Infinite Runner in Phaser 3 with TypeScript

Tommy Leung

Contents

Introduction

Thank you for downloading this book!

We will be going through 11 chapters of game development in Phaser 3 using TypeScript to create an infinite runner like Jetpack Joyride.

Our hero will even have a jetpack!

This book is intended for developers who have dabbled in Phaser 3 by making at least one game or by going through the Infinite Jumper in Phaser 3 with Modern JavaScript book.

We won't be going through the language basics of TypeScript but you can use the TypeScript Handbook for that.

Instead, we will take you through a step-by-step process of making a Phaser 3 game using TypeScript instead of JavaScript.

Phaser 3 comes with official TypeScript support but the TypeScript experience is sometimes wonky and even confusing for those new to the framework or language.

There are certain situations that may be awkward or don't fit cleanly with TypeScript expectations but are completely normal in the greater JavaScript ecosystem.

We'll show you ways around those issues while maintaining the static analysis and other features that make TypeScript so useful and popular in larger codebases.

Why TypeScript?

TypeScript was first released in 2012 as an open-source project by Microsoft. It has slowly gained popularity over the years and appears to be really taking off as of 2019.

The fact that more and more professional teams and open-source projects are using TypeScript is a sure sign that we should be paying attention.

Phaser 4, currently in early development, is one of these modern projects being written with TypeScript.

There are clear signs of momentum but if that isn't enough for you then let's look at the benefits.

TypeScript gives you access to modern language features currently being designed into JavaScript without having to wait for browsers to implement support or users to upgrade devices.

Features like `private` or `protected` access modifiers, `async`/`await`, optional chaining, generics, and more that exist in modern languages like C#, Swift, or Kotlin.

There's also type-safety that lets you write code that is easier to maintain, update, and used by others–including a future version of yourself!

If you've ever had the experience of opening up an old JavaScript project to make a change only to spend hours trying to figure out what the data being passed around looks then TypeScript could be your new best friend.

TypeScript's use static types help solve this problem by providing information about what the data is or how it is structured without having to rely on having comments or trusting that they are up to date.

There are more benefits to using TypeScript for making games in Phaser 3 but we won't list them all.

The fact that you picked up this book probably means you are already sold on TypeScript so let's start using it!

How to Use this Book

This book is designed 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 increases as you progress.

The goal is to build a working game that you can feel comfortable modifying and adding features on your own.

Example Source Code

You can download the full source code of the infinite runner game we'll be making on Github at:

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

The repository uses `git-lfs` so be sure to have it installed before you use `git clone` to pull down the image assets as well.

Alternatively, you can just download a `zip` from the releases tab.

We intend to improve 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 make a game, 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

This book will go over making an infinite runner game like Jetpack Joyride called Rocket Mouse. The goal is to keep our jetpack-wearing mouse alive for as long as possible collecting coins and avoiding obstacles.

We recommend that you check out our Infinite Jumper in Phaser 3 with Modern JavaScript book if you are new to Phaser 3 and web development.

This book will be more advanced and use common web development tools like Node.js, the Node Package Manager (npm), and the command line.

As the title of the book suggests, we will be using TypeScript to help write code that is easier to reason about and maintain over months or years.

Installing Node.js and NPM

The simplest way to get Node.js is to install it from https://nodejs.org.

There are other ways to install it as well but we won't go into those options here.

If you already have Node.js and `npm` installed then you can skip this section. Check out this blog post instead.

Once you get to the Node.js download page, select the version recommended for most users. At the time of this writing it is the option on the left for v12.16.x.

The exact version does not matter for what we'll be doing.

Download the installer and follow the on-screen instructions to install Node.js on your computer.

Then open a terminal or command prompt and run this command:

```
node --version
```

It will respond with a version of Node.js if installed properly. This should mean that the Node Package Manager or `npm` is also installed.

Let's check that by running this command:

```
npm --version
```

This will also respond with a version. Note that the versions for Node.js and `npm` will not match each other and that is to be expected.

Creating the project

We will be using the `phaser3-typescript-parcel-template` to bootstrap our project.

You can find it on Github here.

If you are familiar with `git` then feel free to clone it.

If not, you can download it as a zip by clicking the green "Clone or download" button.

Feel free to change the name of the folder from 'phaser3-typescript-parcel-template' to something like 'rocket-mouse-game'.

This project template uses Parcel as the application bundler and development web server.

If you've heard of `webpack` or `rollup` then `parcel` is similar to those tools except it prides itself on being zero-config to start and blazingly fast.

You'll need to have Parcel installed so open a command line window and run this command:

```
npm install -g parcel-bundler
```

To confirm that Parcel was installed successfully, run this command:

```
parcel --version
```

It will respond with a version number like `1.12.4`.

Now let's get this project template ready for use! Open the project folder in the command line and run the command:

```
npm install
```

This will install dependencies like Phaser 3 and a couple of Parcel plugins to automate some mundane tasks.

Next, run this command to start the development server:

```
npm run start
```

You'll know that the development server is ready when you see a message like "Built in 3.25s".

There will also be a line above it that says:

> Server running at http://localhost:8000.

Open a new browser window and go to http://localhost:8000.

You should see a black window with the Phaser logo bouncing around!

Code Editing and Testing

This book assumes you are using Visual Studio Code as your code editor and a modern web browser like Google Chrome.

If you don't have an attachment to another editor like Atom, Brackets, or Webstorm then we highly recommend you download and install VS Code here.

Now open the project in your code editor and we'll start writing some code in the next chapter!

Repeating Background

The first thing we need with an infinite runner is background scenery that repeats forever.

We will be using assets from Game Art Guppy.

There's a lot of different art assets that you can download from there for free.

We'll be using the House 1 Background - Repeatable asset that comes with a laser obstacle, coins to collect, and various decor.

There's also a sleeping dog and cat but we won't be using them in this book. You are welcome to find creative uses for them!

Where to put static assets?

Our project template is set up to serve static files like images in a folder named **`public`**.

Create one at your project root on the same level as the `src` folder.

It should look like this:

```
rocket-mouse-game
    o-- node_modules
    o-- public // <-- here
    o-- src
        o-- scenes
            -- HelloWorldScene.ts
        -- index.html
        -- main.ts
    -- .eslintignore
    -- .eslintrc.js
    -- .gitignore
    -- package.json
    -- tsconfig.json
```

Then create a folder named `house` inside the **`public`** folder. This is where we will put all the background and environment images.

The Game Art Guppy assets have a standard resolution and `@2x` high-resolution versions. We will be using the standard resolution versions in this book.

Copy those from the asset's "Object" folder into `house` as well as the "bg_repeat_340x640.png" file from the "Background" folder.

Your **`public`** folder should look like this:

```
rocket-mouse-game
    o-- node_modules
    o-- public
        o-- house
            -- bg_repeat_340x640.png
            -- object_bookcase1.png
            -- object_bookcase2.png
            -- object_coin.png
            -- object_laser_end.png
            -- object_laser.png
            -- object_mousehole.png
            -- object_window1.png
            -- object_window2.png
    // other files...
```

Files and folders in **`public`** will be served from the root of our web server.

For example, you can load the bookcase at http://localhost:8000/house/object_bookcase1.png.

Try restarting the web server by terminating the process in the command line with `ctrl + c` and running `npm run start` again if the image does not load.

If it still doesn't work then double-check the directory structure to make sure it matches.

Adjusting Game size

You'll notice that our background asset is 640 pixels tall but our game is currently 600 pixels tall. Let's adjust the height of our game to match.

Open `main.ts` and change the `height` property of the `config` variable from `600` to `640`.

```ts
import Phaser from 'phaser'

import HelloWorldScene from './scenes/HelloWorldScene'

const config: Phaser.Types.Core.GameConfig = {
    type: Phaser.AUTO,
    width: 800,
    height: 640,    // <-- here
    // other code...
}

export default new Phaser.Game(config)
```

Save `main.js` and your browser window should automatically reload with the updates. It might be hard to tell that 40 extra pixels were added but you can confirm by right-clicking on the game and selecting "Inspect".

The DevTools window will show up and highlight the `canvas` element being used to render the game.

It should look like this:

```
<canvas width="800" height="640">
```

Create a new Game Scene

Let's create a new Scene called `Game` in the `scenes` folder where `HelloWorldScene.ts` is.

Make a new file named `Game.ts` with the following code:

```
import Phaser from 'phaser'

export default class Game extends Phaser.Scene
{
    constructor()
    {
        super('game')
    }

    preload()
    {

    }

    create()
    {

    }
}
```

This creates a new `Game` Scene by subclassing `Phaser.Scene`. It is given the key `'game'` to uniquely identify it from other Scenes.

Now, let's load our repeating background image and display it in the Scene.

```
preload()
{
    this.load.image('background', 'house/bg_repeat_340x640.png')
}

create()
{
    this.add.image(0, 0, 'background')
        .setOrigin(0, 0)
}
```

The `preload()` method now loads the `bg_repeat_340x640.png` file and assigns it to the key `'background'`.

Then we create an image in `create()` using that key. We place it at `(0, 0)` and then use `setOrigin(0)` to change the origin–it is also known as a pivot or anchor point–to the top left of the image.

By default, most `GameObjects` like images and sprites have an origin of `(0.5, 0.5)` which is the middle of the object.

Next, we'll need to add this `Game` Scene to the `scene` property of our `GameConfig` in `main.ts` to see it in the browser.

Go to `main.ts` and makes these updates:

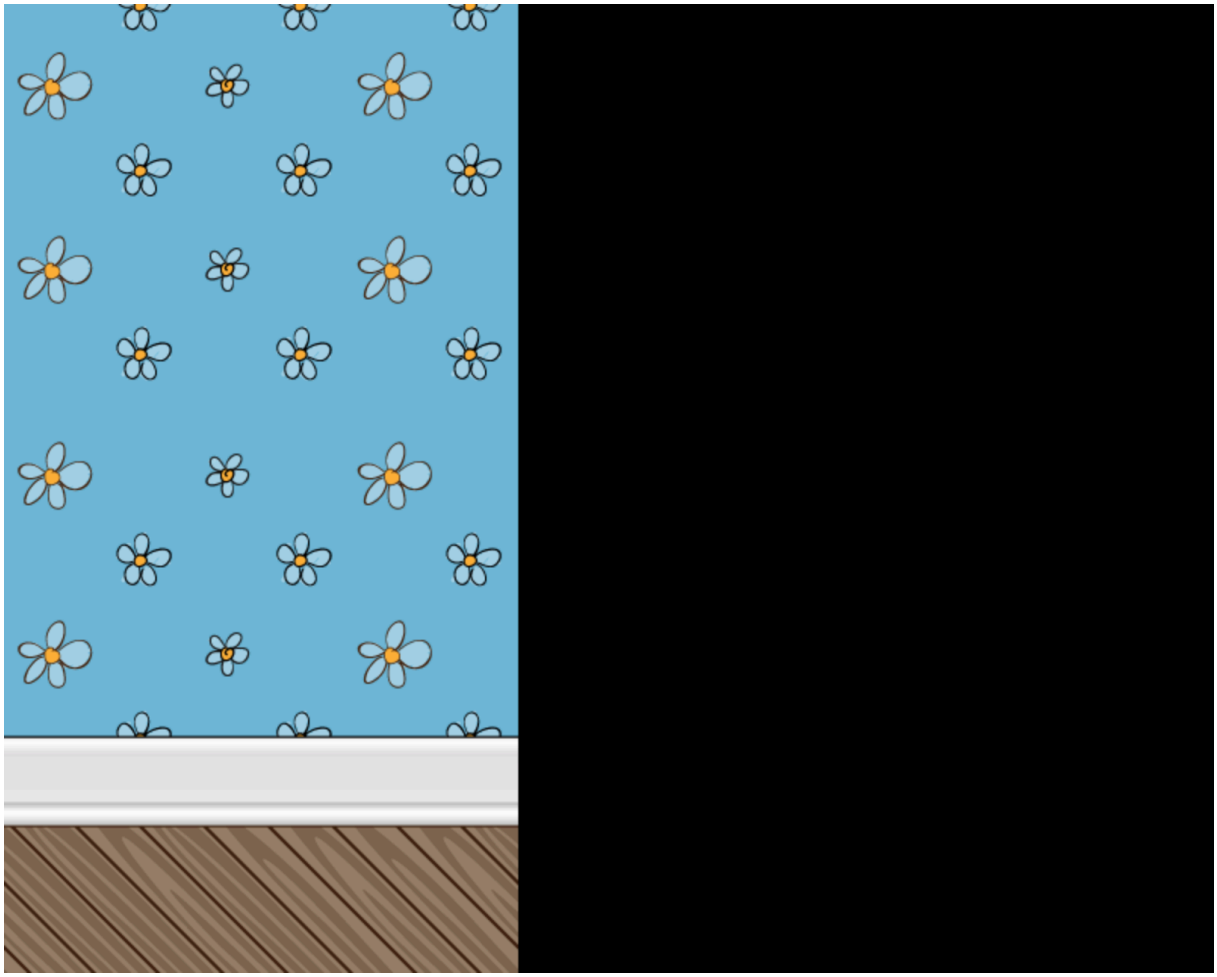
```
import Phaser from 'phaser'

// we can delete this line
// import HelloWorldScene from './scenes/HelloWorldScene'

// add this line to import Game
import Game from './scenes/Game'

const config: Phaser.Types.Core.GameConfig = {
    type: Phaser.AUTO,
    width: 800,
    height: 640,
    physics: {
        default: 'arcade',
        arcade: {
            gravity: { y: 200 }
        }
    },
    // replace HelloWorldScene with Game in the 'scene' property
    scene: [Game]
}

export default new Phaser.Game(config)
```

Notice that we import `Game` at the top and then add it to the `scene` property as an `Array`.

The `scene` property does not have to be an `Array` but it often is because your game will likely have more than one Scene.

Feel free to delete `HelloWorldScene.ts` as we will not be using it anymore.

After these changes, your game should look something like this:



Understanding Origin

Play around with `setOrigin(x, y)` to get a feel for how it works.

The origin designates where `(0, 0)` should be local to the image or sprite.

The values are from `0 - 1` where `0` is the left or top and `1` is the right or bottom for `x` and `y` respectively.

This is different than the Scene coordinates of `(0, 0)` for placing the background image in the game.

Adjusting the origin of an image or sprite is useful to help us more easily reason about transforming them.

For example, rotating an image is usually done with the origin set to `(0.5, 0.5)` because most things spin from the center like a wheel.

It would take a lot more mental work to emulate a rotation from the center with an origin set to `(0, 0)`. You'd have to set the rotation and then figure out what the `x` and `y` offset should be for it to look accurate.

Repeating with a TileSprite

Phaser has a built-in `TileSprite` that will conveniently repeat the same image over a given width.

We can achieve the same effect manually by creating 3 images and setting the `x` value to be next to the previous image.

But why do that when Phaser can handle it for us?

