Up: Index

Phaser 3 Rocket Patrol Tutorial

The purpose of this tutorial is to use Phaser 3 to recreate *Rocket Patrol* (1978), a simple gallery shooter included as a pack-in game for the obscure APF MP-1000 console. If you're not familiar with the game, you can check out a short demonstration on YouTube. The object of the game is to destroy as many spaceships as you can in one minute. The APF version has an asynchronous two-player mode and a player-versus-computer mode. For this tutorial, we're going to construct a single-player mode.

Project Overview

Before we start, it's important to understand the scope of the project and what we need to design.

Scenes

Rocket Patrol has two scenes: the *main menu* and the *game* itself.

Assets

Rocket Patrol has fairly minimal assets. For graphics, we'll need:

- a rocket (the thing the player can move and fire)
- a spaceship (the things we're shooting)
- an animated explosion (for when spaceships are destroyed)
- a starry background (to scroll in the background)
- some shape primitives to draw the UI

For audio, we need sounds for:

- rocket firing
- rocket/spaceship collision
- UI selection

Implementation Details

The following list outlines the primary components and systems we'll need to implement to make the game functional:

- One-minute timer
- Score tallying and display
- Menu with player input

- Rocket left/right movement input and fire button
- Collision detection with spaceships
- Spaceship spawning
- Spaceship horizontal movement
- Game reset

Learning Goals

This tutorial introduces the following concepts:

- Setting up a Phaser 3 project
- Running a local server
- Phaser game architecture
- Phaser Scenes & Scene management
- Phaser sprites, tile sprites, and sprite sheets
- JavaScript functions and objects
- JavaScript classes and inheritance
- JavaScript scope and context
- JavaScript arrow functions
- Phaser Shapes
- Phaser Keyboard events
- AABB collision detection and handling
- Phaser animation
- Phaser timers
- Phaser audio

Expertise

This tutorial assumes you understand general programming concepts (e.g., variables, functions, classes, etc.) and have some familiarity with JavaScript (ES6). I try to provide succinct explanations for each new concept or links to relevant learning resources.

Time

If you have the necessary expertise, the tutorial should take about 2–4 hours to complete.

Software Tools

This tutorial is written in ES6 JavaScript using the Phaser 3 framework. To follow along, you will need to download Visual Studio Code (aka VS Code) and have python installed on your computer. To implement version control, it's also recommended that you download git and set up a GitHub account.

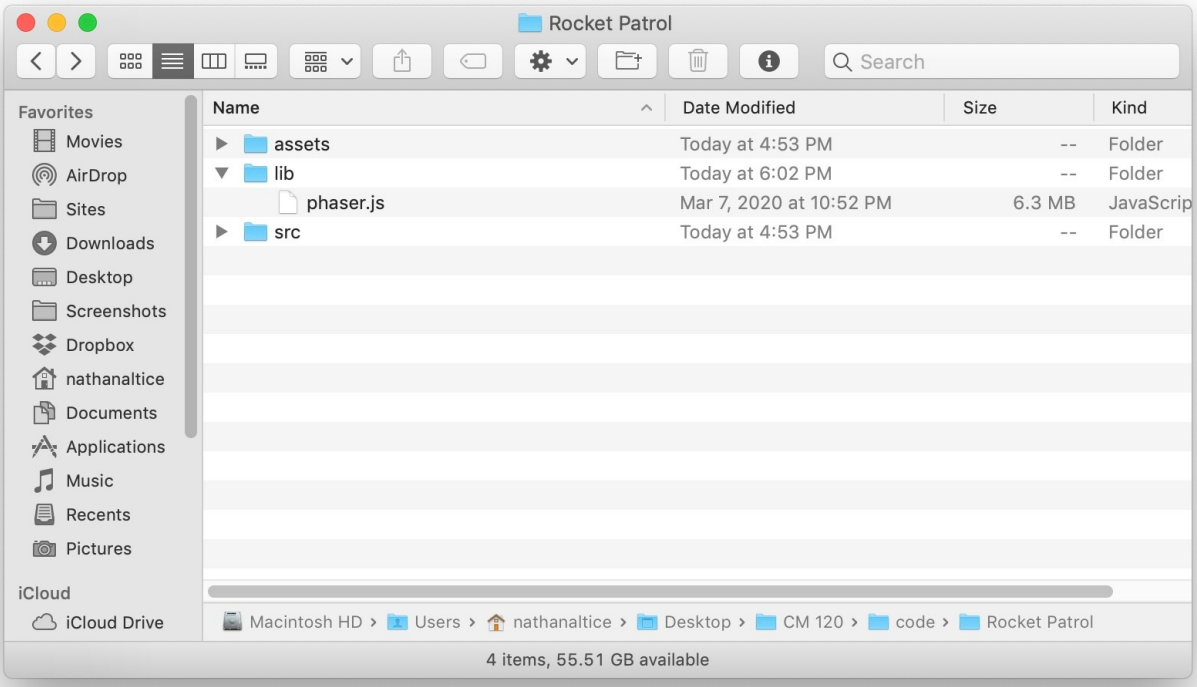
Getting Started

Folder Structure

Before we start programming, we need to set up our basic folder structure so we can keep our code (and other dependencies) organized:

