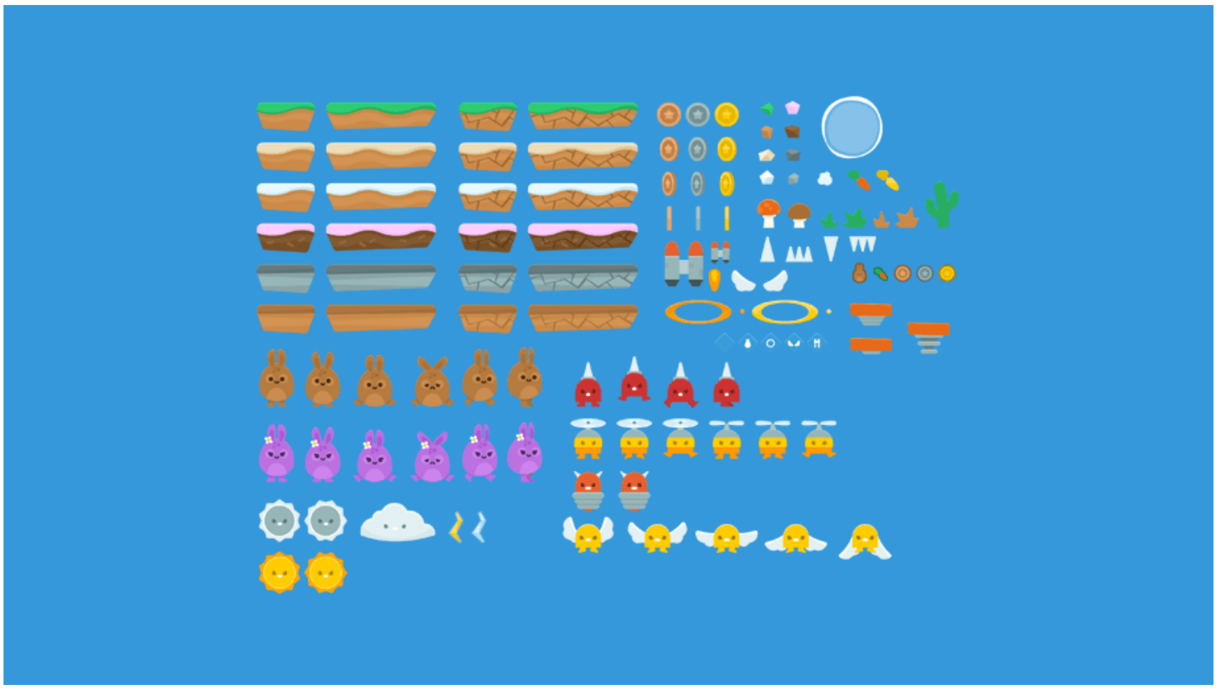
Your project is now set up and ready to go.

In the next chapters we will start creating a simple infinite jumper game like Doodle Jump.

Let's get started!

Setting Up the Game

We will be using the Jumper Pack assets from Kenney in our Bunny Jump game. Below is a preview of the assets.



You can download and use them for free. Kenney has a huge library of Public Domain art assets to help you make prototypes.

Go to http://kenney.nl, then click on Assets in the navigation and search for "Jumper Pack". Download and unzip the file.

This book will only use a few of the assets to build out a basic infinite jumper.

But, you can use the rest of the assets to add more features!

Here's the 5 assets we'll be using:

```
Jumper Pack
    o-- PNG
        o-- Background
            -- bg_layer_1.png
        o-- Environment
            -- ground_grass.png
        o-- Items
            -- carrot.png
        o-- Player
            -- bunny1_stand.png
            -- bunny1_jump.png
```

Copy those 5 images into your project under an `assets` folder. The folder should be at the base of the project on the same level as the `src` folder.

Creating a Phaser Game

Next, we will create a `Phaser.Game` instance in our `main.js` file.

`Phaser.Game` takes a configuration object that defines settings like the width and height of the game.

There are many other options that you can see here. Most have default values that are likely to be fine.

Add this to `main.js` after we import Phaser:

```
export default new Phaser.Game({
    type: Phaser.AUTO,
    width: 480,
    height: 640
})
```

You'll see that we are setting our game to have a width and height of 480x640. The `type` property is set to `Phaser.AUTO` meaning that Phaser will decide to use Canvas or WebGL mode depending on the browser and device.

Canvas and WebGL are two different ways Phaser can render your game in the browser.

Save your changes and you should see a black rectangle once the browser reloads. The Console will also have a message from Phaser noting the version, rendering mode, and other information.

The next step is to create a Scene.

Creating a Game Scene

Scenes are a way to group related logic and GameObjects.

It is a lot harder to explain than understand once you start using it.

For example, we will be creating a Game Scene that holds all the logic and GameObjects for the core infinite jumper part of our game.

Later, we will make a GameOver Scene with just the GameObjects and logic related to that.

Scenes give us a logical way to organize different parts of our game.

To make a Game Scene, create a new folder called `scenes` under the `src` folder. Then create a new file named `Game.js` in the `scenes` folder.

Our `Game` Scene will be a class that extends from the base `Phaser.Scene`.

Detour into Classes

Classes are an Object Oriented Programming concept for organizing related data and logic that you can then create instances from.

Imagine you have a `Rabbit` class with data like `energy`. With the `Rabbit` class you can make instances like `youngRabbit` with `energy = 100` and `oldRabbit` with `energy = 50`.

Both `youngRabbit` and `oldRabbit` have the same make up and structure but different stats.

Going back to our `Game` Scene, this is what the code in `Game.js` should look like:

```
1  import Phaser from '../lib/phaser.js'
2
3  export default class Game extends Phaser.Scene
4  {
5      constructor()
6      {
7          super('game')
8      }
9
10     preload()
11     {
12     }
13
14     create()
15     {
16     }
17 }
```

Every Scene has to define a unique key. We do that on line 7 in the `constructor` by calling `super('game')`.

The `preload()` and `create()` methods are hooks that get called at appropriate times by Phaser.

`preload()` is called to allow us to specify images, audio, or other assets to load before starting the Scene.

`create()` is called once all the assets for the Scene have been loaded. Only assets that have been loaded can be used in `create()`. Trying to use an asset that has not been loaded will result in an error.

Starting the Game Scene

We have a basic `Game` Scene class created but Phaser doesn't know about it!

Open the `main.js` file and add a `scene` property to the configuration object passed to `Phaser.Game` like this:

```
 1  import Phaser from './lib/phaser.js'
 2
 3  import Game from './scenes/Game.js'
 4
 5  export default new Phaser.Game({
 6      type: Phaser.AUTO,
 7      width: 480,
 8      height: 640,
 9      scene: Game
10  })
```

Our newly created `Game` Scene is imported on line 3. It is then used on line 9 to specify what Scene Phaser should start.

Save your changes and there should see no visible changes in the browser after a reload. It should still be a black screen.

We will change that next!

Preload a Background

There should be a `bg_layer1.png` file in our `assets` folder.

Let's load it in `preload()` so that we can use it in `create()`.

```
preload()
{
    this.load.image('background', 'assets/bg_layer1.png')
}
```

Notice that we are using the keyword **`this`**. It refers to the instance of the class. In this specific example, **`this`** is the current instance of the `Game` Scene.

The `Game` Scene has a member called `load` which is inherited from `Phaser.Scene`. See the documention here.

`Game` will have all the methods and properties that `Phaser.Scene` has because `Game` extends from `Phaser.Scene`. This is called inheritance in Object Oriented Programming.

The `load` property is an instance of `Phaser.Loader.LoaderPlugin` that has logic for loading images, audio, spritesheets, and more.

Use VS Code IntelliSense to quickly see the various methods and properties when you type `this.load.`.

Our example simply loads an image. We give it a key called `'background'` and then a path to the image file.

The key will be used later as a reference for creating images or sprites that are rendered by Phaser.