Here's how to make a `TileSprite`:

```
create()
{
    // store the width and height of the game screen
    const width = this.scale.width
    const height = this.scale.height

    // change this.add.image to this.add.tileSprite
    // notice the changed parameters
    this.add.tileSprite(0, 0, width, height, 'background')
        .setOrigin(0)
}
```

This is fairly similar to adding an image except we use `this.add.tileSprite` and pass in `width` and `height` to let the `TileSprite` know how much to repeat the background image.

The `width` and `height` values are retrieved from Phaser's `ScaleManager`.

Now, your browser should reload and show you the background repeated across the game screen. There should be no more black space.

Next, we will add our jetpack-wearing hero: Rocket Mouse!

Adding Rocket Mouse

First, download the Rocket Mouse assets from Game Art Guppy here.

Our intrepid hero will have these animations:

- running
- flying
- falling
- dead

He will be running whenever he is on the ground. Flying when the jetpack is engaged. Falling when he's in the air but no longer ascending. And *dead* when he runs into the deadly home security laser.

This must be a house owned by spies… or a supervillain.

Creating a Sprite Sheet

Animations usually use sprite sheets or atlases because it is more efficient to pick a different frame from a single texture than to switch textures constantly.

The art did not come with a sprite sheet or atlas so we will have to make one.

We recommend using TexturePacker by CodeAndWeb. There are other options as well but TexturePacker's free version will do everything we need.

Download and install TexturePacker and then open it or your texture packing tool of choice.

Drag the standard resolution images of Rocket Mouse (the files without `@2x`) from the "rocketmouse_animations" folder into TexturePacker's Sprite panel on the left.

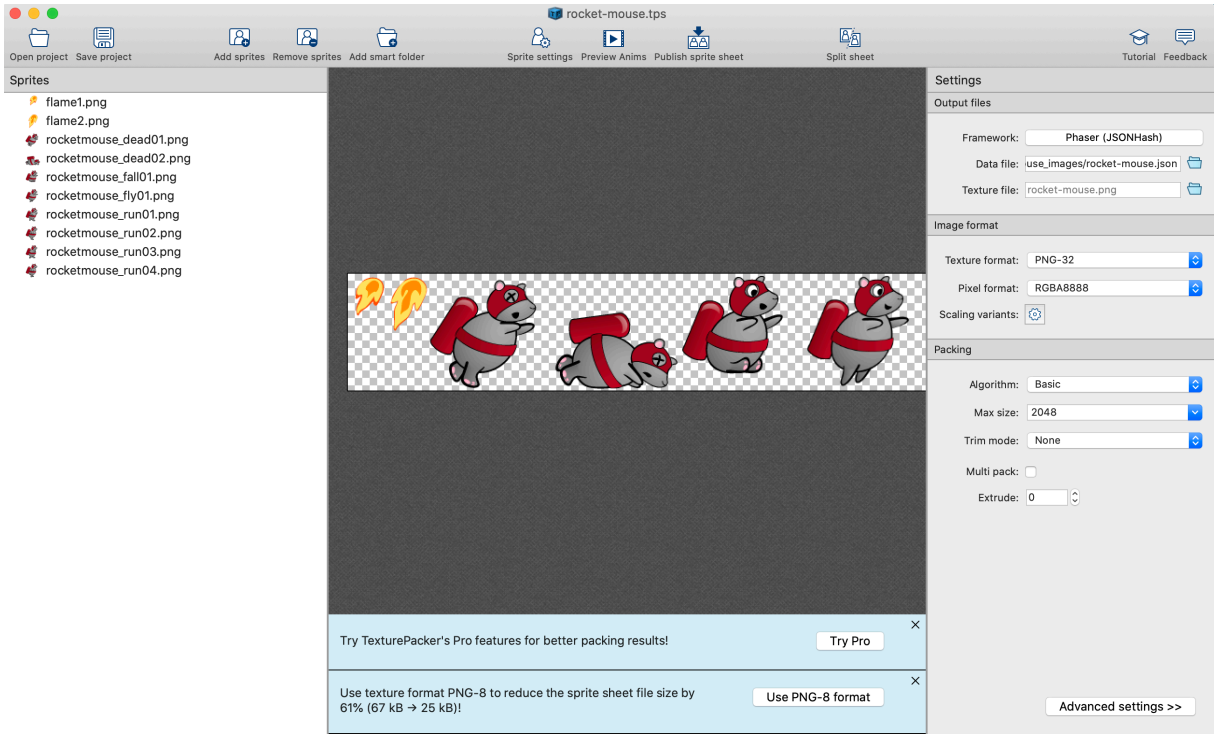
Note: the flying animation image at standard resolution is missing from the download. Go to this issue from our Github repository for this project and grab `rocketmouse_fly01.png`.

In the Settings panel on the right side of TexturePacker select "Phaser (JSONHash)" for `Framework`. Then set the `Data file` and `Texture file` paths to a `characters` folder inside this project's **`public`** folder.

We are using the file name `rocket-mouse`.

You will have to create this folder before being able to select it in TexturePacker. Make it in the same way we did for the house assets.

The rest of the settings you can leave alone. This is what TexturePacker should look like:



You can also choose “Phaser 3” as the `Framework` if you have TexturePacker Pro. Both options will work for this project.

Click the “Publish sprite sheet” button near the middle top of TexturePacker to generate the PNG and JSON files required for Phaser.

Your project should look like this:

```
rocket-mouse-game
    o-- node_modules
    o-- public
        o-- house
        o-- characters  // <-- the new folder
            -- rocket-mouse.json
            -- rocket-mouse.png
    o-- src
    // other files...
```

Loading a Sprite Sheet

We can load the sprite sheet created with TexturePacker like this in the `Game` Scene:

```
1  preload()
2  {
3      // previous code to load background
4      this.load.image('background', 'house/bg_repeat_340x640.png')
5
6      // load as an atlas
7      this.load.atlas(
8          'rocket-mouse',
9          'characters/rocket-mouse.png',
10         'characters/rocket-mouse.json'
11     )
12 }
```

Lines 7 - 11 is how we load the sprite sheet. We are using multiple lines for better book formatting. You can have it as a single line like `this.load.image` above it.

Note on Sprite Sheet vs Sprite Atlas

The terms sprite sheet and sprite atlas are often used interchangeably.

But you'll notice that we are using `this.load.atlas` instead of `this.load.spritesheet` in the example above.

A sprite sheet or sprite atlas is one image that contains many smaller images.

Phaser considers sprite sheets to be images where each frame is a fixed size and can be referenced using a numerical index.

On the other hand, sprite atlases have frames of different sizes and are referenced by alphanumeric names like file names.

The sprite sheet generated by TexturePacker fits the second type and that's why we are using `this.load.atlas`.

Creating a Sprite from an Atlas

We can create either an `Image` or a `Sprite` from an atlas. We'll be using `Sprite` because Rocket Mouse has animations.

Let's make sure the atlas loaded properly by creating a `Sprite` of a flying Rocket Mouse.

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

    // previous code...

    this.add.sprite(
        width * 0.5,    // middle of screen
        height * 0.5,
        'rocket-mouse', // atlas key given in preload()
        'rocketmouse_fly01.png'
    )
}
```

Creating a `Sprite` is very similar to creating an `Image` or `TileSprite`.

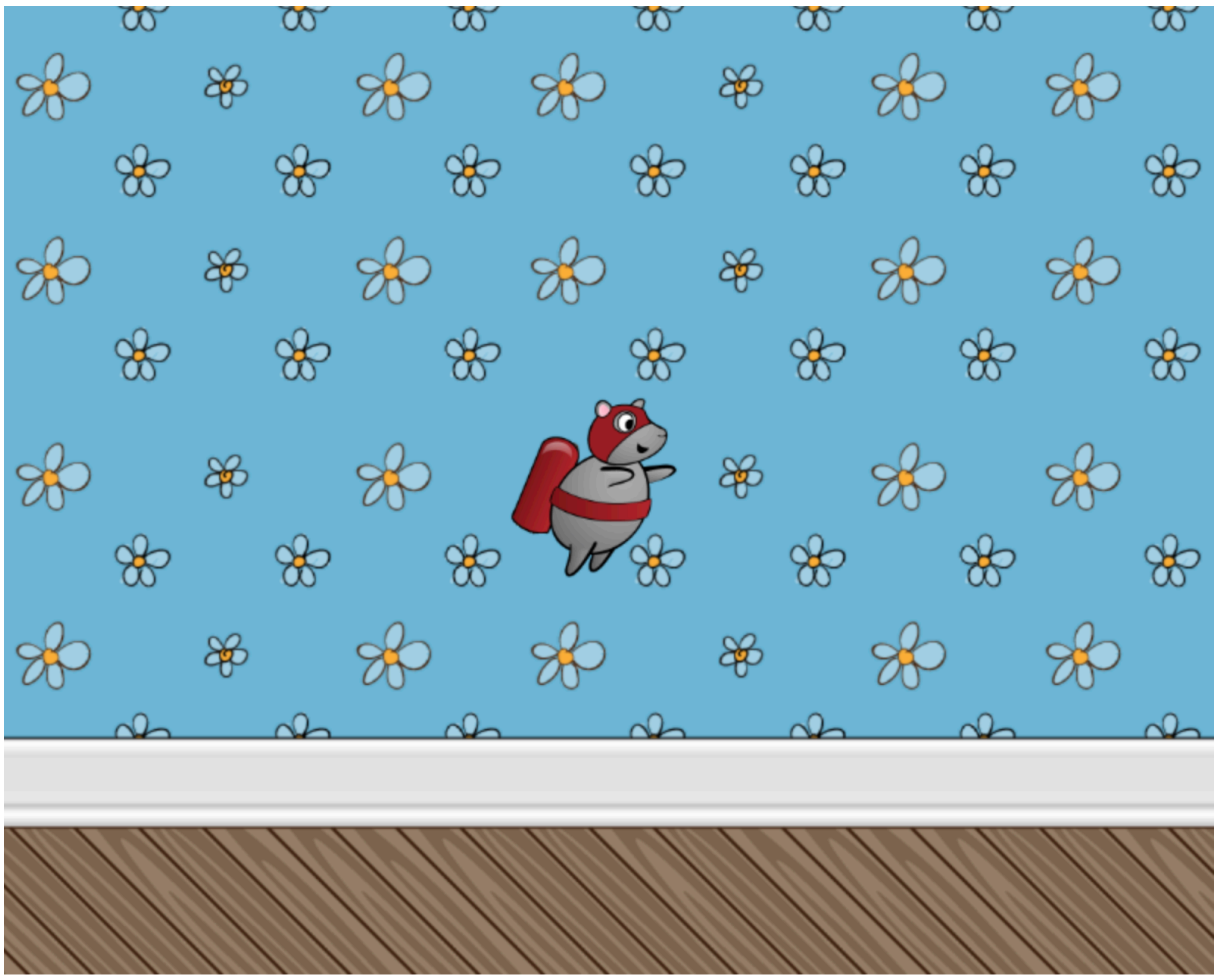
We use `this.add.sprite` and then pass in an `x` and `y` value along with the key of the atlas and the frame to use.

The key `'rocket-mouse'` is the same value used to load the atlas in `preload()`.

Then the frame name `rocketmouse_fly01.png` is from the JSON file. Open `rocket-mouse.json` to see that the original file names of the individual images making up the atlas are used as the key for the properties defining where in the atlas a particular image is.

```
{
    // other keys above
    "rocketmouse_fly01.png":
    {
        "frame": {"x":490,"y":0,"w":134,"h":126},
        "rotated": false,
        "trimmed": false,
        "spriteSourceSize": {"x":0,"y":0,"w":134,"h":126},
        "sourceSize": {"w":134,"h":126}
    },
    // other keys below
}
```

Save your changes and the browser should reload showing Rocket Mouse in the middle of the screen.



Creating the Run Animation

The core difference between an `Image` and a `Sprite` in Phaser is that sprites can play animations and images cannot.

So let's make an animation to play!

Animations are global and created by the `AnimationManager`. Once created, it can be used by any `Sprite` in any Scene.

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

```
create()
{
    this.anims.create({
        key: 'rocket-mouse-run',    // name of this animation
        // helper to generate frames
        frames: this.anims.generateFrameNames('rocket-mouse', {
            start: 1,
            end: 4,
            prefix: 'rocketmouse_run',
            zeroPad: 2,
            suffix: '.png'
        }),
        frameRate: 10,
        repeat: -1  // -1 to loop forever
    })

    // previous code...
}
```

The run animation is created by accessing the `AnimationManager` with `this.animas.create` and passing in a configuration.

The configuration needs a `key` which is the name of the animation: `'rocket-mouse-run'`.

Then `frames` needs an `Array` of frame definitions that we generate using `this.anims.generateFrameNames()`.

We can also manually create the frame definitions without the helper method like this:

```
this.anims.create({
    frames: [
        { key: 'rocket-mouse', frame: 'rocketmouse_run01.png' },
        { key: 'rocket-mouse', frame: 'rocketmouse_run02.png' },
        { key: 'rocket-mouse', frame: 'rocketmouse_run03.png' },
        { key: 'rocket-mouse', frame: 'rocketmouse_run04.png' }
    ]
    // other properties...
})
```

This is the same as what `this.anims.generateFrameNames()` will generate using the data it is given.

The `zeroPad` property is not strictly necessary in this case because we only have 4 frames. Setting `prefix` to `rocketmouse_run0` would have also worked.

But `zeroPad` is necessary if the animation has more than 9 frames.

You can adjust the `frameRate` property to speed up or slow down the animation.

Lastly, `repeat` is set to `-1` so that the animation will loop for as long as it is active.

To play this animation, call `play('rocket-mouse-run')` on the created `Sprite`:

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

    this.add.sprite(
        width * 0.5,
        height * 0.5,
        'rocket-mouse',
        'rocketmouse_fly01.png'
    )
    .play('rocket-mouse-run')
}
```

Now, you should see a floating Rocket Mouse running in mid-air.

We haven't written too much code yet but there are already a couple of things we see that will make the code harder to read and more error-prone.

There will be at least 4 animations and creating them in the `Game` Scene's `create()` method will create clutter. Animations are global so we don't need to create these animations in `Game`.

String literals for key names like `'rocket0mouse'` will be used many times. Having to manually type it out each time will lead to a greater chance of making a typo and introducing a bug.

In the next chapter, we will add a `Preloader` Scene and `enums` to avoid these problems!

Preloader and Enums

We mentioned in the last chapter that animations are global so we can create them outside of the `Game` Scene.

This is also true of loaded assets like images and atlases. Once they are loaded, they can be accessed by any Scene.

Those two things allow for a `Preloader` Scene to handle loading all assets and creating animations when the game starts.

Creating a Preloader

Let's start by creating a `Preloader.ts` file in the `scenes` folder with the following set up:

```
import Phaser from 'phaser'

export default class Preloader extends Phaser.Scene
{
    constructor()
    {
        super('preloader')
    }

    preload()
    {

    }

    create()
    {

    }
}
```

Very simple and looks just like how we started the `Game` Scene a couple of chapters ago.