- Create a new folder with the project name “Rocket Patrol,” and create 3 sub-folders within that folder: “src” for our source code, “lib” for our library (ie, the Phaser framework), and “assets” for graphics and sound.
- You can put this project folder anywhere, but I’d suggest keeping all of your programming projects in a single location. I, for instance, have a folder called “code,” where I keep all of my projects. Good organizational habits will benefit you (and any potential teammates) as your projects get more complex.
- If you haven’t already, download the Phaser framework and move it into the “lib” folder.

Your folder structure should look something like this (on OSX):



Creating index.html

Phaser helps us make browser-based web games. So before we do anything else, we need a “home” page that will serve as the container for our entire project. This page will reference all of our *external dependencies*—ie, the JavaScript files that make our game run. By default, web browsers look for a page called `index.html` at the root of any web site. The index is the hub of most web pages. So we’re going to start our project by creating `index.html` first.

Open VS Code and drag your Rocket Patrol folder on top of it. VS Code will open the folder for you. (If VS Code still has a Welcome tab open, feel free to dismiss it.) For me, using this “folder-first”

approach is the best way to keep your VS Code projects organized.

In the root folder (ie, **not** in “src,” “lib,” or “assets"), create an `index.html` file by clicking the New File icon (see image below) in VS Code and typing the file name (index) and extension (.html). `index.html` should now be in the "OPEN EDITORS” section of VS Code (in the left column), so we can start typing some HTML.

Note: The New File icon is the leftmost icon in the image above. It looks like a sheet of paper with a plus sign on the bottom right, and it’s located in a small bar in the EXPLORER panel in VS Code. Keep in mind that you can also create a new file by selecting File -> New File. 👍

VS Code can save us some time by constructing a basic HTML template for us. In the `index.html` file, type an exclamation point (`!`) and press Enter/Return. VS Code will generate a “minimum viable” web page for us.

For now, don’t worry about the various HTML tags, but if you want to give your web page a title that will appear in a browser tab, replace the text “Document” that’s in between the `<title>` tags to “Rocket Patrol” (see the screenshot in the next section below).