Creating the Background

With the `bg_layer1.png` image preloaded, we can add it to our `Game` Scene in `create()`.

```
create()
{
    this.add.image(240, 320, 'background')
}
```

You'll see that the style is very similar to what we saw in `preload()`. Instead of loading with `this.load` we are adding with `this.add`.

And the thing we are adding is an `image`. The two number values are the `x` and `y` positions of the image. The last parameter is the same key we gave for `bg_layer1.png` in `preload()`.

Save your changes and a blue background should appear in the browser after a reload.

With our `Game` Scene set up, we can move to adding game logic!

Creating Platforms

The goal of an infinite jumper game is to stay in the air for as long as possible by jumping off platforms.

So let's add some platforms!

First, we need to preload the platforms image in `preload()` like we just did for the background.

```
preload()
{
    this.load.image('background', 'assets/bg_layer1.png')

    // load the platform image
    this.load.image('platform', 'assets/ground_grass.png')
}
```

This code should look familiar now. The first value given to `this.load.image()` is the key and the second is the path to the image when served from the development server.

The path is **not** relative to the current file.

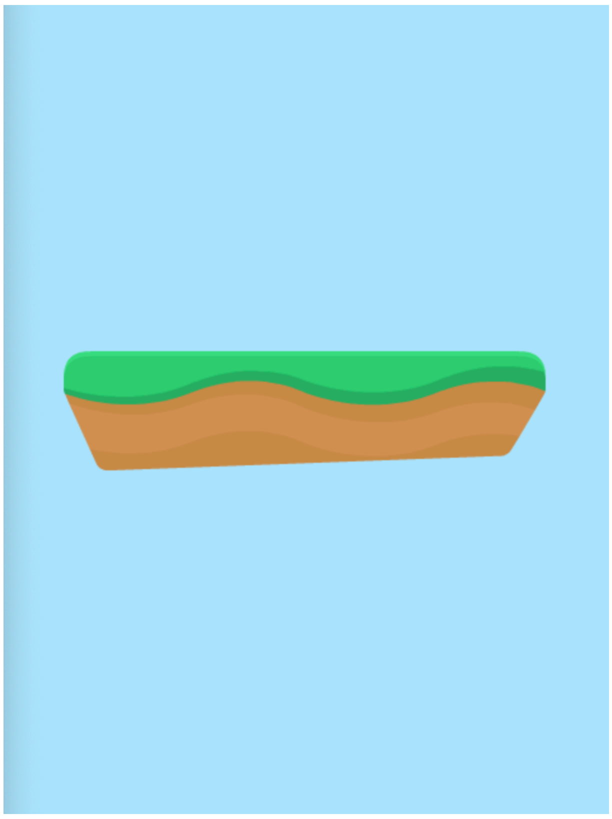
You can see this in action by going to `localhost:5500/assets/ground_grass.png` in your browser when the development server is running.

The Live Server extension takes your project folder and serves the whole thing.

With the platform image loaded, we can create it in `create()`.

```
create()
{
    this.add.image(240, 320, 'background')

    // add a platform image in the middle
    this.add.image(240, 320, 'platform')
}
```

Your game should now look like this:



There's a couple of problems here. First, the platform is HUGE! And secondly, it is just an image without physics.

We can solve the first by changing the scale like this:

```
this.add.image(240, 320, 'platform')
    .setScale(0.5)
```

Platforms look much better at half size. Now, let's add some physics!

Add Arcade Physics

Phaser has a few options for physics. We will use Arcade Physics to keep things simple.

The first thing we need to do is enable Arcade Physics in the game's configuration.

Let's update `main.js` like this:

```
export default new Phaser.Game({
    type: Phaser.AUTO,
    width: 480,
    height: 640,
    scene: Game,
    physics: {
        default: 'arcade',
        arcade: {
            gravity: {
                y: 200
            },
            debug: true
        }
    }
})
```

The key thing to note is the new `physics` property. We have also enabled `debug` to see the collision boxes.

Add a Physics-Enabled Platform

Now that we have Arcade Physics enabled, we can create a physics-enabled platform. Replace creating a platform image with this:

```
// instead of this..
// this.add.image(240, 320, 'platform')
//   .setScale(0.5)

// use this
this.physics.add.image(240, 320, 'platform')
    .setScale(0.5)
```

With this change to the `Game` Scene, you should see a platform fall off the screen.

We actually want our platforms to stay where they are. This is known as a *static* physics body.

Using `this.physics.add.staticImage()` is one option but, we also want a bunch of platforms.

So Let's take care of both issues by using `this.physics.add.staticGroup()`!

Adding Multiple Platforms

Here's how we create a `StaticGroup` for physics-enabled platforms:

```
create()
{
    // remove this:
    // this.physics.add.image(240, 320, 'platform')
    //  .setScale(0.5)

    // create the group
    const platforms = this.physics.add.staticGroup()

    // then create 5 platforms from the group
    for (let i = 0; i < 5; ++i)
    {
        const x = Phaser.Math.Between(80, 400)
        const y = 150 * i

        /** @type {Phaser.Physics.Arcade.Sprite} */
        const platform = platforms.create(x, y, 'platform')
        platform.scale = 0.5

        /** @type {Phaser.Physics.Arcade.StaticBody} */
        const body = platform.body
        body.updateFromGameObject()
    }
}
```

First, we create a `StaticGroup` and assign it to `platforms`.

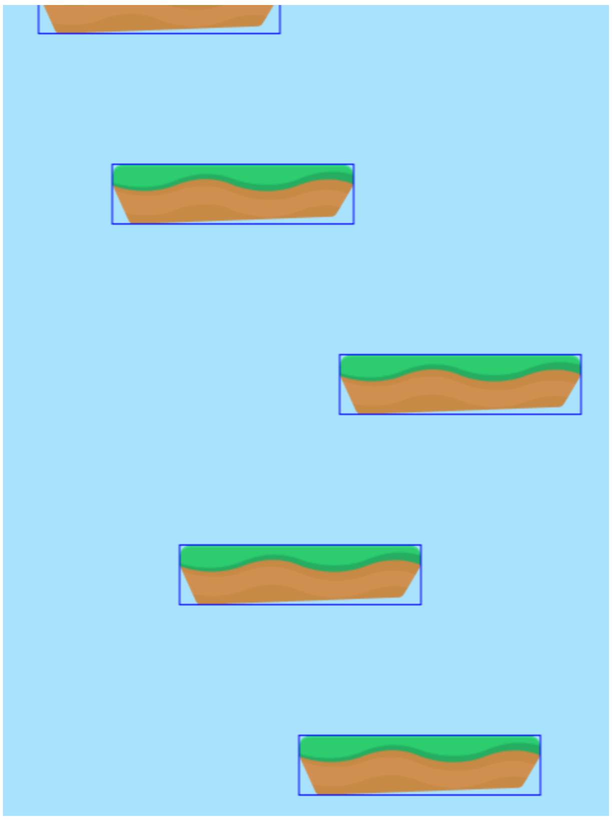
Then we use a **`for`** loop to create 5 platforms at a random `x` position between 80 and 400 and a `y` position that is 150 pixels apart.

Notice that we scale it to `0.5` like we did before.

The notation between `/**` and `*/` are JSDoc annotations. Don't worry about them for now. Just know that they help VS Code give us the right code completion.

The last thing is the call to `updateFromGameObject()`. This will refresh the physics body based on any changes we made to the `GameObject` like position and scale.

The game should look something like this:



The exact placement of the platforms will differ because we are using random x values. You can refresh the browser to see the platform positions change each time.

We have properly sized platforms that stay in place.

What we need next is something or someone to jump on them!

Creating the Player

Our player character is a bunny that will automatically hop on any platform it lands on.