Next, let's copy the code for `preload()` in the `Game` Scene over to the `Preloader` Scene.

```
preload()
{
    this.load.image('background', 'house/bg_repeat_340x640.png')

    this.load.atlas(
        'rocket-mouse',
        'characters/rocket-mouse.png',
        'characters/rocket-mouse.json'
    )
}
```

Phaser will only call `create()` after all the assets specified in `preload()` have been loaded.

This means we can use `create()` to know when we can go to the `Game` Scene.

We'll use the `SceneManager` to change Scenes once the assets are loaded like this:

```
create()
{
    this.scene.start('game')
}
```

You can delete the `preload()` method from the `Game` Scene since we won't be needing it anymore.

Next, we need to add the `Preloader` Scene to the GameConfig in `main.ts` as we did for the `Game` Scene earlier.

```
import Phaser from 'phaser'

// import Preloader scene
import Preloader from './scenes/Preloader'
import Game from './scenes/Game'

const config: Phaser.Types.Core.GameConfig = {
    // other config...

    // add Preloader as the first element in the Array
    scene: [Preloader, Game]
}

export default new Phaser.Game(config)
```

The first Scene in the `Array` given to the `scene` property will be automatically started by Phaser.

We want the `Preloader` Scene to start first so we add it to the beginning of the list.

Now, your browser should reload and show the same thing we had at the beginning of the chapter.

No visual changes mean we did this code refactor properly!

Creating Animations in the Preloader

The last thing we'll bring over to the `Preloader` Scene is the animations.

Copy the code that creates the `rocket-mouse-run` animation from the `Game` Scene and add it to the `create()` method of `Preloader`.

```
create()
{
    // copied from Game Scene
    this.anims.create({
        key: 'rocket-mouse-run',
        frames: this.anims.generateFrameNames('rocket-mouse', {
            start: 1,
            end: 4,
            prefix: 'rocketmouse_run',
            zeroPad: 2,
            suffix: '.png'
        }),
        frameRate: 10,
        repeat: -1
    })

    // now start the Game Secne
    this.scene.start('game')
}
```

Everything should still look the same once your browser reloads. Rocket Mouse is floating mid-air as he runs in place.

What are Enums?

Enums are a TypeScript language feature that can be replicated in JavaScript using constants. It can also be found in other languages like C#.

For example, an `enum` is an enumeration of values like this:

```
enum Days
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}
```

This is an example of an `enum` called `Days` with values that are each day of the week. Enum values are numbers by default so `Days.Sunday` is equal to `0` and `Days.Saturday` is equal to `6`.

In the modern world, `enums` can help us avoid typos as VS Code will autocomplete it for us, and TypeScript will give us an error if we type something like `Days.Frday` since it was not defined.

This alone makes `enums` a useful feature but it also makes code more readable in situations where you need maximum space savings like coding in the 1980s, for an embedded device, or sending network data in a real-time multiplayer game.

It is easier for humans to read `Days.Sunday` than it is to parse out that `0` means Sunday within the specific context.

We don't have any limited space requirements but we can still benefit from type checking and autocompletion by using string values with `enums`.

Using Enums for Texture Keys

Let's start by creating a new folder named `consts` in the `src` folder on the same level as `scenes`.

```
rocket-mouse-game
    o-- node_modules
    o-- public
    o-- src
        o-- consts // <-- create here
        o-- scenes
    // other files...
```

Then, create a file in `consts` named `TextureKeys.ts` with the following code:

```
enum TextureKeys
{
    Background = 'background',
    RocketMouse = 'rocket-mouse'
}

export default TextureKeys
```

This creates an `enum` called `TextureKeys` with 2 string values named `Background` and `RocketMouse`.

We then export it as the default export on the last line. This is just like how we've been creating Scenes when we use `export default class Preloader` in the declaration.

In this example, we specify the default export after declaring the `enum`.

Exporting is what allows us to use the **`import`** syntax later.

Now, let's go back to the `Preloader` Scene and replace the string literals when we load assets.

```
// import at top of Preloader.ts
import TextureKeys from '../consts/TextureKeys'

export default class Preloader extends Phaser.Scene
{
    // other code...

    preload()
    {
        this.load.image(
            TextureKeys.Background, // <-- here
            'house/bg_repeat_340x640.png'
        )

        this.load.atlas(
            TextureKeys.RocketMouse,    // <-- and here
            'characters/rocket-mouse.png',
            'characters/rocket-mouse.json'
        )
    }

    // create()...
}
```

This is pretty straight forward so go do the same for the keys used to create the background and Rocket Mouse in the `Game` Scene and anywhere else these string literals are being used.

Using Enums for Scene and Animation Keys

Now let's apply what we did with `TextureKeys` to Scene and Animation keys.

For Scenes, we are using the strings `'game'` and `'preloader'`. An `enum` would look like this:

```
enum SceneKeys
{
    Preloader = 'preloader',
    Game = 'game'
}

export default SceneKeys
```

Put the above code in a file named `SceneKeys.ts` in the `consts` folder with `TextureKeys.ts`.

Then go and replace usages of `'game'` in `this.scene.start('game')` and the `Game` Scene constructor as well as `'preloader'` in the `Preloader` Scene constructor.

Animation keys should have an `AnimationKeys.ts` file in the `consts` folder with the following code:

```
enum AnimationKeys
{
    RocketMouseRun = 'rocket-mouse-run'
}

export default AnimationKeys
```

Then replace usages of `'rocket-mouse-run'` in the `Preloader` and `Game` Scenes.

You couldn't tell that we did any work in this chapter by looking at the game but our code is now much better than when we first started!

Next, we will start having Rocket Mouse run along the floor!

Running and Scrolling

The key action in any infinite runner is running.

We will add that in this chapter but first, we need to make Rocket Mouse a physics sprite and add a physics world bounds so that he doesn't fall into the abyss.

Adding Physics to Rocket Mouse

Make these updates to the `Game` Scene:

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

    // change this.add.sprite to this.physics.add.sprite
    // and store the sprite in a mouse variable
    const mouse = this.physics.add.sprite(
        width * 0.5,
        height * 0.5,
        TextureKeys.RocketMouse
        'rocketmouse_fly01.png'
    )
    .play(AnimationKeys.RocketMouseRun)

    const body = mouse.body as Phaser.Physics.Arcade.Body
    body.setCollideWorldBounds(true)

    this.physics.world.setBounds(
        0, 0,   // x, y
        Number.MAX_SAFE_INTEGER, height - 30    // width, height
    )
}
```

Adding physics to Rocket Mouse is as easy as replacing **`this`**`.add.sprite` with **`this`**`.physics.add.sprite`.

The created physics sprite is stored in the `mouse` variable that we use to access the physics body with `mouse.body`.

Notice that we create a local `body` variable to store `mouse.body` and then cast it as a `Phaser.Physics.Arcade.Body`.

Type-casting is a concept in typed languages that lets the programmer tell the compiler that a certain object should be considered a different or more specific type.

The `body` property of `GameObjects` can be a `Phaser.Physics.Arcade.Body` or `Phaser.Physics.Arcade.StaticBody`. We use type-casting so that VS Code can continue to give us accurate auto-complete options.

Then we call `body.setColliderWorldBounds(true)` so that Rocket Mouse will collide with the world bounds set on the next line.

Notice that we use `Number.MAX_SAFE_INTEGER` for the width. We are making an *infinite* runner but computer memory is not infinite so numbers have a maximum size.

> We will go over how to create the feeling of infiniteness from a finite world in a later chapter.

Lastly, we set height as the height of the game screen minus 30 pixels. This will make it look like Rocket Mouse is running on the floor of the background.

Once your browser reloads, Rocket Mouse should appear running in mid-air, fall to the floor, and continue running in place.

Run with Velocity

The player doesn't get to the control when Rocket Mouse runs in an infinite runner. He is always running so we can use `setVelocityX()` as soon as we create him.

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

    const body = mouse.body as Phaser.Physics.Arcade.Body
    body.setCollideWorldBounds(true)

    // set the x velocity to 200
    body.setVelocityX(200)

    // set physics world bounds...
}
```

We add `body.setVelocityX(200)` to make Rocket Mouse move right as soon as he is created.

You should see him run off screen after your browser reloads.

So let's follow him by calling `startFollow()` on the main camera.

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

    this.cameras.main.startFollow(mouse)
    this.cameras.main.setBounds(0, 0, Number.MAX_SAFE_INTEGER, height)
}
```

The first line tells the camera to follow Rocket Mouse as he moves around the world.

The second line sets the bounds for the camera so that it doesn't move past the top or bottom of the screen.

This will keep the environment within the screen bounds as Rocket Mouse falls to the floor or flies to the ceiling with his jetpack.

You'll notice that the background scrolls off the screen quickly leaving our mouse to run in the dark!

Scrolling the Background

Phaser's `TileSprite` is built with scrolling backgrounds in mind so we only need to make a few changes.

```
create()
{
    // width and height defined here...

    this.add.tileSprite(0, 0, width, height, TextureKeys.Background)
        .setOrigin(0, 0)
        .setScrollFactor(0, 0) // <-- keep from scrolling

    // other code...
}
```

We added `setScrollFactor(0, 0)` to stop the background from being scrolled by the camera.

But now it just looks like Rocket Mouse is running in place so we need to add code that will update the `tilePosition` property of the background `TileSprite`.

The camera's scroll value is changing every frame so we'll need to make updates to `tilePosition` in the Scene's `update()` method. And to do that we'll need to store the background `TileSprite` as a class property.

```
// imports...

export default class Game extends Phaser.Scene
{
    // create the background class property
    private background!: Phaser.GameObjects.TileSprite

    // constructor...

    create()
    {
        // width and height...

        // store the TileSprite in this.background
        this.background = this.add.tileSprite(
            0, 0,
            width, height,
            TextureKeys.Background
        )
        .setOrigin(0, 0)
        .setScrollFactor(0, 0)

        // other code...
    }

    update(t: number dt: number)
    {
        // scroll the background
        this.background.setTilePosition(this.cameras.main.scrollX)
    }
}
```

We create a **`private`** class property named `background` and set the type to `Phaser.GameObjects.TileSprite`.

The `!` is TypeScript's non-null assertion operator. This lets us tell TypeScript that the `background` property will never be `undefined` or **`null`**.

TypeScript thinks it can be **`null`** because we are not setting the `background` property in the constructor. Phaser 3's design doesn't let us create a `TileSprite` in the constructor so we are using the `!` operator as a workaround.

Other alternatives are to check that **`this`**`.background` exists before using it or using the optional chaining operator (`?`). We feel using the non-null assertion operator when declaring a class property is the cleanest.

Next, we store the created `TileSprite` in **`this`**`.background` and use it in `update()`.

The two paramters in the `update()` method are the total time elapsed since the game started (`t`) and

the time elapsed since the last frame (`dt`).

The background is scrolled based on the main camera's `scrollX` value by passing it to `this.background.setTilePosition()`.

Now the game looks like Rocket Mouse is running in a house!

Start Rocket Mouse on the Floor

The last thing for this chapter is to start Rocket Mouse on the floor instead of in mid-air.

A trick to make it easier to position characters that walk or run is to set their origin point to the bottom where their feet are.

Then we can very simply set their `y` value to wherever the top of the floor is.

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

    const mouse = this.physics.add.sprite(
        width * 0.5,
        height - 30,    // set y to top of floor
        TextureKeys.RocketMouse,
        'rocketmouse_fly01.png'
    )
    .setOrigin(0.5, 1)  // <-- set origin to feet
    .play(AnimationKeys.RocketMouseRun)

    // other code...
}
```

We set the origin to the bottom of Rocket Mouse's feet by using `setOrigin(0.5, 1)`.

Remember that `(0, 0)` is the top left and `(1, 1)` is the bottom right. Using `(0.5, 1)` sets it to the bottom middle where the feet are.

We also set the `y` position value to be `height - 30`. This was defined as the floor when we set the physics world bounds.

We're making progress on this infinite runner!

In the next chapter, we will add in background decorations to spruce things up.

Decorating the House

Running infinitely in an empty house is pretty boring. The same flower patterned background for as long as the eye can see.

Let's make things more visually interesting by adding decorations.

The House 1 Background - Repeatable asset from Game Art Guppy includes a good variety that should already be in your project's `public/house` folder.

We'll start by adding the mouse hole.

Adding a Mouse Hole Decorations

First, we need to load the `object_mousehole.png` image in the `Preloader` Scene as we did for the background.

```
import Phaser from 'phaser'

import TextureKeys from '../consts/TextureKeys'
// other imports...

export default class Preloader extends Phaser.Scene
{
    // constructor

    preload()
    {
        // the background
        this.load.image(
            TextureKeys.Background,
            'house/bg_repeat_340x640.png'
        )

        // load the mouse hole image
        // notice we are using TextureKeys.MouseHole
        this.load.image(
            TextureKeys.MouseHole,
            'house/object_mousehole.png'
        )
```

```
24
25          // other code
26      }
27
28      // other code...
29  }
```

We add a new line to load the `object_mousehole.png` file right below loading the background from Chapter 2.

Notice that we are using `TextureKeys.MouseHole`. Make sure you add this new value to the `TextureKeys` `enum`.

```
1  enum TextureKeys
2  {
3      Background = 'background',
4      MouseHole = 'mouse-hole',   // <-- like this
5      RocketMouse = 'rocket-mouse'
6  }
7
8  export default TextureKeys
```

Then in the `Game` Scene we can create an image using the mouse hole texture.

Add it after the background and before Rocket Mouse so that it will layer properly.

```
1  create()
2  {
3      // create background TileSprite...
4
5      this.add.image(
6          Phaser.Math.Between(900, 1500), // x value
7          501,                            // y value
8          TextureKeys.MouseHole
9      )
10
11     // other code...
12 }
```

The use of `this.add.image` should be familiar by now but we added something new.

The value passed in as the `x` position uses `Phaser.Math.Between` to generate a random number between `900` and `1500`.

This will add some variety to where the mouse hole is placed each time the game starts.

Give it a test and you'll see that it looks good but notice that it only appears once.

We can solve this by picking a new random position ahead of Rocket Mouse each time the mouse hole passes the left side of the screen.

Looping the Mouse Hole

First, we need to store the mouse hole `Image` in a class property so that we can check when it has scrolled off the left side of the screen.

Create a **`private`** class property named `mouseHole` and set it to the mouse hole image in `create()`.

```
// imports...

export default class Game extends Phaser.Scene
{
    // other properties
    private mouseHole!: Phaser.GameObjects.Image

    create()
    {
        // create background TileSprite...

        this.mouseHole = this.add.image(
            Phaser.Math.Between(900, 1500),
            501,
            TextureKeys.MouseHole
        )

        // other code...
    }

    // other code
}
```

Next, create a **`private`** method called `wrapMouseHole()`. It will determine when the `mouseHole` scrolls off the left side of the screen and give it a new position ahead of Rocket Mouse.

```
private wrapMouseHole()
{
    const scrollX = this.cameras.main.scrollX
    const rightEdge = scrollX + this.scale.width

    if (this.mouseHole.x + this.mouseHole.width < scrollX)
    {
        this.mouseHole.x = Phaser.Math.Between(
            rightEdge + 100,
            rightEdge + 1000
        )
    }
}
```

First, we create a variable `scrollX` that simply stores the main camera's `scrollX` value. Then we

create a `rightEdge` variable that represents the right side of the screen.