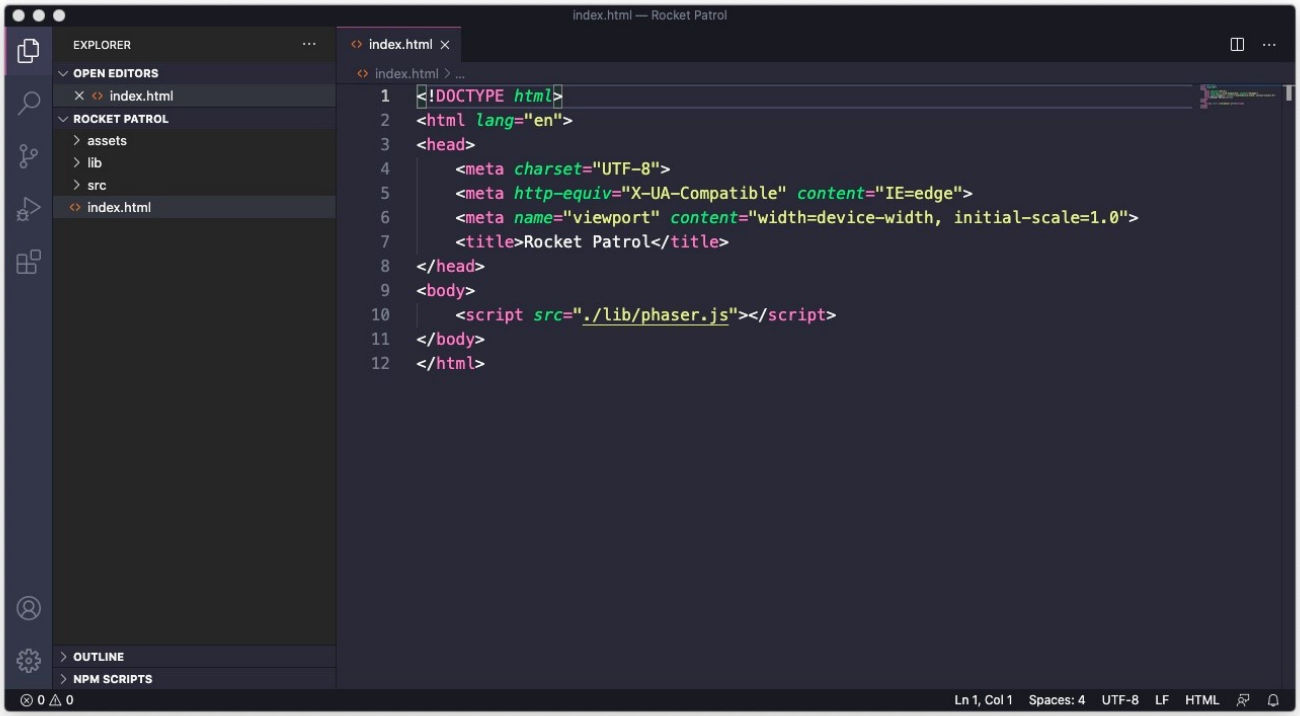
Referencing the Phaser library

In order for our source code to see and use the Phaser framework, we need to create a reference to the `phaser.js` framework file. To do so, type the following in between the `<body></body>` tags:

```
1 <script src=“./lib/phaser.js”></script>
```

This tag tells our index that there is an external script file with the name `phaser.js` inside a folder called “lib,” *relative to the current root directory*.

Your `index.html` file should now look like this:



Note: I use the Dracula Soft VS Code color theme, so my code colors may look different than yours.

Creating main.js

Now we need an entry point into our JavaScript code, so we're going to select the "src" folder within VS Code, create a New File, and name it `main.js`. For testing purposes, type the following code into `main.js`:

```
1 console.log("hello world");
```

The `console.log()` function allows us to "print" messages to the JavaScript Console, a browser tool provided to developers to help debug their code. The Console is essential to web programming, and you'll use it extensively while building web games.

Go back to `index.html`, and *below* our phaser.js `<script>` tag, create another reference, this time pointing to the `main.js` file we just created:

```
1 <script src = "./src/main.js"></script>
```

Testing the Local Web Server

In order to test our game, we need to run a local web server.

In VS Code's menu, select Terminal -> New Terminal to bring up VS Code's Terminal window. (It will appear in a horizontal window beneath your source code, as shown below.) If you haven't used a Terminal program before, don't worry—there's nothing scary about it. A Terminal is a *command-line*

interface to your computer, meaning that you can control the computer using text alone. Before graphical user interfaces, this was how computers were used.

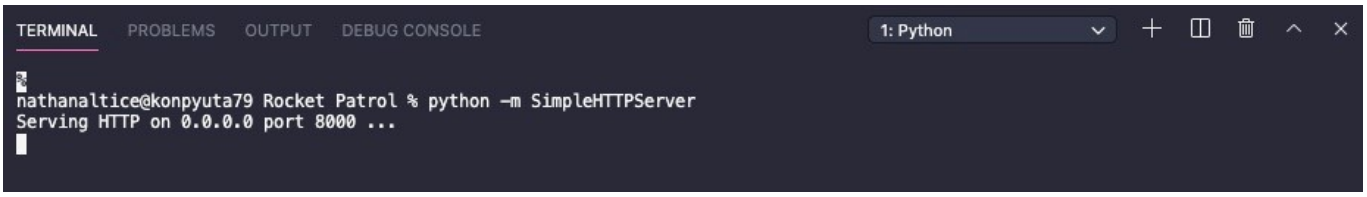
At the c*ommand prompt* (the little rectangle cursor) in the Terminal window, type `python -V` (or `python --version` or `py -3 –version`) to see what version of python you have installed. (If you haven't already installed python, you should do so now.)

Next, the command you type to start a local web server will vary according to (a) your operating system and (b) the version of python you have installed:

- On Mac OSX w/ python 3.X, type: `python3 -m http.server` or `python -m http.server`
- On Mac OSX w/ python 2.X, type: `python -m SimpleHTTPServer`
- On Windows w/ python 3.X, type: `python -m http.server`
- On Windows w/ python 2.X, type: `python -m SimpleHTTPServer`
- On Windows 10 w/ python 3.X, type: `py -3 -m http.server`

Note: It doesn't hurt anything to try different commands, so don't worry about messing up your computer if you type the wrong thing.

Once you've entered the proper command, press Enter/Return. The Terminal window should respond by outputing something like, "Serving HTTP on 0.0.0.0 port 8000 …" (see image below). If you have an error message, you likely (a) don't have python installed properly or (b) didn't type the proper command. Check your work and try again.