To create it we need to first load the `bunny-stand.png` image in `preload()`:

```
1  preload()
2  {
3      // we added these earlier...
4      this.load.image('background', 'assets/bg_layer1.png')
5      this.load.image('platform', 'assets/ground_grass.png')
6
7      // add this new line
8      this.load.image('bunny-stand', 'assets/bunny1_stand.png')
9  }
```

This image loading code should be pretty familiar by now!

So, this might seem… silly, but try telling yourself what is happening on line 8.

You don't need to speak out loud. That would just make you look crazy! Just explain it to youself with the in-your-head voice.

Once you're done talking to yourself, create a physics Sprite to represent the player like this:

```
create()
{
    // previous platform code...

    // create a bunny sprite
    this.physics.add.sprite(240, 320, 'bunny-stand')
        .setScale(0.5)
}
```

Save your changes and a bunny should appear in the game. It will pass through all the platforms and fall off screen like it just don't care.

Adding Collisions

Arcade Physics does not assume that everything should collide with each other.

We have to tell it what things should collide with each other by creating a collider like this:

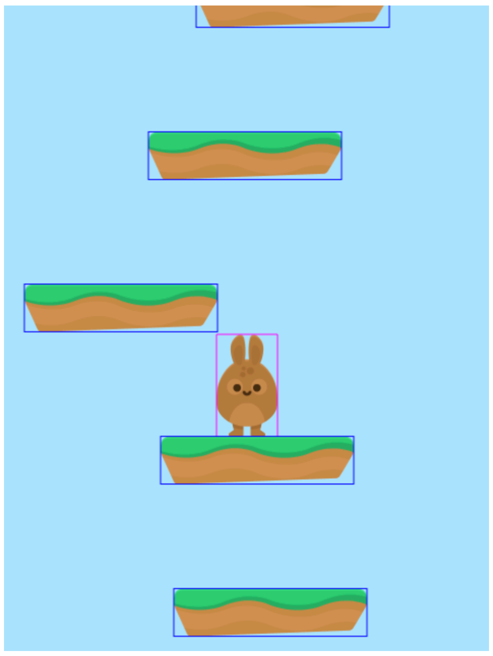
```
create()
{
    // previous platform code...

    // create a bunny sprite
    const player = this.physics.add.sprite(240, 320, 'bunny-stand')
        .setScale(0.5)

    this.physics.add.collider(platforms, player)
}
```

Notice that we created a `player` variable to hold the bunny.

Then we add a `collider` between the `platforms` created earlier and the new `player` variable.

The game should now show the bunny landing on a platform.

There's a chance that the gods did not smile upon you and randomly created all the platforms away from the center. Just refresh until there is a platform for the bunny to land on.



Using Class Properties

We have a player character but it doesn't do anything.

Let's change that by adding an automatic jump as soon as the bunny lands on a platform.

We will need to add logic to an `update()` method. You may have heard the term "update loop".

This refers to code that gets called every frame. Putting inefficient code in `update()` is usually the reason for poor performance.

You don't have to worry about any of that in this game. Just remember that anything in `update()` will be executed over and over.

So let's make some changes that will allow us to check when the bunny lands on a platform.

The check will happen in `update()` and we'll need a reference to the bunny. Currently, we only have access to the player in `create()` using the local `player` variable.

Here's what we need to change:

```
 1  export default class Game extends Phaser.Scene
 2  {
 3      /** @type {Phaser.Physics.Arcade.Sprite} */
 4      player
 5
 6      // constructor, preload ...
 7
 8      create()
 9      {
10          // platform code...
11
12          // change to use class property this.player
13          this.player = this.physics.add.sprite(240, 320, 'bunny-stand')
14              .setScale(0.5)
15
16          // same thing here in the second parameter
17          this.physics.add.collider(platforms, this.player)
18      }
19
20      update()
21      {
22      }
23  }
```

We've seen the notation on Line 3 before. It is a JSDoc annotation. JSDoc is a JavaScript documentation syntax for generating documentation.

Cool story but… we don't actually care about that.

We are using it because VS Code uses it to give type information about properties and variables.

This allows IntelliSense to continue providing help when we use the `player` property. It also helps us or someone else reading the code know that `player` is a `Phaser.Physics.Arcade.Sprite`.

Right after that on line 4 is a declaration of the `player` property that will be used as `this.player`.

You can see in `create()` that we've changed from using a local variable `player` to the class property `this.player`.

Be sure to notice the change to `this.physics.add.collider()`. This is a test to make sure you paying attention!

The last thing is a new `update()` method. Leave it blank for now.

Test the changes we just made. The game should still work like it did before.

If it is not working like it did before then make sure you are using `this.player` everywhere `player` was previously used.

Let's Jump

Now that we have `player` as a class property, we can use it in `update()` to check when the bunny has landed on a platform.

```
update()
{
    // find out from Arcade Physics if the player's physics body
    // is touching something below it
    const touchingDown = this.player.body.touching.down

    if (touchingDown)
    {
        // this makes the bunny jump straight up
        this.player.setVelocityY(-300)
    }
}
```

Save your changes and the bunny will now jump each time it lands on a platform.

It will also hit its head on platforms above it which is not what we want. The player should be able to jump up through platforms and only collide upon landing.

How should we do that?

Got any ideas?

Seriously, whatcha' got?

Anything? I know you can think of something!

Arcade Physics actually makes this quite easy! But first…

Detour into Velocity

If you know your velocity then just skip this.

Velocity is the change in position… said your physics teacher. That probably doesn't make sense now if it didn't make sense then.

So let's try this…

On line 10 in the code snippet above, we are setting the vertical velocity along the y-axis to `-300`.

This means change the `y` position value by `-300` pixels each frame. If you start at 0 then you'd be at -300 in 1 frame, -600 in 2 frames, etc.

Those paying attention are probably wondering why the bunny isn't flying off the screen then? The game height is only 640 pixels.

It should be off screen in 3 frames! And this ain't no Jazz Jackrabbit game–yes, a 90's reference for the gaming history buffs.

Recall that gravity is set to `y: 200` in `main.js`.

Arcade Physics will do all the math to adjust velocity each frame after the initial `-300` thrust.

You can adjust these numbers to change how the jump feels.

Back to our collision problem at hand. We only want collisions to happen for landing on platforms.

Here's what we need to do:

```
create()
{
    // previous code...

    this.player.body.checkCollision.up = false
    this.player.body.checkCollision.left = false
    this.player.body.checkCollision.right = false
}
```

The `Phaser.Physics.Arcade.Body` class has a `checkCollision` property where we can set which directions we want collision for.

By default, all directions are true.

For our player, we can set `up`, `left`,, and `right` to **`false`** leaving just `down` as the collision we want to check for.

The bunny should now jump up through platforms, land on them, and then jump again.

Refresh a few times to see how it works with different platform positions.

Follow that Rabbit!

The core game mechanic is working but this is supposed to be an infinite jumper.

That means we'll have to follow the bunny as it jumps up.

Phaser makes this quite easy with the `Camera`:

```
create()
{
    // previous code...

    this.cameras.main.startFollow(this.player)
}
```

That's it! Just 1 line.

Now our next problem is that we've run out of platforms to jump on.

Is this game development or a problem gauntlet?

That's a trick question. They are actually one and the same!

Next up, we will handle scrolling platforms.

Scrolling Platforms

The key part to an infinite jumper game is that platforms continuously appear as the player jumps higher and higher.

We could create new platforms as the player moves up but we can also reuse the platforms that are scrolled off screen.

Basically, we take the platforms at the bottom and move them to the top as the player jumps higher and higher.

This also adds a game over condition if the player falls past the last visible platform.

To add the smoke and mirrors of infinite platforms, we have to first create a `platforms` class property like we did with `player` in the last chapter.