It is determined by adding the width of the game screen to `scrollX`.

Then to check when the mouse hole has scrolled off the left side of the screen, we check that `this.mouseHole.x` plus its width is less than `scrollX`.

When that is true, we give `this.mouseHole.x` a new random value that is at least 100 pixels past the right side of the screen and no further than 1,000 pixels.

Last step is to call `this.wrapMouseHole()` from the `update()` method like this:

```
update(t: number, dt: number)
{
    this.wrapMouseHole()

    // other code...
}
```

Watch Rocket Mouse run after your browser reloads and you'll see the mouse hole show up repeatedly at different intervals.

The environment is now much less dull putting aside the many mouse holes in this house–maybe call pest control?

Let There be Windows!

This house is more like an abandoned warehouse with no windows and a rodent problem than a home.

We'll just have to live with the latter but we can add windows in a similar fashion to the mouse hole.

First, load the `object_window1.png` and `object_window2.png` files in the `Preloader` Scene like we did before.

```
preload()
{
    // other images
    // load mouse holes
    this.load.image(
        TextureKeys.MouseHole,
        'house/object_mousehole.png'
    )

    // load windows
    this.load.image(TextureKeys.Window1, 'house/object_window1.png')
    this.load.image(TextureKeys.Window2, 'house/object_window2.png')
```

```
13
14      // other code
15  }
```

Be sure to add the new `enum` values to `TextureKeys` as well.

```
1  enum TextureKeys
2  {
3      // other values...
4
5      Window1 = 'window-1',
6      Window2 = 'window-2'
7  }
```

Next, we can create the images in the `Game` Scene.

We know that we'll need to store them in class properties for the wrapping logic as we did for the mouse hole so add those as well.

```
1  // imports...
2
3  export default class Game extends Phaser.Scene
4  {
5      // other properties...
6
7      private window1!: Phaser.GameObjects.Image
8      private window2!: Phaser.GameObjects.Image
9
10     // constructor...
11
12     create()
13     {
14         // create background and mouse hole...
15
16         this.window1 = this.add.image(
17             Phaser.Math.Between(900, 1300),
18             200,
19             TextureKeys.Window1
20         )
21
22         this.window2 = this.add.image(
23             Phaser.Math.Between(1600, 2000),
24             200,
25             TextureKeys.Window2
26         )
27
28         // other code...
29     }
30
31     // other code...
```

This code should look familiar! It is basically the same as the code for the mouse hole.

You'll see that it also has the same problem where the windows will only show up once.

Let's apply the same technique we used to wrap and repeat the mouse hole for these windows by creating a `wrapWindows()` method:

```
private wrapWindows()
{
    const scrollX = this.cameras.main.scrollX
    const rightEdge = scrollX + this.scale.width

    // multiply by 2 to add some more padding
    let width = this.window1.width * 2
    if (this.window1.x + width < scrollX)
    {
        this.window1.x = Phaser.Math.Between(
            rightEdge + width,
            rightEdge + width + 800
        )
    }

    width = this.window2.width
    if (this.window2.x + width < scrollX)
    {
        this.window2.x = Phaser.Math.Between(
            this.window1.x + width,
            this.window1.x + width + 800
        )
    }
}
```

The concepts here are the same as for the mouse hole except we have 2 windows and they should not overlap each other. We ensure that by using `this.window1.x` as the starting value to generate the random `x` position for `this.window2`.

Notice that `rightEdge` is used for `this.window1` like the mouse hole but then `this.window1.x` is used in place of `rightEdge` for `this.window2.x`.

Remember to call `this.wrapWindows()` from the `update()` method like we did for `this.wrapMouseHole()`.

```
update(t: number, dt: number)
{
    // for mouse holes
    this.wrapMouseHole()

    // add for windows
    this.wrapWindows()

```

```
    // other code...
}
```

Now our jetpack-wearing rodent can see the sights on his infinite run!

Class it up with Bookcases

We've got 1 more set of decorations and they are 2 bookcases. One tall and one short.

Let's start by adding them in the same way we added the windows. There were 2 windows and here we have 2 bookcases. They need to wrap around in the same way and avoid overlapping each other.

We know how to handle that!

First, load the two bookcase PNG files in the `Preloader` Scene.

```
preload()
{
    // other images

    // the previous windows
    this.load.image(TextureKeys.Window1, 'house/object_window1.png')
    this.load.image(TextureKeys.Window2, 'house/object_window2.png')

    // load the bookcases
    this.load.image(TextureKeys.Bookcase1, 'house/object_bookcase1.png')
    this.load.image(TextureKeys.Bookcase2, 'house/object_bookcase2.png')

    // other code...
}
```

Remember to add the new `enum` values to `TextureKeys`!

Next, we can add them to the `Game` Scene with class properties and random positions.

```
// imports...

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

    private bookcase1!: Phaser.GameObjects.Image
    private bookcase2!: Phaser.GameObjects.Image

    // constructor...

    create()
    {
        // window and other decorations...

        this.bookcase1 = this.add.image(
            Phaser.Math.Between(2200, 2700),
            580,
            TextureKeys.Bookcase1
        )
        .setOrigin(0.5, 1)

        this.bookcase2 = this.add.image(
            Phaser.Math.Between(2900, 3400),
            580,
            TextureKeys.Bookcase2
        )
        .setOrigin(0.5, 1)

        // other code...
    }

    // other code...
```

This all looks just like the code we used for windows in the last section.

Wrapping the bookcases back around to the front should be the same too so let's add a `wrapBookcases()` method:

```
private wrapBookcases()
{
    const scrollX = this.cameras.main.scrollX
    const rightEdge = scrollX + this.scale.width

    let width = this.bookcase1.width * 2
    if (this.bookcase1.x + width < scrollX)
    {
        this.bookcase1.x = Phaser.Math.Between(
            rightEdge + width,
            rightEdge + width + 800
        )
    }

    width = this.bookcase2.width
    if (this.bookcase2.x + width < scrollX)
    {
        this.bookcase2.x = Phaser.Math.Between(
            this.bookcase1.x + width,
            this.bookcase1.x + width + 800
        )
    }
}
```

And lastly, add a call to `this.wrapBookcases()` to the `update()` method like we've done before.

```
update(t: number, dt: number)
{
    // wrapping other decorations...
    this.wrapMouseHole()
    this.wrapWindows()

    // wrap bookcases
    this.wrapBookcases()

    // other code...
}
```

You'll notice that the bookcases behave exactly like the windows except… now you get instances when a bookcase overlaps a window!

Maybe that's fine? Some people like books more than sunlight and do block their windows with bookcases…

But if we wanted to solve this how would we do it?

Avoid Overlapping Windows with Bookcases

There are many ways to solve this problem and we are going to use a simple method that has the side-benefit of creating more environmental variety.

What we'll do is check if a window or bookcase's `x` position overlaps with an existing bookcase or window and then hide that window or bookcase if there is an overlap.

First, we'll need an array of bookcases and windows as class properties like this:

```
// imports...

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

    private bookcases: Phaser.GameObjects.Image[] = []
    private windows: Phaser.GameObjects.Image[] = []

    // other code...
}
```

Then we want to add each created window or bookcase to their respective array.

```
create()
{
    // other decorations...

    this.window1 = this.add.image(
        Phaser.Math.Between(900, 1300),
        200,
        TextureKeys.Window1
    )

    this.window2 = this.add.image(
        Phaser.Math.Between(1600, 2000),
        200,
        TextureKeys.Window2
    )

    // create a new array with the 2 windows
    this.windows = [this.window1, this.window2]

    this.bookcase1 = this.add.image(
        Phaser.Math.Between(2200, 2700),
        580,
        TextureKeys.Bookcase1
    )
    .setOrigin(0.5, 1)

```

```
27      this.bookcase2 = this.add.image(
28          Phaser.Math.Between(2900, 3400),
29          580,
30          TextureKeys.Bookcase2
31      )
32      .setOrigin(0.5, 1)
33
34      // create a new array with the 2 bookcases
35      this.bookcases = [this.bookcase1, this.bookcase2]
36
37      // other code...
38  }
```

Now, each time we move a window in `wrapWindows()` we need to check for an overlap like this:

```
1   private wrapWindows()
2   {
3       // other code...
4
5       let width = this.window1.width * 2
6       if (this.window1.x + width < scrollX)
7       {
8           this.window1.x = // pick random position
9
10          // use find() to look for a bookcase that overlaps
11          // with the new window position
12          const overlap = this.bookcases.find(bc => {
13              return Math.abs(this.window1.x - bc.x) <= this.window1.
                    width
14          })
15
16          // then set visible to true if there is no overlap
17          // false if there is an overlap
18          this.window1.visible = !overlap
19      }
20
21      width = this.window2.width
22      if (this.window2.x + width < scrollX)
23      {
24          this.window2.x = // pick random position
25
26          // do the same thing for window2
27          const overlap = this.bookcases.find(bc => {
28              return Math.abs(this.window2.x - bc.x) <= this.window2.
                    width
29          })
30
31          this.window2.visible = !overlap
32      }
33  }
```

The code to pay attention to is the use of `this.bookcases.find()` to find a bookcase that overlaps with a window.

We determine overlap using a simple calculation of distance along the x-axis. Any distance smaller than the width of the window is considered an overlap.

Then we do the same logic for `this.window2`.

Now let's do the same thing for bookcases except we will check for overlaps against windows.

```
private wrapBookcases()
{
    // other code...

    let width = this.bookcase1.width * 2
    if (this.bookcase1.x + width < scrollX)
    {
        this.bookcase1.x = // pick random position

        // use find() to look for a window that overlaps
        // with the new bookcase position
        const overlap = this.windows.find(win => {
            return Math.abs(this.bookcase1.x - win.x) <= win.width
        })

        // then set visible to true if there is no overlap
        // false if there is an overlap
        this.bookcase1.visible = !overlap
    }

    width = this.bookcase2.width
    if (this.bookcase2.x + width < scrollX)
    {
        this.bookcase2.x = // pick random position

        // do the same as we did above for bookcase2
        const overlap = this.windows.find(win => {
            return Math.abs(this.bookcase2.x - win.x) <= win.width
        })

        this.bookcase2.visible = !overlap
    }
}
```

The code is almost the same except we check the list of windows instead of the list of bookcases. We still use the window's width because it is larger than the bookcase's width.

This code can be made more generic to reduce some of the repetition but we will leave it like this for simplicity.

Watch Rocket Mouse run for a while and you should see that the windows and bookcases will never

overlap each other. They simply become hidden when they do!

The mouse hole will get overlapped by the bookcase but we are okay with that.

If you don't want the bookcase to cover the mouse hole then use the techniques we've learned in this chapter to come up with a solution!

Game development is problem-solving.

Our game is looking pretty good with all the new decor!

In the next chapter, we will implement the most exciting part of this game: the jetpack.

Adding a Jetpack

What's more fun than flying around with a jetpack?

The technology isn't quite available to us in real life, yet, but we can do anything in a video game!

Adding jetpack functionality to Rocket Mouse requires turning on a flame animation and applying some upward thrust in the physics engine.

Let's tackle the first part by creating a `RocketMouse` class that we can attach a flame animation to.

Creating a RocketMouse Class

Phaser 3 `GameObjects` do not have children. This means that you cannot attach an `Image` to another `Image` or `Sprite`.

Unless you are using a `Container`.

We will need to attach a flame animation as a child of Rocket Mouse so let's subclass `Phaser.GameObjects.Container` in our `RocketMouse` class.

Create a new folder named `game` in the `src` folder on the same level as `scenes`. Then create a `RocketMouse.ts` file in the new `game` folder.

```
rocket-mouse-game
    o-- node_modules
    o-- public
    o-- src
        o-- consts
        o-- game    // <-- new folder
            -- RocketMouse.ts   // <-- new file
        o-- scenes
    // other files...
```

Then add this code to `RocketMouse.ts`:

```
import Phaser from 'phaser'

export default class RocketMouse extends Phaser.GameObjects.Container
{
    constructor(scene: Phaser.Scene, x: number, y: number)
    {
        super(scene, x, y)
    }
}
```

This should look somewhat familiar as it is similar to creating new Scenes except we subclass `Phaser.GameObjects.Container` instead of `Phaser.Scene`.

A `Container` holds children so let's add a Rocket Mouse sprite as we did in `Game`. The difference is that we don't need to add a physics sprite. We'll add physics to the `Container` instead.

```
import Phaser from 'phaser'

// import these
import TextureKeys from '../consts/TextureKeys'
import AnimationKeys from '../consts/AnimationKeys'

export default class RocketMouse extends Phaser.GameObjects.Container
{
    constructor(scene: Phaser.Scene, x: number, y: number)
    {
        super(scene, x, y)

        // create the Rocket Mouse sprite
        const mouse = scene.add.sprite(0, 0, TextureKeys.RocketMouse)
            .setOrigin(0.5, 1)
            .play(AnimationKeys.RocketMouseRun)

        // add as child of Container
        this.add(mouse)
    }
}
```

We create a `Sprite` in the constructor similar to how we did it in the `Game` Scene. Then we add it to the `Container` with `this.add(mouse)`.

Now, let's try using `RocketMouse` in the `Game` Scene.

Using RocketMouse in the Game Scene

We will replace the call to `this.physics.add.sprite()` in `create()` by creating a new `RocketMouse` instance.

```
// import RocketMouse
import RocketMouse from '../game/RocketMouse'

// then in create()...
create()
{
    // decorations...

    // remove line with const mouse = this.physics.add.sprite(...)

    // add new RocketMouse
    const mouse = new RocketMouse(this, width * 0.5, height - 30)
    this.add.existing(mouse)

    // error happens here
    const body = mouse.body as Phaser.Physics.Arcade.Body
    body.setCollideWorldBounds(true)
    body.setVelocityX(200)

    // other code...
}
```

We removed the previous way of creating a physics sprite that represented Rocket Mouse and replaced it with `new RocketMouse(...)`.

Then we use `this.add.existing(mouse)` to add the `RocketMouse` instance to the Scene.

You will get errors after making this change because the `mouse` variable no longer has a `body` property.

Let's fix that by adding a physics body to `RocketMouse`.

Adding Physics to a Container

Go back to `RocketMouse.ts` and add this line to the constructor:

```
constructor(scene: Phaser.Scene, x: number, y: number)
{
    super(scene, x, y)

    const mouse = scene.add.sprite(0, 0, TextureKeys.RocketMouse)
        .setOrigin(0.5, 1)
        .play(AnimationKeys.RocketMouseRun)

    this.add(mouse)

    // add a physics body
    scene.physics.add.existing(this)
}
```

You'll notice that Rocket Mouse now appears higher in the game. Let's see what's going on by turning on physics debug draw.

Go to `main.ts` and add `debug: true` to the physics config:

```
const config: Phaser.Types.Core.GameConfig = {
    // other config...
    physics: {
        default: 'arcade',
        arcade: {
            gravity: { y: 200 },
            debug: true     // <-- turn on debug draw
        }
    }
}
```

Now you should see that it looks like Rocket Mouse is standing on a box!