We need this in order to check each platform in `update()` and move them once they are no longer visible.

```
1  export default class Game extends Phaser.Scene
2  {
3      /** @type {Phaser.Physics.Arcade.StaticGroup} */
4      platforms
5  
6      // player property...
7  
8      create()
9      {
10         // background...
11 
12         // change to use class property intead of local variable
13         this.platforms = this.physics.add.staticGroup()
14 
15         // then create 5 platforms from the group
16         for (let i = 0; i < 5; ++i)
17         {
18             const x = Phaser.Math.Between(80, 400)
19             const y = 150 * i
20 
21             // use this.platforms here as well
22             /** @type {Phaser.Physics.Arcade.Sprite} */
23             const platform = this.platforms.create(x, y, 'platform')
24             platform.scale = 0.5
25 
26             /** @type {Phaser.Physics.Arcade.StaticBody} */
27             const body = platform.body
28             body.updateFromGameObject()
29         }
30 
31         // other code...
32 
33         // and change it here too
34         this.physics.add.collider(this.platforms, this.player)
35 
36         // other code...
37     }
```

Recall that we create class properties by adding them to the class as shown on line 4.

Right above it we use JSDoc to define the type to help VS Code give us proper code completion.

Then in `create()` we change all references to the local `platforms` variable to the class property `this.platforms`.

Save your changes and make sure the game still works as expected. If not, make sure `this.platforms` is being used in place of `platforms` everywhere.

Reuse and Recycle

Now we can add logic that takes platforms from the bottom of the screen and moves them to the top.

This same concept is used in mobile apps with huge or infinitely scrolling lists.

We will implement a very rudimentary version for this game.

```
update()
{
    this.platforms.children.iterate(child => {
        /** @type {Phaser.Physics.Arcade.Sprite} */
        const platform = child

        const scrollY = this.cameras.main.scrollY
        if (platform.y >= scrollY + 700)
        {
            platform.y = scrollY - Phaser.Math.Between(50, 100)
            platform.body.updateFromGameObject()
        }
    })

    // previous code...
}
```

The above code iterates over each platform in the `this.platforms` group.

To iterate means going over each one and performing some action on them.

The action we are performing is to check if each platform's `y` value is greater than or equal to the vertical distance that the camera has scrolled plus a fixed 700 pixels.

If that is true, we move the platform to some random amount between 50 and 100 pixels above where the top of the camera is.

Then we refresh the platform's physics body to match changes made to the platform–namely the change in `y`.

Save these changes and you should see platforms continuing to appear as the bunny jumps higher.

You should also see that we quickly jump past the background as well.

That's not what we want…

Keep the Background from Scrolling

There is no real depth in a 2D game but we can fake it for the background by making sure it never moves no matter how much we scroll.

```
1  create()
2  {
3      this.add.image(240, 320, 'background')
4          .setScrollFactor(1, 0)
5
6      // previous code...
7  }
```

The addition of `.setScrollFactor(1, 0)` on line 4 is all we need.

By setting the `y` scroll factor to 0 we can keep the background from scrolling up and down with the camera.

Let's talk about the Reuse and Recycle logic above to wrap up this chapter.

It works specifically for the set up that we have.

You can try creating more platforms in `create()` to start and see how it doesn't quite work any-more.

Changing the game's height could also potentially break it.

All of this can be fixed and it is left as an exercise for the reader–that's you.

We recommend that you come back to this after you've finished the book.

Remember: game development is a problem gauntlet!

Moving Left and Right

Currently, our game is more of an infinite video than a game.

It just plays itself.

Let's change that by adding player input via the keyboard.

We need to be able to move left and right to land on platforms that are not directly below us.

Phaser gives us a very convenient way to get and use keyboard arrow keys.

```
export default class Game extends Phaser.Scene
{
    // previous code...

    /** @type {Phaser.Types.Input.Keyboard.CursorKeys} */
    cursors

    // constructor...

    preload()
    {
        // previous preload code...

        this.cursors = this.input.keyboard.createCursorKeys()
    }

    // other code...
```

We add another class property called `cursors` and use JSDoc to give it a type. This should be familiar by now!

Then at the end of the `preload()` method we set `this.cursors` to the result from `this.input.keyboard.createCursorKeys()`.

Note that we can also set `this.cursors` in `create()`. Where you want to have it is a matter of style or specific implementation requirements.

In this case, it is entirely a matter of style.

Handling Move Logic

Now that we have the arrow or cursor keys, we can check when they are pressed and change the player's velocity.

We will do this in the `update()` method:

```
update(t, dt)
{
    // platform reuse code...

    const touchingDown = this.player.body.touching.down

    if (touchingDown)
    {
        this.player.setVelocityY(-300)
    }

    // left and right input logic
    if (this.cursors.left.isDown && !touchingDown)
    {
        this.player.setVelocityX(-200)
    }
    else if (this.cursors.right.isDown && !touchingDown)
    {
        this.player.setVelocityX(200)
    }
    else
    {
        // stop movement if not left or right
        this.player.setVelocityX(0)
    }
}
```

We can check if the left arrow key is pressed using `this.cursors.left.isDown`. The same is done for the right arrow key with `this.cursors.right.isDown`.

If the keys are pressed then we set the `x` velocity to `-200` for left and `200` for right.

When nothing is pressed we set the `x` velocity to `0` to stop horizontal movement.

We also only allow moving left and right when the bunny is **not** touching a platform as determined by `touchingDown`.

The game should now let you move the bunny left and right when it is in the air.

Give it a try!

You'll notice that the camera follows you left and right. That is not what we want.

The camera should not scroll horizontally and the player should wrap around when it passes the left and right sides.

Using Camera Dead Zones

We can stop the camera from scrolling horizontally with a wide dead zone.

A dead zone is an area around the player where the camera won't scroll.

You've likely seen this in many games that let you run around without giving you motion sickness from all the camera movement.

Here's how we set a horizontal dead zone in our game:

```
create()
{
    // previous code...

    this.cameras.main.startFollow(this.player)

    // set the horizontal dead zone to 1.5x game width
    this.cameras.main.setDeadzone(this.scale.width * 1.5)
}
```

First thing to note is that we are using the Phaser `ScaleManager` to get the width of the game with `this.scale.width`.

This is the recommended way of getting the game's width and height instead of using fixed numbers.

We are multiplying by `1.5` to give us room to implement wrapping the player around the screen later.

To actually set the dead zone we use `this.camera.main.setDeadzone()`.

Easy peasy!

Save your changes and the game should no longer scroll left and right unless you move way off screen.

We'll fix that next with a little screen wrapping!

Horizontal Wrap Around

The intended behavior for an infinite jumper is to have the player wrap around the screen when they go past the left and right sides.

Arcade Physics has a wrap() method in the World but it also wraps vertically.

This is perfect for a game like Asteroids but not so much for this game.

The good news is that code for a horizontal screen wrap is mighty simple!

```
export default class Game extends Phaser.Scene
{
    // previous class code...

    update()
    {
        // previous code...

        this.horizontalWrap(this.player)
    }

    /**
     * @param {Phaser.GameObjects.Sprite} sprite
     */
    horizontalWrap(sprite)
    {
        const halfWidth = sprite.displayWidth * 0.5
        const gameWidth = this.scale.width
        if (sprite.x < -halfWidth)
        {
            sprite.x = gameWidth + halfWidth
        }
        else if (sprite.x > gameWidth + halfWidth)
        {
            sprite.x = -halfWidth
        }
    }
}
```

The horizontalWrap(sprite) method is where the screen wrapping logic is.

The annotation above the method declaration is more JSDoc. This time it is defining the type of the sprite parameter to be a Phaser.GameObjects.Sprite.

We will only be wrapping the player but this method can wrap any Sprite.

You should generally aspire to write methods that are more like this and have less hidden internal dependencies. If that just sounded way too confusing then don't worry about it!

Just remember that generic code is more reusable and reusable code helps us make better games more quickly.

The logic in horizontalWrap is fairly straight forward. If the passed in sprite goes past the left side more than half its width then teleport it to the right side plus half its width.

Then do the reverse when it goes past the right side more than half its width.