We will need to adjust the size and offset of the physics body to match where the Rocket Mouse sprite is.

The top left corner of the box and the point by Rocket Mouse's feet is the origin of the `Container`. Recall that we set the Rocket Mouse sprite origin to `(0.5, 1)` which is the bottom center.

This means we will need to offset the physics body by the negative height and negative half-width of Rocket Mouse.

We can do it like this:

```
constructor(scene: Phaser.Scene, x: number, y: number)
{
    super(scene, x, y)

    const mouse = scene.add.sprite(0, 0, TextureKeys.RocketMouse)
        .setOrigin(0.5, 1)
        .play(AnimationKeys.RocketMouseRun)

    this.add(mouse)

    scene.physics.add.existing(this)

    // adjust physics body size and offset
    const body = this.body as Phaser.Physics.Arcade.Body
    body.setSize(mouse.width, mouse.height)
    body.setOffset(mouse.width * -0.5, -mouse.height)
}
```

Rocket Mouse should be running on the floor like before with a box around him after these changes.

Add Some Flames

Now, we can add some flames for the jetpack. The flames are animated by switching between two different images.

You can see them in the `rocket-mouse.png` file that we created using TexturePacker in Chapter 3.

Let's create a flames animation in `Preloader` like we did for Rocket Mouse's running animation so that we can use it for the jetpack.

```
// Preloader.ts

create()
{
    // previous run animation
    this.anims.create({
        key: AnimationKeys.RocketMouseRun,
        frames: this.anims.generateFrameNames(TextureKeys.RocketMouse,
            { start: 1, end: 4, prefix: 'rocketmouse_run', zeroPad: 2,
            suffix: '.png' }),
        frameRate: 10,
        repeat: -1
    })

    // create the flames animation
    this.anims.create({
        key: AnimationKeys.RocketFlamesOn,
```

```
        frames: this.anims.generateFrameNames(TextureKeys.RocketMouse,
            { start: 1, end: 2, prefix: 'flame', suffix: '.png'}),
        frameRate: 10,
        repeat: -1
    })

    this.scene.start(SceneKeys.Game)
}
```

Notice that we are using `AnimationKeys.RocketFlameOn` so be sure to add that to `AnimationKeys.ts` like this:

```
enum AnimationKeys
{
    RocketMouseRun = 'rocket-mouse-run',
    RocketFlamesOn = 'rocket-flames-on'
}

export default AnimationKeys
```

Now we can go back to `RocketMouse.ts` and create a flame sprite and play the `RocketFlamesOn` animation.

Add this code to the constructor of `RocketMouse`:

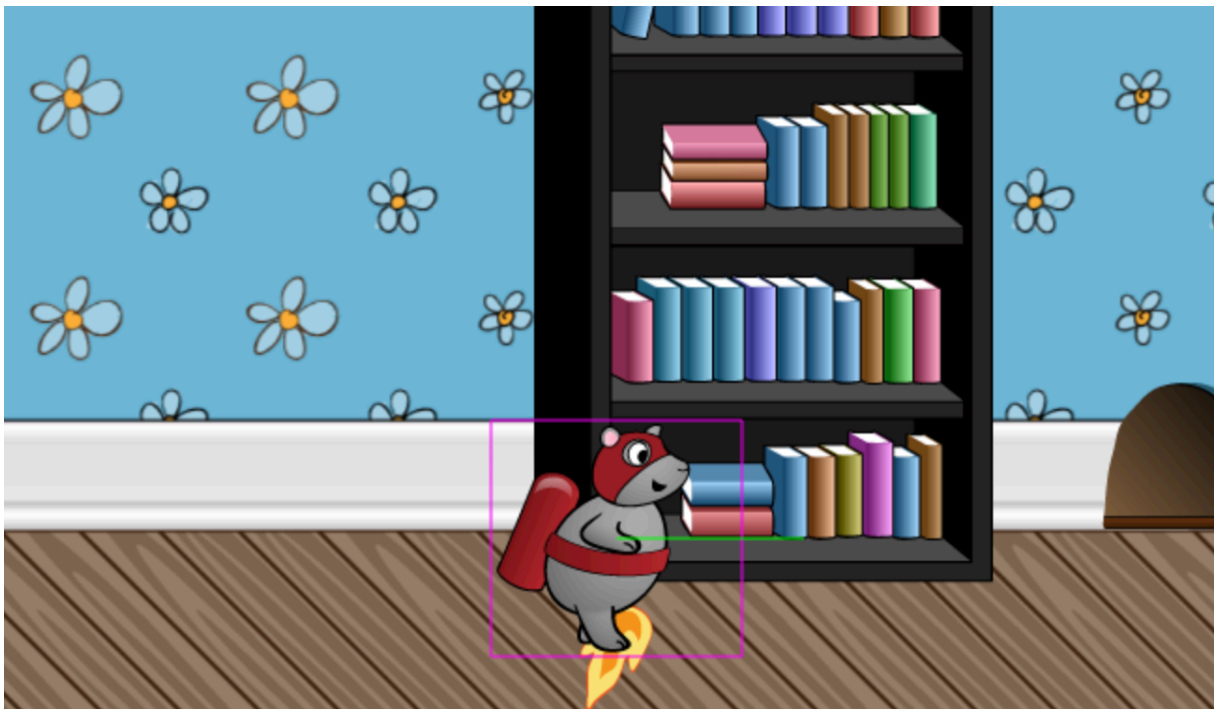
```
constructor(scene: Phaser.Scene, x: number, y: number)
{
    // create mouse sprite...

    // create the flames and play the animation
    const flames = scene.add.sprite(0, 0, TextureKeys.RocketMouse)
        .play(AnimationKeys.RocketFlamesOn)

    // add this first so it is under the mouse sprite
    this.add(flames)
    this.add(mouse)

    // other code...
}
```

Creating the flames sprite and playing an animation should look familiar. It is the same as what we did for the `mouse` sprite.

Pay attention to the order in which we added the `flames` and `mouse` sprites to the `Container`. The `flames` are added first so that it layers behind the `mouse`.

This will give us the intended effect after we position the flames under the jetpack.

After the browser reloads you'll see the flame animation playing by the feet of Rocket Mouse.



We can fix this by changing the position of the `flame` from `(0, 0)` to `(-63, -15)` like this:

```
1  const flames = scene.add.sprite(-63, -15, TextureKeys.RocketMouse)
```

Now it should look better! You are welcome to adjust these values if it doesn't look right to you.

Turn the Jetpack On and Off

We have the flame animation working but that's just the first part of getting the jetpack to work.

The second part is to apply thrust.

But before we do that, let's add a method to turn the jetpack on or off. We'll need it when the player wants to go back to the ground.

First, create a new `flames` class property so that we can access the flames sprite from a method.

```
export default class RocketMouse extends Phaser.GameObjects.Container
{
    private flames: Phaser.GameObjects.Sprite

    constructor(scene: Phaser.Scene, x: number, y: number)
    {
        // create mouse...

        this.flames = scene.add.sprite(-63, -15, TextureKeys.
            RocketMouse)
             .play(AnimationKeys.RocketFlamesOn)

        this.add(this.flames)

        // other code...
    }
```

All we did was replace the local `const flames` with the class property `this.flames`.

Next, create a method called `enableJetpack(enabled: boolean)` with the following implementation:

```
enableJetpack(enabled: boolean)
{
    this.flames.setVisible(enabled)
}
```

This method simply sets the `visible` property of the `Sprite` to `true` or `false` depending on what was passed in for `enabled`.

Now, let's start with the jetpack turned off by calling `this.enableJetpack(false)` in the constructor.

```
constructor(scene: Phaser.Scene, x: number, y: number)
{
    // create mouse...

    this.flames = scene.add.sprite(-63, -15, TextureKeys.RocketMouse)
        .play(AnimationKeys.RocketFlamesOn)

    this.enableJetpack(false)

    // other code...
}
```

Rocket Mouse should now be running on the floor without the jetpack turned on!

Adding Thrust

We will allow the player to turn on the jetpack when they press the Space bar.

That means we'll set the upwards acceleration of Rocket Mouse and enable the jetpack flames animation when the Space bar is down.

For simplicity, we'll access the Space bar using Phaser's `CursorKeys`.

Let's start by creating a `cursors` class property to hold an instance of `CursorKeys`:

```
// imports...

export default class RocketMouse extends Phaser.GameObjects.Container
{
    // other properties...

    // create the cursors property
    private cursors: Phaser.Types.Input.Keyboard.CursorKeys

    constructor(scene: Phaser.Scene, x: number, y: number)
    {
        // previous code...

        // get a CursorKeys instance
        this.cursors = scene.input.keyboard.createCursorKeys()
    }
}
```

Phaser's `CursorKeys` is a convenient way to get access to the 4 arrow keys and the Space bar.

With this we can check if the Space bar is pressed in a `preUpdate()` method.

A `Container` does not normally implement a `preUpdate()` method but it will get called if we create one.

> Note: we can create a more traditional `update()` method and then call it from the `Game` Scene instead. But for simplicity, we will just use `preUpdate()`. You are welcome to use `update()` instead.

```
preUpdate()
{
    const body = this.body as Phaser.Physics.Arcade.Body

    // check is Space bar is down
    if (this.cursors.space?.isDown)
    {
        // set y acceleration to -600 if so
        body.setAccelerationY(-600)
        this.enableJetpack(true)
    }
    else
    {
        // turn off acceleration otherwise
        body.setAccelerationY(0)
        this.enableJetpack(false)
    }
}
```

In `preUpdate()` we set the `y` acceleration of Rocket Mouse to `-600` if the Space bar is down. Then we turn off the acceleration when the Space bar is no longer pressed.

We also enable and disable the jetpack flames animation at the appropriate times.

Try it out and you should see the jetpack turn on when you press and hold the Space bar as Rocket Mouse takes flight.

He will still be running because we have not created his fly or fall animations yet.

Adding Fly and Fall Animations

Let's put some finishing touches on our jetpack functionality by adding the fly and fall animations for Rocket Mouse.

These animations will only be 1 frame long so they are animations in concept only but it sets you up to add more complex animations with flapping ears or wiggling feet.

We can create these two animations in the `Preloader` Scene as we've done before:

```
// Preloader.ts

create()
{
    // previous run animation
    this.anims.create({
        key: AnimationKeys.RocketMouseRun,
        frames: this.anims.generateFrameNames(TextureKeys.RocketMouse,
            { start: 1, end: 4, prefix: 'rocketmouse_run', zeroPad: 2,
            suffix: '.png' }),
        frameRate: 10,
        repeat: -1
    })

    // new fall animation
    this.anims.create({
        key: AnimationKeys.RocketMouseFall,
        frames: [{
            key: TextureKeys.RocketMouse,
            frame: 'rocketmouse_fall01.png'
        }]
    })

    // new fly animation
    this.anims.create({
        key: AnimationKeys.RocketMouseFly,
        frames: [{
            key: TextureKeys.RocketMouse,
            frame: 'rocketmouse_fly01.png'
        }]
    })

    // other animations...
}
```

These animations are created in the same way we explained in Chapter 3. Refer back to that chapter if you are unsure of what we did.

Remember to add the new `RocketMouseFall` and `RocketMouseFly` `enum` values to `AnimationKeys.ts`.

Now, we can play the fly animation when we enable the jetpack and then play the fall animation when Rocket Mouse's velocity is greater than 0.

First, head back to `RocketMouse.ts` and create a new class property called `mouse` to store the reference to the mouse sprite.

```
// imports...

export default class RocketMouse extends Phaser.GameObjects.Container
{
    // other properties...
    private mouse: Phaser.GameObjects.Sprite

    constructor(scene: Phaser.Scene, x: number, y: number)
    {
        super(scene, x, y)

        // replace const mouse with this.mouse
        this.mouse = scene.add.sprite(0, 0, TextureKeys.RocketMouse)
            .setOrigin(0.5, 1)
            .play(AnimationKeys.RocketMouseRun)

        // add flames...

        // add this.mouse instead of mouse
        this.add(this.mouse)

        scene.physics.add.existing(this)

        // change calls to mouse variable to this.mouse
        const body = this.body as Phaser.Physics.Arcade.Body
        body.setSize(this.mouse.width, this.mouse.height)
        body.setOffset(this.mouse.width * -0.5, -this.mouse.height)

        // cursors...
    }

    // other code...
}
```

This change is necessary so that we can access the mouse sprite in `preUpdate()` to play different animations.

Next, we can play the fly and fall animations when appropriate:

```
preUpdate()
{
    const body = this.body as Phaser.Physics.Arcade.Body

    if (this.cursors.space?.isDown)
    {
        body.setAccelerationY(-600)
        this.enableJetpack(true)

        // play the fly animation
        this.mouse.play(AnimationKeys.RocketMouseFly, true)
    }
    else
    {
        body.setAccelerationY(0)
        this.enableJetpack(false)
    }

    // check if touching the ground
    if (body.blocked.down)
    {
        // play run when touching the ground
        this.mouse.play(AnimationKeys.RocketMouseRun, true)
    }
    else if (body.velocity.y > 0)
    {
        // play fall when no longer ascending
        this.mouse.play(AnimationKeys.RocketMouseFall, true)
    }
}
```

We play the fly animation right after we turn on the jetpack.

The newer block of code after that checks if Rocket Mouse is touching the ground using `body.blocked.down`. If he is touching the ground then switch to the run animation.

If he is not touching the ground and has a `y` velocity greater than 0 then play the fall animation.

Give it a try and you should see Rocket Mouse fly, then fall, then land and run!

This was a meaty chapter!

We've got a lot of the game working but it isn't particularly challenging since there's nothing to do…

In the next chapter, we will add some laser obstacles to spice it up!

Creating a Laser Obstacle

You might be wondering what kind of house has lasers for security?

Perhaps, a house with as many mouse holes as this one does? Clearly, there's a pest-control problem.

Lasers might be overkill but… go big or go home, right?

Preload the Laser Images

The first thing we need to do is load the 2 laser images in the `Preloader` Scene.

```
// Preloader.ts

preload()
{
    // other images...

    this.load.image(TextureKeys.LaserEnd, 'house/object_laser_end.png')
    this.load.image(TextureKeys.LaserMiddle, 'house/object_laser.png')
}
```

Remember to add the new `enum` values to `TextureKeys` like this:

```
enum TextureKeys
{
    // previous values...

    LaserEnd = 'laser-end',
    LaserMiddle = 'laser-middle'
}

export default TextureKeys
```

Now, we can compose the laser obstacle using the two images we loaded.

Composing the Laser Obstacle

We will be creating our laser obstacle in a similar fashion to what we did in the last chapter with `RocketMouse.ts`.

There will be 3 pieces to the laser obstacle including a top, middle, and bottom. You may have noticed that we only have 2 images: an end and a middle.

The bottom will be created by flipping the end image vertically.