We call `this.horizontalWrap` at the end of `update()` and pass in `this.player` as the `sprite`.

Give it a try and you should see that the camera no longer scrolls horizontally at all and the player will wrap around the screen like a happy bunny!

Collecting Carrots

Every good game needs something to collect. And bunnies love carrots.

So let's give our bunny some carrots to munch on.

This time we are going to create a separate class for our carrot and import it into the `Game` Scene.

To start, create a new folder named `game` under `src`. It should be on the same level as `scenes` and `lib`.

Then create a new file named `Carrot.js` in the `game` folder.

The `Carrot` class code will simply look like this:

```
 1  import Phaser from '../lib/phaser.js'
 2
 3  export default class Carrot extends Phaser.GameObjects.Sprite
 4  {
 5      /**
 6       * @param {Phaser.Scene} scene
 7       * @param {number} x
 8       * @param {number} y
 9       * @param {string} texture
10       */
11      constructor(scene, x, y, texture)
12      {
13          super(scene, x, y, texture)
14
15          this.setScale(0.5)
16      }
17  }
```

First, we import `Phaser` from `../lib/phaser.js` because we will be extending from `Phaser.GameObjects.Sprite`.

The `Carrot` class declaration comes on line 3 and it is the default export.

This is a modern JavaScript feature meaning that we can later import `Carrot` like this:

```
1  import Carrot from './Carrot.js`
```

The class body is just a `constructor` that takes in a `scene` reference and then an `x` and `y` position.

Then the last parameter `texture` is the key for the loaded image to use.

The line with `super()` is to call the constructor of the parent class: `Phaser.GameObjects.Sprite`. We must do this so that initialization code in the `Sprite` class is executed.

We aren't doing any work with the `x`, `y`, or `texture` values but someone has to in order to create a `Sprite`. Nothing is magic!

That someone doing the work is the `Sprite` class and calling `super()` with the given parameters allows it to do its work.

Lastly, we set the scale to `0.5` because we know that all the assets are twice as big as we want.

In the `Game` Scene we've previously called `setScale` after creating platforms and the player.

One benefit of using classes is that we can hide these types of internal configuration details–some would argue it is a negative. The more zen among us understand it is both.

But anyway…

Adding Carrots to the Game

Now that we have a `Carrot` class, let's load the `carrot.png` image and create some carrots!

First, in `preload()`:

```
preload()
{
    // load other images..

    this.load.image('carrot', 'assets/carrot.png')

    // cursor keys...
}
```

You should understand what is going on here pretty well by now.

Next, we need to import the `Carrot` class.

```
import Phaser from '../lib/phaser.js'

// import the Carrot class here
import Carrot from '../game/Carrot.js'

export default class Game extends Phaser.Scene
{
    // class body...
}
```

Adding line 4 is what will allow us to create new `Carrot` class instances.

So let's do that in `create()`:

```
create()
{
    // previous code...

    // create a carrot
    const carrot = new Carrot(this, 240, 320, 'carrot')
    this.add.existing(carrot)
}
```

This will show a carrot in the middle of the game. It is just a `Sprite` so nothing will interact with it.

That's a problem since we want to be able to collect carrots and we want to be able to collect many of them.

Hmmm…

What systems have we already used that lets us do these things?

Take a guess.

Did you guess?

It's Arcade Physics and groups!

Create a Group of Physics-Enabled Carrots

Let's remove the carrot code we just added to `create()`. We are going to replace it with much better code shortly.

First, we want a `Group` that creates `Carrot` instances.

```
 1  export default class Game extends Phaser.Scene
 2  {
 3      // previous properties...
 4
 5      /** @type {Phaser.Physics.Arcade.Group} */
 6      carrots
 7
 8      create()
 9      {
10          // previous code...
11
12          this.carrots = this.physics.add.group({
13              classType: Carrot
14          })
15
16          this.carrots.get(240, 320, 'carrot')
17      }
18  }
```

Just like we did with `platforms`, create a `carrots` class property like we do on line 6.

Then at the bottom of `create()` we add a physics group and pass in a configuration object specifying `classType` as `Carrot`.

The physics group will create `Phaser.Physics.Arcade.Sprite` instances by default. Specifying `classType` lets us change that.

Lastly, we create a test `Carrot` in the middle of the screen.

Save your changes and you should see a carrot appear and then fall to its doom.

Clean up Detour

We are going to do a very minor clean up before continuing.

Our `Carrot` class extends from `Phaser.GameObjects.Sprite` but since we are using it in a physics group we know that it will really be a `Phaser.Physics.Arcade.Sprite`.

Nothing in your code will break if you don't make this change. This is just a matter of tidiness and accuracy.

The `Carrot` class declaration should look like this:

```
export default class Carrot extends Phaser.Physics.Arcade.Sprite
{
    // class body...
}
```

Put Carrots on Platforms

We can have floating carrots but pigs will probably get wings and hover long before carrots.

Instead of floating let's put our carrots on top of platforms to give them a place to stay.

To start, let's add a collider between platforms and carrots lke this:

```
create()
{
    // previous code...

    // remove this test carrot
    // this.carrots.get(240, 320, 'carrot')

    // add this collider
    this.physics.add.collider(this.platforms, this.carrots)
}
```

Next, let's add an `addCarrotAbove(sprite)` method that will put a `Carrot` above the given `sprite`.

We are using a generic `sprite` for reusable code reasons we discussed in the last chapter for `horizontalWrap(sprite)`.

Put this at the bottom of the `Game` Scene:

```
 1  export default class Game extends Phaser.Scene
 2  {
 3      /**
 4       * @param {Phaser.GameObjects.Sprite} sprite
 5       */
 6      addCarrotAbove(sprite)
 7      {
 8          const y = sprite.y - sprite.displayHeight
 9  
10          /** @type {Phaser.Physics.Arcade.Sprite} */
11          const carrot = this.carrots.get(sprite.x, y, 'carrot')
12  
13          this.add.existing(carrot)
14  
15          // update the physics body size
16          carrot.body.setSize(carrot.width, carrot.height)
17  
18          return carrot
19      }
20  }
```

This code should be familiar as it is a combination of things we've done before.

The `carrot` instance is positioned above the given `sprite` using its `displayHeight` as a guide.

A more accurate formula is to subtract half of `sprite.displayHeight` and half of the carrot's `displayHeight` from `sprite.y`.

The much simpler formula we are using is fine for this game. You might have to switch to the more accurate version if you try using different assets.

The one new piece of code is on line 16 where we resize the physics body to the width and height of the `carrot`.

Test it with and without this line to see the difference.

Lastly, we can call this method when we reuse a plaform:

```
update(t, dt)
{
    this.platforms.children.iterate(child => {
        /** @type {Phaser.Physics.Arcade.Sprite} */
        const platform = child

        const scrollY = this.cameras.main.scrollY
        if (platform.y >= scrollY + 700)
        {
            platform.y = scrollY - Phaser.Math.Between(50, 100)
            platform.body.updateFromGameObject()

            // create a carrot above the platform being reused
            this.addCarrotAbove(platform)
        }
    })

    // other code...
}
```

`this.addCarrotAbove()` is called after we move the platform to the top of the screen.

Give it a try! You should see carrots appear on top of platforms after you've jumped up a few.

But you still can't collect them…

Let's Collect Some Carrots

Now that our carrots are on platforms, we can finally add logic to collect them.

We will be using an overlap instead of a collider to collect carrots.

Add this to the end of `create()`:

```
create()
{
    // previous code...

    this.physics.add.collider(this.platforms, this.carrots)

    // formatted this way to make it easier to read
    this.physics.add.overlap(
        this.player,
        this.carrots,
        this.handleCollectCarrot,    // called on overlap
        undefined,
        this
    )
}
```

See that we add an overlap with `this.physics.add.overlap()`.

We've put each passed in parameter on a different line for better book formatting. You can put it on a single line on your computer.