Let's start by creating a `LaserObstacle.ts` file in the `game` folder on the same level as `RocketMouse.ts`.

```
import Phaser from 'phaser'

export default class LaserObstacle extends Phaser.GameObjects.Container
{
    constructor(scene: Phaser.Scene, x: number, y: number)
    {
        super(scene, x, y)
    }
}
```

This starting point is basically the same as what we did in the last chapter for `RocketMouse`.

Next, we can create each of the 3 sections and place them appropriately to look like a laser beam between two ends.

```
constructor(scene: Phaser.Scene, x: number, y: number)
{
    super(scene, x, y)

    // create a top
    const top = scene.add.image(0, 0, TextureKeys.LaserEnd)
        .setOrigin(0.5, 0)

    // create a middle and set it below the top
    const middle = scene.add.image(
        0,
        top.y + top.displayHeight,
        TextureKeys.LaserMiddle
    )
    .setOrigin(0.5, 0)

    // set height of middle laser to 200px
    middle.setDisplaySize(middle.width, 200)

    // create a bottom that is flipped and below the middle
    const bottom = scene.add.image(0, middle.y + middle.displayHeight,
        TextureKeys.LaserEnd)
```

```
        .setOrigin(0.5, 0)
        .setFlipY(true)

    // add them all to the Container
    this.add(top)
    this.add(middle)
    this.add(bottom)
}
```

There's quite a bit of code here but it is all relatively simple.

First, we create a `top` which is just the `LaserEnd` image with the origin set to the middle top. This will make it easier to calculate where the middle laser part should go.

Next, we create a `middle` which is the `LaserMiddle` image with the origin set to the middle top as well. Notice that the `y` position value is set to `top.y + top.displayHeight`. This will set the `middle` image right below the `top` image.

Then, we set the height of the `middle` to be 200. This is the laser part between the top and bottom ends.

After that, we create a `bottom` which is the same `LaserEnd` image as the `top` but flipped vertically with the same origin as everyone else. Its `y` position is set to `middle.y + middle.displayHeight` just like how the `y` position for `middle` was set to `top.y + top.displayHeight`. This ensures the `bottom` image is right below the `middle` image.

Lastly, we add all 3 parts to the `Container`.

Add LaserObstacle to Game Scene

Let's see our handiwork in action by adding a `LaserObstacle` to the `Game` Scene.

It will be very similar to how we created the `RocketMouse` class instance.

```
// import LaserObstacle
import LaserObstacle from '../game/LaserObstacle'

// then in create()...
create()
{
    // create decorations...

    const laserObstacle = new LaserObstacle(this, 900, 100)
    this.add.existing(laserObstacle)

    // create RocketMouse and other code...
}
```

We've omitted the previous code but you'll want to add the `LaserObstacle` after the last decoration and before `RocketMouse` so that it layers properly.

This is what you should see as Rocket Mouse runs toward 900 pixels on the x-axis:

‘

Looping the LaserObstacle

We'll want this `LaserObstacle` to loop back around once it has scrolled off the left side of the screen just like the decorations in Chapter 6.

That means we'll need to create a class property and create a method called `wrapLaserObstacle()` that gets called in the `update()` method.

We've created many class properties already so you should know what to do!

```
// imports...

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

    private laserObstacle!: LaserObstacle

    create()
    {
        // create decorations...

        this.laserObstacle = new LaserObstacle(this, 900, 100)
        this.add.existing(this.laserObstacle)

        // create RocketMouse and other code...
    }

}
```

We create the class property `laserObstacle` and then replace uses of `const laserObstacle` with `this.laserObstacle`.

Next, let's add the `wrapLaserObstacle()` method:

```
private wrapLaserObtacle()
{
    const scrollX = this.cameras.main.scrollX
    const rightEdge = scrollX + this.scale.width

    const width = this.laserObstacle.width
    if (this.laserObstacle.x + width < scrollX)
    {
        this.laserObstacle.x = Phaser.Math.Between(
            rightEdge + width,
            rightEdge + width + 1000
        )

        this.laserObstacle.y = Phaser.Math.Between(0, 300)
    }
}
```

You'll notice that this is very similar to `wrapMouseHole()` except we also pick a random value for the `y` position of `this.laserObstacle`.

This will put the laser higher or lower on the screen forcing the player to maneuver with the jetpack.

Next, add a call to `this.wrapLaserObstacle()` in the `update()` method:

```
update(t: number, dt: number)
{
    // wrap decorations...

    this.wrapLaserObtacle()

    // scroll background...
}
```

You may notice that the laser will disappear before it has fully scrolled off the screen

This is because the width of the `LaserObstacle` is currently 0. This will be fixed once we set a physics body and define a size as we did with `RocketMouse` in the last chapter.

For now, just check that the laser is looping back to the front and picking different `y` position values.

Making the Laser Dangerous

Let's add a physics body to the `LaserObstacle` so that our jetpack-wearing hero can run into it and get zapped.

Make the following changes to the constructor of `LaserObstacle`:

```
constructor(scene: Phaser.Scene, x: number, y: number)
{
    // previous code...

    scene.physics.add.existing(this, true)

    const body = this.body as Phaser.Physics.Arcade.StaticBody
    const width = top.displayWidth
    const height = top.displayHeight + middle.displayHeight +
        bottom.displayHeight

    body.setSize(width, height)
    body.setOffset(-width * 0.5, 0)

    // reposition body
    body.position.x = this.x + body.offset.x
    body.position.y = this.y
}
```

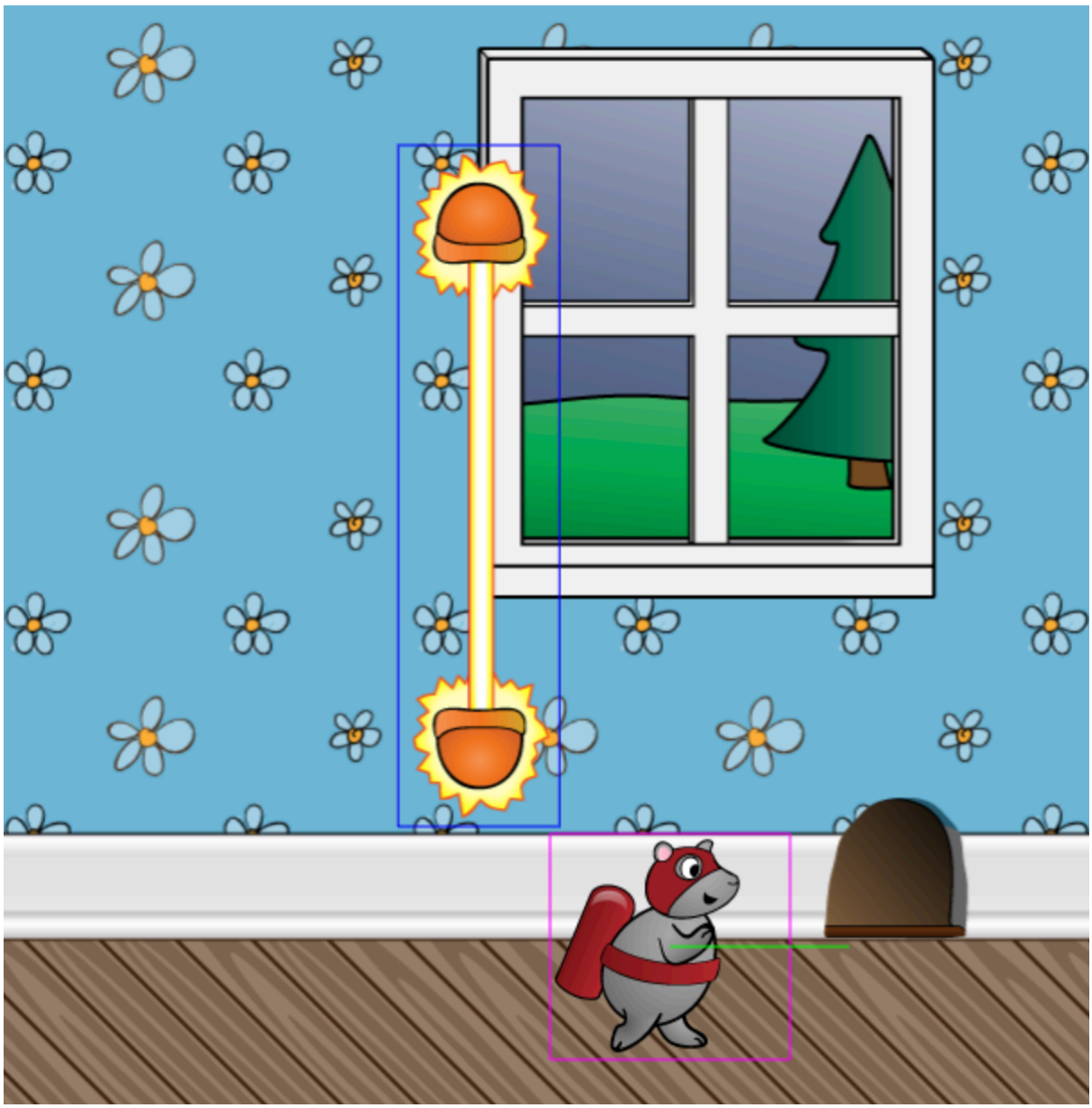
The first thing we do is use `scene.physics.add.existing(this, true)` to add a static physics body to the `Container`.

The second parameter given the **`true`** value designates that the physics body should be static. A static physics body will collide with normal bodies but will not be moved by gravity or pushed by other objects.

This is perfect for the `LaserObstacle` because it needs to collide with Rocket Mouse and stay where we put it.

Next, we set the size by using the width of the `top` and the combined height of all the pieces. Then the offset is set to negative half-width and `0` based on the values we used for `setOrigin()` earlier.

Now, you should see a box around the `LaserObstacle` like this:



Let it scroll off the screen and loop back around. You'll notice that the collision box disappears!

This is because the physics body is static. We'll need to manually set the body's position when we move the `LaserObstacle` in the `Game` Scene.

Update `wrapLaserObstacle()` with the follow code:

```
private wrapLaserObtacle()
{
    const scrollX = this.cameras.main.scrollX
    const rightEdge = scrollX + this.scale.width

    // body variable with specific physics body type
    const body = this.laserObstacle.body as
        Phaser.Physics.Arcade.StaticBody

    // use the body's width
    const width = body.width
    if (this.laserObstacle.x + width < scrollX)
    {
        this.laserObstacle.x = Phaser.Math.Between(
            rightEdge + width,
            rightEdge + width + 1000
        )
        this.laserObstacle.y = Phaser.Math.Between(0, 300)

        // set the physics body's position
        // add body.offset.x to account for x offset
        body.position.x = this.laserObstacle.x + body.offset.x
        body.position.y = this.laserObstacle.y
    }
}
```

Now the physics collision box should move with the `LaserObstacle` when we loop it around to the front of Rocket Mouse.

Collide and Zap

A laser obstacle is only good if it stings!

Let's add a check for overlap between Rocket Mouse and the `LaserObstacle` so that we can make it dangerous.

Update the `create()` method and add `handleOverlapLaser()` in the `Game` Scene:

```
// Game.ts

create()
{
    // previous code...

    this.physics.add.overlap(
        this.laserObstacle,
        mouse,
        this.handleOverlapLaser,
        undefined,
        this
    )
}

private handleOverlapLaser(
    obj1: Phaser.GameObjects.GameObject,
    obj2: Phaser.GameObjects.GameObject
)
{
    console.log('overlap!')
}
```

We use `this.physics.add.overlap()` between the `mouse` and `this.laserObstacle` so that the `handleOverlapLaser` method will be called when the two `GameObjects` overlap.

Give it a try and you should see the message `overlap!` in the Console of the browser's Developer Tools. You can bring up up the DevTools by right-clicking and selecting "Inspect" or going to View > Developer > Developer Tools.

Now that we know when Rocket Mouse collides with the laser, we can play the death animation and launch his body forward.

First, for the death animation we need to create it in the `Preloader` class like this:

```
// Preloader.ts

create()
{
    // other animations...

    this.anims.create({
        key: AnimationKeys.RocketMouseDead,
        frames: this.anims.generateFrameNames(TextureKeys.RocketMouse,
            {
            start: 1,
            end: 2,
            prefix: 'rocketmouse_dead',
            zeroPad: 2,
            suffix: '.png'
        }),
        frameRate: 10
    })

    this.scene.start(SceneKeys.Game)
}
```

Remember to add the `enum` value `RocketMouseDead` to `AnimationKeys.ts`.

Next, let's create a `kill()` method in the `RocketMouse` class that will play this death animation and launch our furry friend.

```
kill()
{
    this.mouse.play(AnimationKeys.RocketMouseDead)

    const body = this.body as Phaser.Physics.Arcade.Body
    body.setAccelerationY(0)
    body.setVelocity(1000, 0)
    this.enableJetpack(false)
}
```

Give it a try and you'll see Rocket Mouse play the death animation, get launched, and then run faster than he's ever run before!

This looks more like a power-up than death…

We can fix this by applying specific logic depending on what state Rocket Mouse is in. For example, he can be dead or alive and the logic to handle what happens in those two states would be different.

This is where a State Machine comes in handy.

Poor Man's State Machine

For simplicity's sake we will use a very basic state machine: the `swtich` statement.

First, create an `enum` at the top of the `RocketMouse.ts` file. This `enum` will define the different states Rocket Mouse can be in.

```
enum MouseState
{
    Running,
    Killed,
    Dead
}
```

Next, add a class property called `mouseState`:

```
export default class RocketMouse extends Phaser.GameObjects.Container
{
    private mouseState = MouseState.Running

    // other code...
}
```

Our first use of this `mouseState` property is to only perform the kill logic if Rocket Mouse is in the `RUNNING` state.

Update `kill()` with this change:

```
kill()
{
    // don't do anything if not in RUNNING state
    if (this.mouseState !== MouseState.Running)
    {
        return
    }

    // set state to KILLED
    this.mouseState = MouseState.Killed

    this.mouse.play(AnimationKeys.RocketMouseDead)

    const body = this.body as Phaser.Physics.Arcade.Body
    body.setAccelerationY(0)
    body.setVelocity(1000, 0)
    this.enableJetpack(false)
}
```

We add a check at the top to see if `mouseState` is not equal to `MouseState.RUNNING` and early exit if that is true. That means all the code after the `if` statement is not run if `mouseState` is not `RUNNING`.

This will ensure that Rocket Mouse doesn't get killed multiple times or when he is already dead.

It just isn't right to kick a mouse when he's down. Right?

Next, let's add a **switch** statement to `preUpdate()` and move the logic we already have for the `MouseState.RUNNING` state.

```
preUpdate()
{
    // switch on this.mouseState
    switch (this.mouseState)
    {
        // move all previous code into this case
        case MouseState.Running:
        {
            const body = this.body as Phaser.Physics.Arcade.Body

            if (this.cursors.space?.isDown)
            {
                body.setAccelerationY(-600)
                this.enableJetpack(true)

                this.mouse.play(AnimationKeys.RocketMouseFly, true)
            }
            else
            {
                body.setAccelerationY(0)
                this.enableJetpack(false)
            }

            if (body.blocked.down)
            {
                this.mouse.play(AnimationKeys.RocketMouseRun, true)
            }
            else if (body.velocity.y > 0)
            {
                this.mouse.play(AnimationKeys.RocketMouseFall, true)
            }

            // don't forget the break statement
            break
        }
    }
}
```

There's a lot of code here but we only added about 6 lines and 4 of them are braces.