We are testing for an overlap between the `carrots` group and the `player`.

And then when an overlap occurs, the `handleCollectCarrot` method is called.

The `undefined` argument is for the process callback that we don't need and lastly, `this` is passed in as the context–which we do need and is important.

Providing the right context here is important to ensure that `this` is the `Game` Scene instance when the `handleCollectCarrot` method is called by Phaser.

If that made no sense to you then don't worry about it! By using modern JavaScript with Phaser you will almost always want to pass in `this` as the context anyway.

So just do that. You can learn more about JavaScript scope and context later.

Now let's add the `handleCollectCarrot` method to the bottom of the `Game` Scene:

```
export default class Game extends Phaser.Scene
{
    // previous code...

    /**
     * @param {Phaser.Physics.Arcade.Sprite} player
     * @param {Carrot} carrot
     */
    handleCollectCarrot(player, carrot)
    {
        // hide from display
        this.carrots.killAndHide(carrot)

        // disable from physics world
        this.physics.world.disableBody(carrot.body)
    }
}
```

We are given the `carrot` that was overlapped by the `player` in this method.

The code deactiviates and hides the `carrot` by calling the group's `killAndHide()` method and passing in the `carrot` insance.

An important step comes next in disabling the physics body from the world.

Save your changes and try it out. Carrots should disappear once you run into them.

You should also notice that they don't reappear when a carrot gets reused!

Game development is a problem gauntlet, isn't it?

Reusing Carrots

By creating a `carrots` group and calling `killAndHide()` we are basically half way towards reusing carrots.

Phaser Groups will automatically recycle any inactive members when `get()` is called like we do in `addCarrotAbove()`.

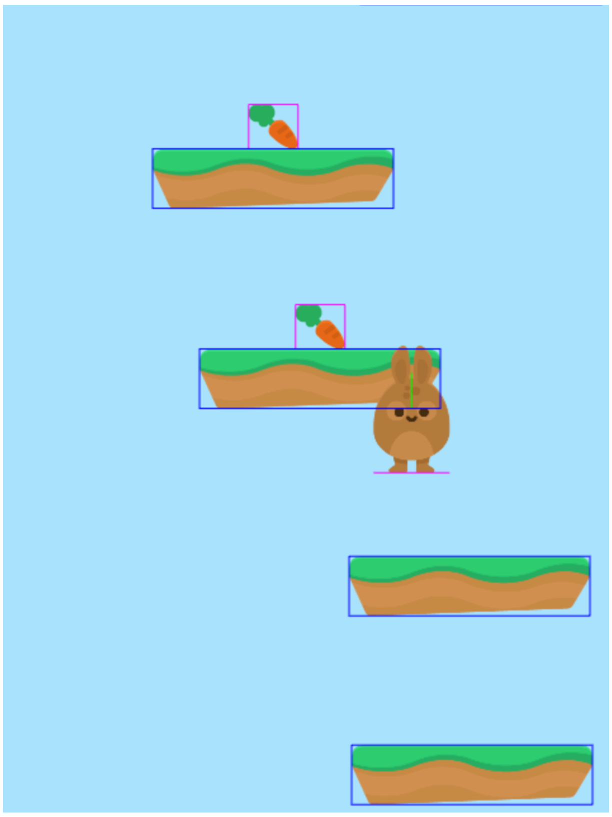
It doesn't automatically reactive or make them visible so we have to do that in `addCarrotAbove()` :

```
 1  /**
 2   * @param {Phaser.GameObjects.Sprite} sprite
 3   */
 4  addCarrotAbove(sprite)
 5  {
 6      const y = sprite.y - sprite.displayHeight
 7
 8      /** @type {Phaser.Physics.Arcade.Sprite} */
 9      const carrot = this.carrots.get(sprite.x, y, 'carrot')
10
11      // set active and visible
12      carrot.setActive(true)
13      carrot.setVisible(true)
14
15      this.add.existing(carrot)
16
17      carrot.body.setSize(carrot.width, carrot.height)
18
19      // make sure body is enabed in the physics world
20      this.physics.world.enable(carrot)
21
22      return carrot
23  }
```

Line 12 and 13 is where we make sure carrots are active and visible.

Then on line 20 we make sure the `body` is enabled in the physics world.

Try it now and everything should work as expected!



There is one problem though… no one is keeping of track of how many carrots we've collected!

Let's tackle that next.

There's actually another problem! But we will leave it for you to solve.

Any carrots that are not collected by the player will eventually scroll off screen and then continue falling forever as the platforms get recycled.

This could lead to a lot of orphaned carrots. Odds are you'll never notice that this is a problem.

But it is unintended behavior and may come back to bite us in the behind later.

The good news: you actually know how to solve this problem!

Just apply what we did with reusing platforms.

Keeping Track of the Score

We've got a bunny that can collect carrots but we need to know how many carrots have been collected.

The higher you jump the more carrots you would have collected.

Number of carrots collected is how we'll determine a score!

First, we need to create a class property to hold the number of carrots collected:

```
export default class Game extends Phaser.Scene
{
    carrotsCollected = 0

    // previous code...
}
```

This is the simplest class property we've created yet! It is just a number so we don't need any JSDoc annotations.

Next in the `handleCollectCarrot()` method we need to increment `carrotsCollected`:

```
/**
 * @param {Phaser.Physics.Arcade.Sprite} player
 * @param {Carrot} carrot
*/
handleCollectCarrot(player, carrot)
{
    this.carrots.killAndHide(carrot)

    this.physics.world.disableBody(carrot.body)

    // increment by 1
    this.carrotsCollected++
}
```

We use the ++ shorthand to increment by 1 but you can also do this:

```
this.carrotsCollected = this.carrotsCollected + 1
```

Next, we need to display the number. Phaser lets us make text pretty easily so let's add one in `create()`:

```
create()
{
    // previous code...

    const style = { color: '#000', fontSize: 24 }
    this.add.text(240, 10, 'Carrots: 0', style)
        .setScrollFactor(0)
        .setOrigin(0.5, 0)
}
```

We add a `Text` object with `this.add.text` and place it in the middle at `240` on the x-axis and `10` on the y-axis.

The initial text is set to `'Carrots: 0'` and then a `TextStyle` object is given that sets the color to black and size to 24 pixels.

We've used `setScrollFactor()` before to stop the background from scrolling with the camera. We use it here in the same way to disable scrolling.

Then we use `setOrigin(0.5, 0)` to keep the text top-centered. This is also known as the anchor or pivot point.

Try out these changes. You should see text at the top of the screen.

Updating Text on Carrot Collected

The text is there but you'll notice that it doesn't change when we collect a carrot.

It can fixed by creating a class property to store the `Text` instance and then updating the text in `handleCollectCarrot` after we increment `carrotsCollected`.