We add a `swtich` state for **`this`**`.mouseState` and then only execute the code we had previously for the `RUNNING` state. Don't forget the **`break`** at the end of the **`case`** block.

Now give it a try and you should see a dead Rocket Mouse sliding along the floor… forever.

We are making progress!

Next, we will add logic for the `MouseState.KILLED` and `MouseState.DEAD` states to reduce Rocket Mouse's acceleration over time until his limp body comes to a stop.

Add these two **`case`** blocks to the **`switch`** statement:

```
preUpdate()
{
    // move this out of the run logic to the top of the method
    const body = this.body as Phaser.Physics.Arcade.Body

    switch (this.mouseState)
    {
        case MouseState.Running:
        {
            // run logic
            break
        }

        case MouseState.Killed:
        {
            // reduce velocity to 99% of current value
            body.velocity.x *= 0.99

            // once less than 5 we can say stop
            if (body.velocity.x <= 5)
            {
                this.mouseState = MouseState.Dead
            }
            break
        }

        case MouseState.Dead:
        {
            // make a complete stop
            body.setVelocity(0, 0)
            break
        }
    }
}
```

We set **`this`**`.mouseState` to `MouseState.KILLED` in the `kill()` method and then launch Rocket Mouse forward at high speed.

Then in `preUpdate()` we perform logic for the `KILLED` state by reducing the velocity until it is less than or equal to `5`. Once that is true, we set the `mouseState` to `MouseState.DEAD`.

Finally, in `MouseState.DEAD` we bring Rocket Mouse to a complete stop by setting the velocity to `0`.

Give it a try and the entire death animation will look a lot more believable now!

We went over a lot in this chapter and now the game actually has a challenge. It is, perhaps, even fun?

In the next chapter, we will add a Game Over Scene that will let us play again so we don't have to just sit and stare at a dead Rocket Mouse.

Game Over and Play Again

After 3 beefy chapters, we are going to relax just a little and work on adding a Game Over screen with the ability to play again.

We ended the last chapter by implementing a poor man's State Machine and we will use that to determine when we show the Game Over screen.

Create a GameOver Scene

Our Game Over screen is going to be a Scene that runs in parallel with the `Game` Scene.

This is a common way to organize code for persistent UI like showing a score or a pause button as well as to show modal dialogs.

Let's start by creating a `GameOver.ts` file in the `scenes` folder on the same level as `Game.ts`.

```
1  import Phaser from 'phaser'
2
3  import SceneKeys from '../consts/SceneKeys'
4
5  export default class GameOver extends Phaser.Scene
6  {
7      constructor()
8      {
9          super(SceneKeys.GameOver)
10     }
11
12     create()
13     {
14     }
15 }
```

Creating new Scenes should be fairly familiar by now. This is the third Scene we've made in this book.

Don't forget to add a new `enum` member to `SceneKeys.ts`.

```
enum SceneKeys
{
    Preloader = 'preloader',
    Game = 'game',
    GameOver = 'game-over'  // <-- new member
}

export default SceneKeys
```

Next, we can create some text to tell the player that they can press the Space bar to play again.

Add this code to the `create()` method:

```
create()
{
    // object destructuring
    const { width, height } = this.scale

    // x, y will be middle of screen
    const x = width * 0.5
    const y = height * 0.5

    // add the text with some styling
    this.add.text(x, y,  'Press SPACE to Play Again', {
        fontSize: '32px',
        color: '#FFFFFF',
        backgroundColor: '#000000',
        shadow: { fill: true, blur: 0, offsetY: 0 },
        padding: { left: 15, right: 15, top: 10, bottom: 10 }
    })
    .setOrigin(0.5)
}
```

The first thing we do is create a `width` and `height` variable using a language feature called Object Destructuring or a Destructuring Assignment. This is a TypeScript and modern JavaScript feature that creates variables by taking the values of the properties in an object that have the same names.

In this example, the `ScaleManager` has the properties `width` and `height`. Object Destructuring lets us take those properties and create 2 constant local variables from them.

It is the same as writing this:

```
const width = this.scale.width
const height = this.scale.height
```

After that we choose an `x` and `y` position and create a `Text` object with some styling to make it more pleasing to look at.

We used the Text Styler for Phaser 3 tool that you can find here. It is a free web-based tool from Our-

cade that lets you quickly and visually design text styles.

It will generate the appropriate code as you make changes. Just copy the config code when you are happy with the design.

Having to tweak designs in code and reload the browser each time can be frustratingly slow so we recommend you try out the tool!

Show the Game Over Scene

Before we can use the `GameOver` Scene, we need to add it to the list of scenes in Phaser's `GameConfig` in `main.ts`.

```
// import the scene
import GameOver from './scenes/GameOver'

// add it to the config
const config: Phaser.Types.Core.GameConfig = {
    // other code...
    scene: [Preloader, Game, GameOver]  // <-- add it at the end
}
```

Now, we can use this Scene by running it after Rocket Mouse hits a laser obstacle and comes to a stop.

There are a few ways to implement this and we have chosen the simplest one for this book. We'll give you a more advanced option later that you can try implementing yourself!

Let's go to the poor man's State Machine that we have in the `RocketMouse` class and make this change to the `DEAD` state:

```
// import SceneKeys at the top of RocketMouse.ts
import SceneKeys from '../consts/SceneKeys'

// then in the preUpdate() method
switch (this.mouseState)
{
    // other cases...

    case MouseState.Dead:
    {
        body.setVelocity(0, 0)

        // add this line
        this.scene.scene.run(SceneKeys.GameOver)
        break
    }
```

```
17  }
```

This is a very simple one-liner that tells the `SceneManager` to run the `GameOver` Scene. Notice that we are using `run()` and not `start()`.

> We will be calling `run()` every frame that we are in the `DEAD` state. Phaser appears to be okay with this but you can also use `this.scene.scene.isActive(SceneKeys.GameOver)` to check that the `GameOver` Scene is not currently running before trying to run it.

We are using the odd-looking `this.scene.scene` because the `Container` has a reference to the `Game` Scene that is stored as a property named `scene` and then the `Game` Scene has a reference to the `SceneManager` that is also named `scene`.

It may look confusing but it is just how Phaser 3 happens to be structured. Just know that using `this.scene.scene` is intended and not a typo.

Give this a try and you'll see a banner with the message `Press SPACE to Play Again`.

Pressing the Space bar won't do anything because we haven't implemented that yet!

Pressing Space to Play Again

We will listen for the Space bar being pressed in the `GameOver` Scene and then restart the game when it does.

Add this to the bottom of `create()` in `GameOver.ts`:

```
1  create()
2  {
3      // previous code to create text
4
5      // listen for the Space bar getting pressed once
6      this.input.keyboard.once('keydown-SPACE', () => {
7          // stop the GameOver scene
8          this.scene.stop(SceneKeys.GameOver)
9
10         // stop and restart the Game scene
11         this.scene.stop(SceneKeys.Game)
12         this.scene.start(SceneKeys.Game)
13     })
14 }
```

We listen for the `keydown-SPACE` event emitted by the `KeyboardManger` one time by using `.once()` instead of `.on()`.

When this happens, we stop the `GameOver` and `Game` Scenes and then we start the `Game` Scene again.

Give it a try and you should now be able to play the game again and again!

Another Way to Show GameOver Scene

The main issue with how we are showing the `GameOver` Scene is that the `RocketMouse` class is making the call.

It is perfectly correct code given the way Phaser 3 is structured but a new developer looking at this project–or you in a couple of months–might not expect `RocketMouse` to make the Scene change.

It is more fitting for a Scene to handle Scene changes like we did with pressing Space to play again.

One way to have the `Game` Scene control showing the `GameOver` Scene is to have `RocketMouse` emit an event when it enters the `DEAD` state. A `Phaser.Events.EventEmitter` can be created as a class property of `RocketMouse` and then wrapper methods can be created to allow the `Game` Scene to listen for an event like this:

```
mouse.once('dead', () => {
    // run GameOver scene
})
```

We will leave implementing something like this as an exercise for you!

Should the Game Over Scene Restart the Game Scene?

If we are saying that it is odd for `RocketMouse` to perform a Scene change then wouldn't it also stand to reason that it is odd for the `GameOver` Scene to control restarting the `Game` Scene?

It was the `Game` Scene that started the `GameOver` Scene so the responsibility to stop the `GameOver` Scene should belong to the `Game` Scene. It would also make more sense if the `Game` Scene restarted itself.

We opted for the simplest implementation given the simplicity of this game but it is good to be thinking about these things!

If you were to give the responsibility of stopping the `GameOver` Scene and restarting itself to the `Game` Scene, how would you do it?

We will leave that implementation as an exercise for you!

One tip is that you can use `this.scene.restart()` from the `Game` Scene instead of calling `stop()` and `start()` like we are doing now.

This Rocket Mouse game is almost completely playable!

In the next chapter, we will look at adding coins to collect and display a score based on the number of collected coins.

Add Coins to Collect

Every game needs something to count for scoring purposes. We can use the total distance ran as a score but collecting coins is more fun.

Calculating a score from coins collected and distance ran is another option that we'll leave as an exercise for you!

Load the Coin Image

Let's start by loading the `object_coin.png` file in the `Preloader` Scene.

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

    this.load.image(TextureKeys.Coin, 'house/object_coin.png')
}
```

Remember to add the new `enum` value to `TextureKeys` like this:

```
enum TextureKeys
{
    // other members...
    Coin = 'coin'
}

export default TextureKeys
```

Creating Coins with a Group

We will be spawning a random number of coins periodically and the best way to handle that is to use a Phaser `Group`.

A `Group` has built-in support for recycling `GameObjects` and we'll want to do that instead of creating new coins as we need them and destroying coins as they scroll off the screen.

Frequent or large amount of object creation and destruction can result in performance problems on less capable devices.

Let's create a class property to start so that we can put the coin spawning logic in a `Game` Scene method.

```
// imports...

export default class Game extends Phaser.Scene
{
    // other properties...
    private coins!: Phaser.Physics.Arcade.StaticGroup

    create()
    {
        // decorations and laser obstacle...

        this.coins = this.physics.add.staticGroup()
        this.spawnCoins()

        // create Rocket Mouse and other code...
    }

    private spawnCoins()
    {
        // code to spawn coins
    }

    // other code...
}
```

Most of this should look familiar! We create a `coins` property using the `!` which is TypeScript's non-null assertion operator. This lets us tell TypeScript that `this.coins` will be set even though it doesn't happen in the constructor.

Then we set `this.coins` to a `Phaser.Physics.Arcade.StaticGroup` and call an empty method named `spawnCoins()`.

We are using a `StaticGroup` so that the physics objects created by the group will be static. This way the coins will float in the air and not be affected by gravity.

The `spawnCoins()` method is where we will add code to create coins randomly.

```
private spawnCoins()
{
    // make sure all coins are inactive and hidden
    this.coins.children.each(child => {
        const coin = child as Phaser.Physics.Arcade.Sprite
        this.coins.killAndHide(coin)
        coin.body.enable = false
    })

    const scrollX = this.cameras.main.scrollX
    const rightEdge = scrollX + this.scale.width

    // start at 100 pixels past the right side of the screen
    let x = rightEdge + 100

    // random number from 1 - 20
    const numCoins = Phaser.Math.Between(1, 20)

    // the coins based on random number
    for (let i = 0; i < numCoins; ++i)
    {
        const coin = this.coins.get(
            x,
            Phaser.Math.Between(100, this.scale.height - 100),
            TextureKeys.Coin
        ) as Phaser.Physics.Arcade.Sprite

        // make sure coin is active and visible
        coin.setVisible(true)
        coin.setActive(true)

        // enable and adjust physics body to be a circle
        const body = coin.body as Phaser.Physics.Arcade.StaticBody
        body.setCircle(body.width * 0.5)
        body.enable = true

        // move x a random amount
        x += coin.width * 1.5
    }
}
```

There's a lot of code to unpack here but most of it is fairly straight forward.

First, we go through all the children of the `coins` group and make sure they are all not visible, not active, and have a disabled physics body.

Then we pick a starting `x` position that is 100 pixels past the right edge of the screen.

A random number of coins is generated between `1` and `20` that we then use to create that number of coins.

Each coin is given a random `y` value that is within 100 pixels of the top and bottom of the screen. Then the coin is set to be visible and active and given a circle physics body. We also make sure the body is enabled.

Lastly, we add 1.5x the coin's width to the `x` position so that the next coin will be spaced away from the previous one.

Give this a try and you should see a bunch of randomly placed coins as Rocket Mouse runs forward.

Other Coin Formations

We won't cover spawning pre-made coin formations because it is outside the scope of this book.

However, you can try implementing that yourself by creating different functions or methods that place coins in specific positions to display a message like "Hello!" or some other pattern.

Then instead of picking random positions, you can pick a random function or method to use.

Collecting Coins

Coin collection is implemented in a similar fashion to colliding with the `LaserObstacle` except nothing bad happens when you do!

First, we need to detect an overlap between Rocket Mouse and any coin like this:

```
// Game.ts

create()
{
    // previous code...

    // create overlap detection
    this.physics.add.overlap(
        this.coins,
        mouse,
        this.handleCollectCoin,
        undefined,
        this
    )
}

// this method will be called when an overlap happens
private handleCollectCoin(
    obj1: Phaser.GameObjects.GameObject,
    obj2: Phaser.GameObjects.GameObject
)
```

```
22  {
23      // obj2 will be the coin
24      const coin = obj2 as Phaser.Physics.Arcade.Sprite
25
26      // use the group to hide it
27      this.coins.killAndHide(coin)
28
29      // and turn off the physics body
30      coin.body.enable = false
31  }
```

Adding code to detect an overlap should be familiar.

The `handleCollectCoin()` callback method will be called when an overlap occurs. At that time, we set the coin to invisible and inactive by calling `killAndHide()` on the group and passing in the `coin` that overlapped with Rocket Mouse.

The coin's physics body is also disabled so that it doesn't register more overlaps.

Give it a try and you'll see that the coins disappear as Rocket Mouse's collision box overlaps with the coin.

Displaying a Score

Now, let's keep track of how many coins we've collected and display it as a UI element at the top left of the screen.

Let's start by creating 2 class properties to hold the score count and a reference to the `Text` object that will display the score.