Creating a new class property should be second nature by now!

```
 1  export default class Game extends Phaser.Scene
 2  {
 3      // other properties...
 4
 5      /** @type {Phaser.GameObjects.Text} */
 6      carrotsCollectedText
 7
 8      create()
 9      {
10          // previous code...
11
12          const style = { color: '#000', fontSize: 24 }
13          this.carrotsCollectedText = this.add.text(240, 10, 'Carrots: 0'
               , style)
14              .setScrollFactor(0)
15              .setOrigin(0.5, 0)
16      }
17
18      // other code...
19
20      /**
21       * @param {Phaser.Physics.Arcade.Sprite} player
22       * @param {Carrot} carrot
23      */
24      handleCollectCarrot(player, carrot)
25      {
26          this.carrots.killAndHide(carrot)
27
28          this.physics.world.disableBody(carrot.body)
29
30          // increment by 1
31          this.carrotsCollected++
32
33          // create new text value and set it
34          const value = `Carrots: ${this.carrotsCollected}`
35          this.carrotsCollectedText.text = value
36      }
37  }
```

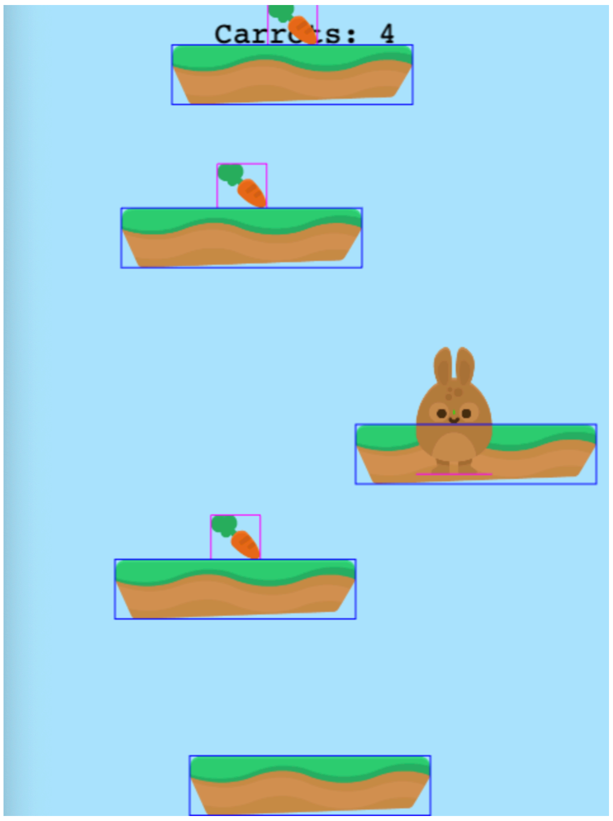
The `carrotsCollectText` class property is created at the top and then assigned the created `Text` instance in `create()`.

With a reference to the `Text` instance in hand we can change the display text in `handleCollectCarrot ()`.

Line 34 uses JavaScript's string interpolation syntax to create a new text value by inserting the value of `this.carrotsCollected` next to `'Carrots: '`.

We then set this new value into `this.carrotsCollectedText.text` to update the displayed text.

Save your changes and give it a try! You should now know exactly how many carrots you've collected as you jump higher and higher!



Our game is almost complete!

Next, we need to handle the player losing when they fall past the bottom most platform.

Handling Game Over

The game over condition for this game is when the player falls past the last platform that can be landed on and continues falling forever into the abyss.

Our job is to detect when this has happened.

The key is to figure out which platform is the bottom most platform and then check that the player has gone past it.

How would you do that?

What steps would you take to figure out which platform in the platforms group is the last one visually?

The steps and process you define to figure this out is called an algorithm.

Computer scientists and engineers have developed or discovered many over the years including one to solve this exact problem!

We'll be using the **selection algorithm** to find the platform with the highest `y` value to determine the bottom most platform.

Let's add a `findBottomMostPlatform()` method to the `Game` Scene:

```
export default class Game extends Phaser.Scene
{
    // previous code...

    findBottomMostPlatform()
    {
        const platforms = this.platforms.getChildren()
        let bottomPlatform = platforms[0]

        for (let i = 1; i < platforms.length; ++i)
        {
            const platform = platforms[i]

            // discard any platforms that are above current
            if (platform.y < bottomPlatform.y)
            {
                continue
            }

            bottomPlatform = platform
        }

        return bottomPlatform
    }
}
```

We start by getting all the platforms as an Array with `this.platforms.getChildren()`.

Then we pick the first one in the Array as the current bottom most platform.

Next, we iterate over the Array and compare each platform against the current `bottomPlatform`. If a platform's `y` position is greater than the `bottomPlatform` then we set it as the new `bottomPlatform`
.

Once we've iteracted over the entire Array, the last platform stored in `bottomPlatform` is the bottom most platform and gets returned.

Now to use the returned value we add this to the bottom of `update()`:

```
update()
{
    // previous code...

    const bottomPlatform = this.findBottomMostPlatform()
    if (this.player.y > bottomPlatform.y + 200)
    {
        console.log('game over')
    }
}
```

First, we get the bottom most platform by calling `this.findBottomMostPlatform()` and store it in `bottomPlatform`.

Then we check that the player is more than 200 pixels past the `bottomPlatform` before logging `'game over'` to the browser Console.

Give this a try and see if the message appears in the Console.

We will create a Game Over Scene next.

Game Over Scene

Just logging the words `'game over'` to the browser Console is not good enough.

The player won't see it! How often do you look at the browser Console?

First, create a new `GameOver.js` file in the `scenes` folder.

```
1  import Phaser from '../lib/phaser.js'
2
3  export default class GameOver extends Phaser.Scene
4  {
5      constructor()
6      {
7          super('game-over')
8      }
9
10     create()
11     {
12
13     }
14 }
```

All of this should look similar to what we did to create the `Game` Scene.

Remember that every Scene needs a unique key. We are providing that in the call to `super()` on line 7.

Next, let's add a simple "Game Over" message in the `create()` method:

```
create()
{
    const width = this.scale.width
    const height = this.scale.height

    this.add.text(width * 0.5, height * 0.5, 'Game Over', {
        fontSize: 48
    })
    .setOrigin(0.5)
}
```

We are using the `ScaleManager` to get the width and height of the game instead of hard coding numbers.

Then we create a simple `Text` object like we've done before and set the origin to `0.5` so that it is centered vertically and horizontally.

Before we can use this Scene, we need to add it to the game configuration.

Head over to `main.js` and update the configuration's `scene` property like this:

```javascript
// other imports...

// import the GameOver Scene
import GameOver from './scenes/GameOver.js'

export default new Phaser.Game({
    type: Phaser.AUTO,
    width: 480,
    height: 640,

    // previously...
    // scene: Game,

    // now it should be this
    scene: [Game, GameOver],

    // physics...
})
```

The first thing we do is import the new `GameOver` scene like we did for the `Game` Scene.

Then we change the `scene` property from just taking the `Game` class to take an Array of `Game` and `GameOver` classes.

Now we can use the `GameOver` Scene when the game over condition is met.

In the `Game` Scene change the `console.log('game over')` line to this:

```
update()
{
    // previous code...

    const bottomPlatform = this.findBottomMostPlatform()
    if (this.player.y > bottomPlatform.y + 200)
    {
        // remove this...
        // console.log('game over')

        // add this...
        this.scene.start('game-over')
    }
}
```

Save changes to all files and then give it a try. You should get sent to the the `GameOver` Scene once you fall past the bottom most platform!

Add Play Again

We now have a `GameOver` Scene but we just get stuck there.

There's no way to play again. How lame.

So let's change that by going back to the `Game` Scene when we hit the space bar from the `GameOver` Scene.

Add this to `create()` in `GameOver.js`:

```
create()
{
    // previous code...

    this.input.keyboard.once('keydown_SPACE', () => {
        this.scene.start('game')
    })
}
```

We use the `InputManager` to listen for when the space key is pressed and then start the `Game` Scene.

Notice that we use `.once` instead `.on`. This saves us from having to clean up the event since it will be automatically cleaned up after the event is fired once. That's how it got its the name.

Give it a try! We now have a game that we can play over and over!

But wait…

You didn't think we were out of problems did you?

Game development is a problem gauntlet, remember?

Play the game a little bit and you'll notice that the carrots collected counter is not working right after restarting from the `GameOver` Scene.

It will always show `Carrots: 0` but sometimes it just collects a bunch when the first carrot is collected.

What is going on here?

Look friend, there's no easy way to tell you this but… you've got a bug!

The problem is that we are not reseting `carrotsCollected` when we leave and restart the `Game` Scene.

Initializing Scene Properties

Most of the `Game` Scene's class properties are set or reset in `create()`.

All but `carrotsCollected`. We can just do `this.carrotsCollected = 0` in `create()` but there's another Scene hook you can use:

```
export default class Game extends Phaser.Scene
{
    // properties...

    // contructor...

    init()
    {
        this.carrotsCollected = 0
    }

    // other code...
}
```

The `init()` method is called by Phaser before `preload()`. It is a great place for initialization as the name would suggest.