```
1  export default class Game extends Phaser.Scene
2  {
3      // class proeprties...
4
5      private scoreLabel!: Phaser.GameObjects.Text
6      private score = 0
7
8      init()
9      {
10         this.score = 0
11     }
12
13     // other code...
14 }
```

Notice that we added an `init()` method which is called by Phaser when the Scene is started. It is called before `preload()` and `create()`.

We use it here to reset `this.score` to `0` so that we start fresh each time we play the game again.

Next, create the `Text` instance in the `create()` method:

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

    this.scoreLabel = this.add.text(10, 10, `Score: ${this.score}`, {
        fontSize: '24px',
        color: '#080808',
        backgroundColor: '#F8E71C',
        shadow: { fill: true, blur: 0, offsetY: 0 },
        padding: { left: 15, right: 15, top: 10, bottom: 10 }
    })
    .setScrollFactor(0)
}
```

The `scoreLabel` is created in a similar way to what we did in the `GameOver` Scene. Recall that we used the Text Styler for Phaser 3 web-based tool to generate the `TextStyle` config options.

Then we use `setScrollFactor(0)` to keep it in place even as the camera scrolls to follow Rocket Mouse.

You'll see a yellow rectangle with `Score: 0` at the top left corner of the screen once the browser reloads.

To finish up, we just need to add `1` to `this.score` each time a coin is collected and update the `scoreLabel` with new text.

```
private handleCollectCoin(
    obj1: Phaser.GameObjects.GameObject,
    obj2: Phaser.GameObjects.GameObject
)
{
    // previous code

    // increment by 1
    this.score += 1

    // change the text with new score
    this.scoreLabel.text = `Score: ${this.score}`
}
```

The code is pretty straight forward. We add `1` to `this.score` and then update `this.scoreLabel.text` with the new score value.

Give it a try and watch the score count change as you collect coins!

Note on Collision Box Sizes

You'll notice that Rocket Mouse has a pretty big physics box. Bigger than his actual size and way bigger than a size that is actually fun.

We can adjust the size of Rocket Mouse's physics box using `setSize` on the physics body.

Let's do this for Rocket Mouse by updating the `setSize` and `setOffset` code in the constructor of `RocketMouse` like this:

```
export default class RocketMouse extends Phaser.GameObjects.Container
{
    constructor(scene: Phaser.Scene, x: number, y: number)
    {
        // other code...

        const body = this.body as Phaser.Physics.Arcade.Body

        // use half width and 70% of height
        body.setSize(this.mouse.width * 0.5, this.mouse.height * 0.7)

        // adjust offset to match
        body.setOffset(this.mouse.width * -0.3, -this.mouse.height +
            15)
    }
}
```

Make these changes and you'll notice that Rocket Mouse starts by falling as his collision box is now smaller.

We can fix this by adjusting the physic's world bounds in the `Game` Scene like this:

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

    // change "- 30" to "- 55"
    this.physics.world.setBounds(
        0, 0,
        Number.MAX_SAFE_INTEGER, height - 55
    )
}
```

Now give it a try and fly around collecting coins.

The way we sized the collision box accounts for the feet tucking in during the fall animation which will make it more accurate for barely avoiding lasers during a fall.

You should also notice that the `LaserObstacle` has a larger collision box than it needs as well. Use

what we did here as a guide for shrinking that collision box.

Creating Coins Periodically

The last thing we need to do is to create coins periodically.

There are a few ways we can do this including using Phaser's `Clock` with `this.time.addEvent` to spawn new coins every X seconds.

We are going to use a much simpler approach and simply call `this.spawnCoins()` when we wrap the second bookcase.

Make this change to the `wrapBookcases()` method:

```
private wrapBookcases()
{
    // other code...

    if (this.bookcase2.x + width < scrollX)
    {
        // move this.bookcase2...

        // call spawnCoins()
        this.spawnCoins()
    }
}
```

Give it a try and you'll see the coins appear once and then appear again but the second wave of coins is not collectible!

Let's fix this by adjusting the `spawnCoins()` method when we set the coin's physics body to be a circle.

```
private spawnCoins()
{
    // other code...

    for (let i = 0; i < numCoins; ++i)
    {
        // create a coin from the group

        const body = coin.body as Phaser.Physics.Arcade.StaticBody
        body.setCircle(body.width * 0.5)
        body.enable = true

        // update the body x, y position from the GameObject
        body.updateFromGameObject()

```

```
            x += coin.width * 1.5
        }
    }
```

Notice the call to `body.updateFromGameObject()` after the physics body is enabled.

This will move the physics body to the same position as to where the `GameObject` is visually displayed.

Try it out and you'll see that the coins are collectible after they are respawned!

The last thing that we'll cover in this book is to make the game truly infinite and avoid the problem of a player being so good that they scroll reach the maximum size of a `Number` as we mentioned in Chapter 5.

We will tackle this in the next chapter!

To Infinity and Beyond

We talked about how our scrolling system is not truly *infinite* in Chapter 5 but let's go a little deeper on why that is.

The game certainly looks infinite right now and you can leave it running for hours and hours and have no problem.

It is so close to infinite that, for all intents and purposes, you can call it infinite and no one will know the difference. Except for you, anyone who reads the code, or someone *really* good at dodging laser beams with a jetpack.

Where Does our World End?

In Chapter 5, we made the width of our physics world bounds `Number.MAX_SAFE_INTEGER` which means Rocket Mouse will run into a wall there and continue running in place.

But the camera will be stationary and no more scrolling will occur.

The game would appear *frozen* in the eyes of the player as they are no longer moving forward.

One way to fix this is to teleport everything back to the beginning. This will create the illusion of an infinite world within finite world bounds.

However, we don't have to wait until we get to `MAX_SAFE_INTEGER` before teleporting Rocket Mouse, the decorations, coins, and laser obstacle back to the beginning.

We can do it at a number like `2500` pixels and keep the player in that range the entire time they are playing the game. If we can make it seamless then they'll never know that they are on a treadmill!

Creating a Seamless Teleport

A teleport consists of moving all the `GameObjects` we've created backward by the same amount once we've scrolled past a specific position.

This will create the illusion that everything is still moving *forward* together even though they all moved backward together.

First, create a new class property in the `Game` Scene to hold a reference Rocket Mouse. We'll need to adjust his `x` position when a teleport happens.

```
// imports...

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

    private mouse!: RocketMouse

    // other code...
}
```

Next, we should create a `teleportBackwards()` method in the `Game` Scene that handles when and how a teleport should happen:

```
private teleportBackwards()
{
    const scrollX = this.cameras.main.scrollX
    const maxX = 2500

    // perform a teleport once scrolled beyond 2500
    if (scrollX > maxX)
    {
        // teleport the mouse and mousehole
        this.mouse.x -= maxX
        this.mouseHole.x -= maxX

        // teleport each window
        this.windows.forEach(win => {
            win.x -= maxX
        })

        // teleport each bookcase
        this.bookcases.forEach(bc => {
            bc.x -= maxX
        })

        // teleport the laser
        this.laserObstacle.x -= maxX
        const laserBody = this.laserObstacle.body
            as Phaser.Physics.Arcade.StaticBody

        // as well as the laser physics body
        laserBody.x -= maxX

```

```
31          // teleport any spawned coins
32          this.coins.children.each(child => {
33              const coin = child as Phaser.Physics.Arcade.Sprite
34              if (!coin.active)
35              {
36                  return
37              }
38
39              coin.x -= maxX
40              const body = coin.body as Phaser.Physics.Arcade.StaticBody
41              body.updateFromGameObject()
42          })
43      }
44  }
```

There is a bit of code here but it is all pretty straight forward.

We decide on `2500` pixels as the `maxX` value for when we perform the teleport. Then we move all the `GameObjects` we've created backward by subtracting `maxX` from their `x` position.

Lastly, call `teleportBackwards()` at the end of the `update()` method like this:

```
1  update(t: number, dt: number)
2  {
3      // previous code...
4
5      this.teleportBackwards()
6  }
```

Give it a try and you'll see that everything will work as expected except for the background and some coins depending on how many were spawned.

Rocket Mouse, the decorations, and the laser obstacle should teleport backward seamlessly. You'll know when a teleport happens when the background changes abruptly.

Teleporting the Background Seamlessly

We are not fully in control of how the background scrolls because we are using the `TileSprite`.

The background is scrolled using the camera's `scrollX` value and that is automatically adjusted after we teleport Rocket Mouse. All we control is the position of Rocket Mouse.

Subtracting `maxX` from `this.background.tilePosition.x` doesn't appear to work even though it seems to make sense!

One way to resolve this is to handle the background scrolling manually by creating 4 background images that we loop to the front as soon as one scrolls off the screen.

But that feels like a lot of work to replicate functionality that the `TileSprite` gives us for free.

There's another approach that we discovered through trial and error: teleport backward when scrolled beyond a *multiple* of the background texture width.

The background image has a width of `340` pixels so can we multiply that by 7 and use `2380` instead of `2500`.

Try it out by changing the `maxX` value in `teleportBackwards()`:

```
1  private teleportBackwards()
2  {
3      const scrollX = this.cameras.main.scrollX
4
5      // change to 2380 from 2500
6      const maxX = 2380
7
8      // previous code...
9  }
```

Play the game for a bit and the background should teleport seamlessly along with the other `GameObjects`.

Fixing Coin Teleportation

If you play the game long enough you'll sometimes notice that uncollected coins will disappear suddenly before they've scrolled off the screen.

Recall that we are spawning coins when we wrap the second bookcase to a new position but this can happen while the previous set of spawned coins is still on screen depending on how many random coins were spawned.

We can fix this by moving the call to `this.spawnCoins()` from `wrapBookcases()` to `teleportBackwards()`. This gives us a predictable size of `2380` pixels that should be enough for 20 spawned coins.

Add this change by deleting the call to `this.spawnCoins()` in `wrapBookcases()` and add it to `teleportBackwards()` like this:

```
private teleportBackwards()
{
    const scrollX = this.cameras.main.scrollX
    const maxX = 2380

    if (scrollX > maxX)
    {
        // previous code...

        // spawn coins here
        this.spawnCoins()

        // continue to perform teleport logic for coins
        this.coins.children.each(child => {
            const coin = child as Phaser.Physics.Arcade.Sprite
            if (!coin.active)
            {
                return
            }

            coin.x -= maxX
            const body = coin.body as Phaser.Physics.Arcade.StaticBody
            body.updateFromGameObject()
        })
    }
}
```

Test the game out for a bit and everything should now be working as one would expect. It's a real infinite runner with some challenge!

Optional Animation Updates

Phaser 3.50 introduced the idea of animations that can be specific or local to individual sprites.

So far we've created global animations that can be used by any `Sprite`. But you'll notice that they are only used by Rocket Mouse. While there's nothing wrong with keeping these animations global it would be convenient to have everything together in the `RocketMouse` class.

Let's start by moving the animations in the `Preloader` Scene into a `createAnimations()` method in the `RocketMouse` class.

```
// imports...

export default class RocketMouse extends Phaser.GameObjects.Container
{
    // other code...

```

```
7       private createAnimations()
8       {
9           this.mouse.anims.create({
10              key: AnimationKeys.RocketMouseRun,
11              frames: this.mouse.anims.generateFrameNames(TextureKeys.
                    RocketMouse, { start: 1, end: 4, prefix: '
                    rocketmouse_run', zeroPad: 2, suffix: '.png' }),
12              frameRate: 10,
13              repeat: -1
14          })
15
16          this.mouse.anims.create({
17              key: AnimationKeys.RocketMouseFall,
18              frames: [{
19                  key: TextureKeys.RocketMouse,
20                  frame: 'rocketmouse_fall01.png'
21              }]
22          })
23
24          this.mouse.anims.create({
25              key: AnimationKeys.RocketMouseFly,
26              frames: [{
27                  key: TextureKeys.RocketMouse,
28                  frame: 'rocketmouse_fly01.png'
29              }]
30          })
31
32          this.mouse.anims.create({
33              key: AnimationKeys.RocketMouseDead,
34              frames: this.mouse.anims.generateFrameNames(TextureKeys.
                    RocketMouse, { start: 1, end: 2, prefix: '
                    rocketmouse_dead', zeroPad: 2, suffix: '.png' }),
35              frameRate: 10
36          })
37
38          this.flames.anims.create({
39              key: AnimationKeys.RocketFlamesOn,
40              frames: this.flames.anims.generateFrameNames(TextureKeys.
                    RocketMouse, { start: 1, end: 2, prefix: 'flame', suffix
                    : '.png'}),
41              frameRate: 10,
42              repeat: -1
43          })
44      }
45  }
```

Notice that instead of `this.anims.create()` we are using `this.mouse.anims.create()`. This will create animations just for the `this.mouse` instance. We are also using `this.mouse.anims.generateFrameNames()` instead of just `this.anims.generateFrameNames()`.

Next, we have to call `createAnimations()` from the `constructor()` like this:

```
constructor(scene: Phaser.Scene, x: number, y: number)
{
    super(scene, x, y)

    this.mouse = scene.add.sprite(0, 0, TextureKeys.RocketMouse)
        .setOrigin(0.5, 1)

    this.flames = scene.add.sprite(-63, -15, TextureKeys.RocketMouse)

    this.createAnimations()

    this.mouse.play(AnimationKeys.RocketMouseRun)
    this.flames.play(AnimationKeys.RocketFlamesOn)

    // other code
}
```

The `createAnimations()` method needs to be called after **`this`**`.mouse` and **`this`**`.flames` are created but before `play()` is called. The key is to split the chained method calls of `setOrigin()` and `play()` so that we can create the sprites, then the animations, and then play the starting animations.

Remember to delete the animation code from `Preloader` as we no longer need it!

Give these new changes a try to make sure that everything is still be working.

You now have a functional infinite runner game created in Phaser 3 using TypeScript.

There's still more you can do like adding sound effects and music as well as custom fonts for the UI.

We won't be covering that in this book but check out the Ourcade Blog and YouTube Channel for more content to help you add some finishing touches to your infinite runner!

Epilogue

Thank you for reading this book!

I hope it will give you a head start into making more scalable games with Phaser 3 and TypeScript.

Perhaps you've also learned some things related to game development that can be taken with you to other frameworks and platforms.

The basic gameplay and mechanics of Rocket Mouse were modeled after Jetpack Joyride and there's a lot more you can add like vehicles and missiles.

There are many resources online for art assets and programming tutorials to help you add those features.

We used Game Art Guppy in this book but there's also Kenney.nl, Open Game Art, and Itch.io. Just do a Google search for even more options!

There's never been a better time or a world with more resources for making games.

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

The process of making games should be pleasant and enjoyable at every skill level.

That's why Ourcade believes in 5 core principles:

- Playfulness
- Friendliness
- Helpfulness
- Open-mindedness
- Optimism

Join us if you find yourself nodding in agreement!

Find us on Twitter @ourcadehq.

Our YouTube Channel strives to show and explain the process of game development without editing away the problem solving.

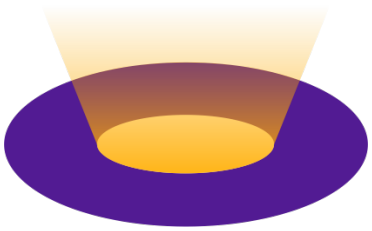
Github is where we share source code for anyone to read, learn, or steal. We don't believe that programming code should be protected like nuclear launch codes.

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
May 2020, *revised January 2021*



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 an avid weight lifter.

He resides in Colorado, USA and has 2 greyhounds.