Add the `init()` method like we have above to your code and give it a try.

Now everything should be working properly!

We are almost done with this game.

There's just one thing we want to show you.

And what's a book with 9 chapters? Let's make it a round 10.

We also have 1 more art asset that we haven't used yet.

The next chapter will wrap up by using the `bunny1_jump.png` image to add a bit of animation.

Simple Animation with a Jump Texture

This will not be a *real* animation in the traditional sense with a spritesheet.

We will simply switch the active texture of the bunny when a jump happens and switch it back when the bunny starts falling.

First, let's preload the `bunny1_jump.png` like we've done with other images in the `Game` Scene:

```
preload()
{
    // other images...

    this.load.image('bunny-jump', 'assets/bunny1_jump.png')

    // cursor keys...
}
```

You should have no problem understanding what's happening here.

Next, in `update()` where we set the `y` velocity to `-300` we want to switch to using this jump texture:

```
update()
{
    // previous code...

    const touchingDown = this.player.body.touching.down

    if (touchingDown)
    {
        this.player.setVelocityY(-300)

        // switch to jump texture
        this.player.setTexture('bunny-jump')
    }

    // other code...
}
```

We use `this.player.setTexture('bunny-jump')` to switch to the jump texture that we preloaded.

Save your changes and check it out in the game. The bunny should start in standing position until it jumps.

Looks a bit better doesn't it?

One problem though… it never changes back!

Switch Back to Stand

We want to switch back to the standing image once the bunny starts falling.

How do we detect when it is no longer rising from a jump? And is instead being pulled down by gravity?

What do you think?

Maybe it has something to do with velocity?

Indeed it does!

We just need to check that the `y` velocity is great than `0` to know that the bunny is falling and not rising.

Add this to `update()` under the code from the example above:

```
update()
{
    // previous code...

    const touchingDown = this.player.body.touching.down

    if (touchingDown)
    {
        this.player.setVelocityY(-300)

        // switch to jump texture
        this.player.setTexture('bunny-jump')
    }

    const vy = this.player.body.velocity.y
    if (vy > 0 && this.player.texture.key !== 'bunny-stand')
    {
        // switch back to jump when falling
        this.player.setTexture('bunny-stand')
    }

    // other code...
}
```

We get the `y` velocity by accessing the `velocity` property of the player's physics body.

Then we check that it is greater than `0` and that the player's current texture is not already `'bunny-stand'`.

If that's true then we change the texture back to the standing image.

Test it out and see how it looks!

Feels much more lively doesn't it?

Just One More Thing...

We've got just one more thing to show you and that is adding a sound effect for jumping.

The jump sound we're using is from Kenney's Digital Audio pack.

Just go to http://kenney.nl, click on Assets, then select Audio, and find Digial Audio.

He has a bunch of other sound effects as well that you can use in this game or others.

The audio files will come as `ogg` files and that will be fine if you are using Windows.

If you are on a Mac then you'll need to convert it to an `mp3` or `wav`.

Find the `phaseJump1.ogg` file from the Digital Audio pack and copy it into an `sfx` folder under `assets` where the images are.

We can load audio in a similar way to images. Add this to `preload()` in the `Game` Scene:

```
preload()
{
    // images...

    this.load.audio('jump', 'assets/sfx/phaseJump1.ogg')

    // cursor keys...
}
```

Just like loading images, we give `this.load.audio()` a key as the first parameter and the path to the audio file as the second.

Now to play the sound we can call `this.sound.play()` each time the bunny jumps:

```
 1  update()
 2  {
 3      // previous code...
 4
 5      const touchingDown = this.player.body.touching.down
 6
 7      if (touchingDown)
 8      {
 9          this.player.setVelocityY(-300)
10
11          // switch to jump texture
12          this.player.setTexture('bunny-jump')
13
14          // play jump sound
15          this.sound.play('jump')
16      }
17
18      // other code...
19  }
```

That simple piece of code on line 15 is all you need.

Save your changes and give it a try. Make sure your volume is not at 0.

You'll be greeted with a nice jump sound each time the bunny jumps.

Alright, we are truly finished now.

Congratulations! You've learned a whole bunch for making games in modern JavaScript with Phaser 3.

There is, at least, one more problem with this game that we will leave for you to solve.

Bunny Jump is an *infinite* jumper and it will appear infinite for 99.9% of players.

The problem for .01% of players is that number values in JavaScript or any programming language have a maximum value. They cannot hold an infinite value.

The player increases the `y` value with every jump. This can go on until the maximum value is passed and then unexpected things happen.

What ideas can you come up to solve this problem?

How can we make something finite act like it was infinite?

Game development is problem solving gauntlet.

Good luck!

There is always more to learn so keep it tuned to Ourcade!

Check out the blog, follow us on Twitter, see our code on Github, and subscribe to us on YouTube!

Epilogue

Thank you for reading this book!

I hope that it has taught you something about game development.

The game we've created isn't super fun. It doesn't get any harder as you play.

That's where you come in.

Games are a creative endeavor. Think of all the skills you need to learn before realizing the creative vision as the 12 Labors of Hercules.

It might sound daunting but that's just the reality. You'd rather go in knowing it is hard and having it end up being easier than the reverse.

No one will have the exact answer you need. It will be a journey of discovery.

The infinite jumper we've created in this book is just a starting point.

You'll have to inject your own personality, creativity, and perseverance to take it over the finish line.

What kind of spacing makes jumping fun? Should there be enemies? What should they do? What if some platforms break when landed on? Would teleporting be fun? What if we add portals? Can we make this multiplayer? In real-time?

The possibilities are literally limitless.

And anything you want to do, you can learn. Trust me. The complexities of game development looked impossible to me when I started. The plethora of information and game engines didn't exist then.

The black magic of game development lived mostly in the minds of developers. Books on the topic were scarce. Books for beginners were even rarer.

Now we have great information all over the Internet! So much code is open source and available for anyone to read, learn from, or just steal.

You can find fantastic game assets like the ones we've used from Kenney and elsewhere–just use Google.

There has never been a better time to make games.

Our goal at Ourcade is to create the most approachable community for game development whether you are a beginner, expert, or anything in between.

No matter what our skill levels are, there's no reason not to be pleasant and enjoy the process.

Don't miss the forest for the trees: we're making games, it should be fun.

That's why Ourcade has 5 pillars:

- Playful
- Friendly
- Helpful
- Open-minded
- Optimistic

Come join us if you agree!

You can interact with us on Twitter @ourcadehq.

Our YouTube Channel strives to show the raw experience of game development without editing away the problem solving. Superhuman developers who can write perfect code in one try do not exist.

Github is where we will share source code for anyone to read, learn, or steal. We don't believe that programming code should be protected like nuclear launch codes.

And finally the Ourcade blog and website is the central hub. You'll find everything we're teaching, sharing, and building there.

Best of luck with your game making,

Tommy Leung (aka supertommy)

Colorado, USA
April 2020

About the Author

Tommy Leung, aka supertommy, has been making games professionally since 2007. He is a self-taught programmer with a business degree from Pace University.

He has worked on games for brands like Cartoon Network, Nickelodeon, LEGO, and more.

Before founding Ourcade, he was Senior Engineering Manager at Zynga, the makers of Words with Friends and FarmVille, for a hit mobile game played by tens of millions of people.

Tommy has developed games in Flash, C/C++, Objective-C, Java, Unity/C# HTML5/JavaScript, and more.

Other than game development, he also enjoys fullstack web development and native iOS development.

He believes in continuous learning, self improvement, and giving back by helping others because he would not be where he is without the generocity of those who came before him.

Tommy is also the founder of Captivate Games and an avid weight lifter.

He resides in Colorado, USA and has 2 greyhounds.