By default, starting up a local web server via python will run the contents of the target directory (ie, the "Rocket Patrol" folder) on port 8000 (the port isn't particularly important—8000 is the default for web development, and there's no real reason to change it). You can go to this server by typing the URL `http://localhost:8000` in your web browser. Navigating to this link will launch our game in the browser.

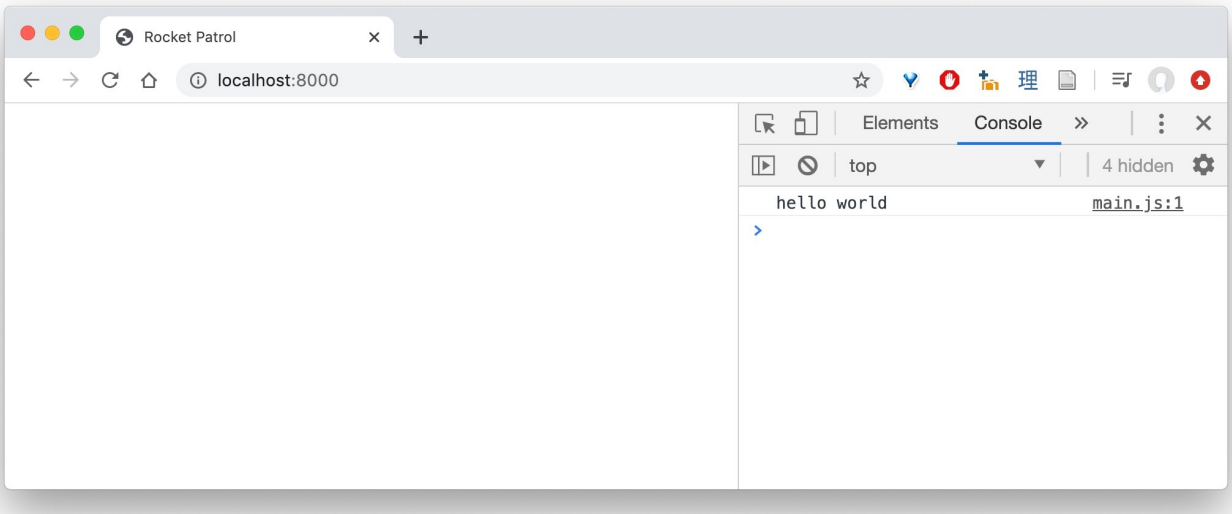
Note: Any time I tell you to "return to your browser" to check your program, be sure to save your files in VS Code. If you don't—you guessed it—you won't be able to see your changes.

Once you point your browser to `localhost`, you're only going to see a blank white page. This is

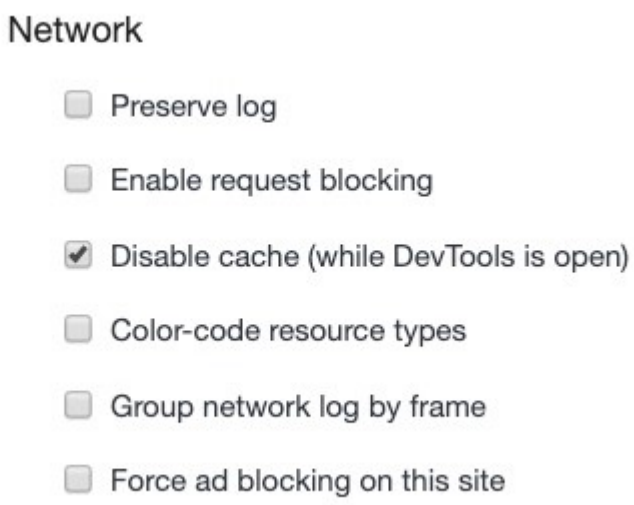
because we only logged the “hello world” text to the Developer console, so we need to open that console to see our output.

To do so in Chrome, press Ctrl+Shift+I (or select the stacked three-dot menu and choose More Tools -> Developer tools), then choose the “Console” tab.

If all went well, you should see the text `hello world` in the Console window:



Note: While you’re in Chrome’s Developer view, click the Preferences menu, scroll down to Network, and make sure the “Disable cache (while DevTools is open)” option is *checked* (see image below). Browsers like to cache website data so it doesn’t need to be loaded every time a user visits a page. This is *not* what we want to happen while we’re programming games. We want the page to reload every time we refresh the page.



Also note that running games while the DevTools are open gives us a slight performance hit. But in most cases, you should leave it open while developing so you will see any important debugging messages.

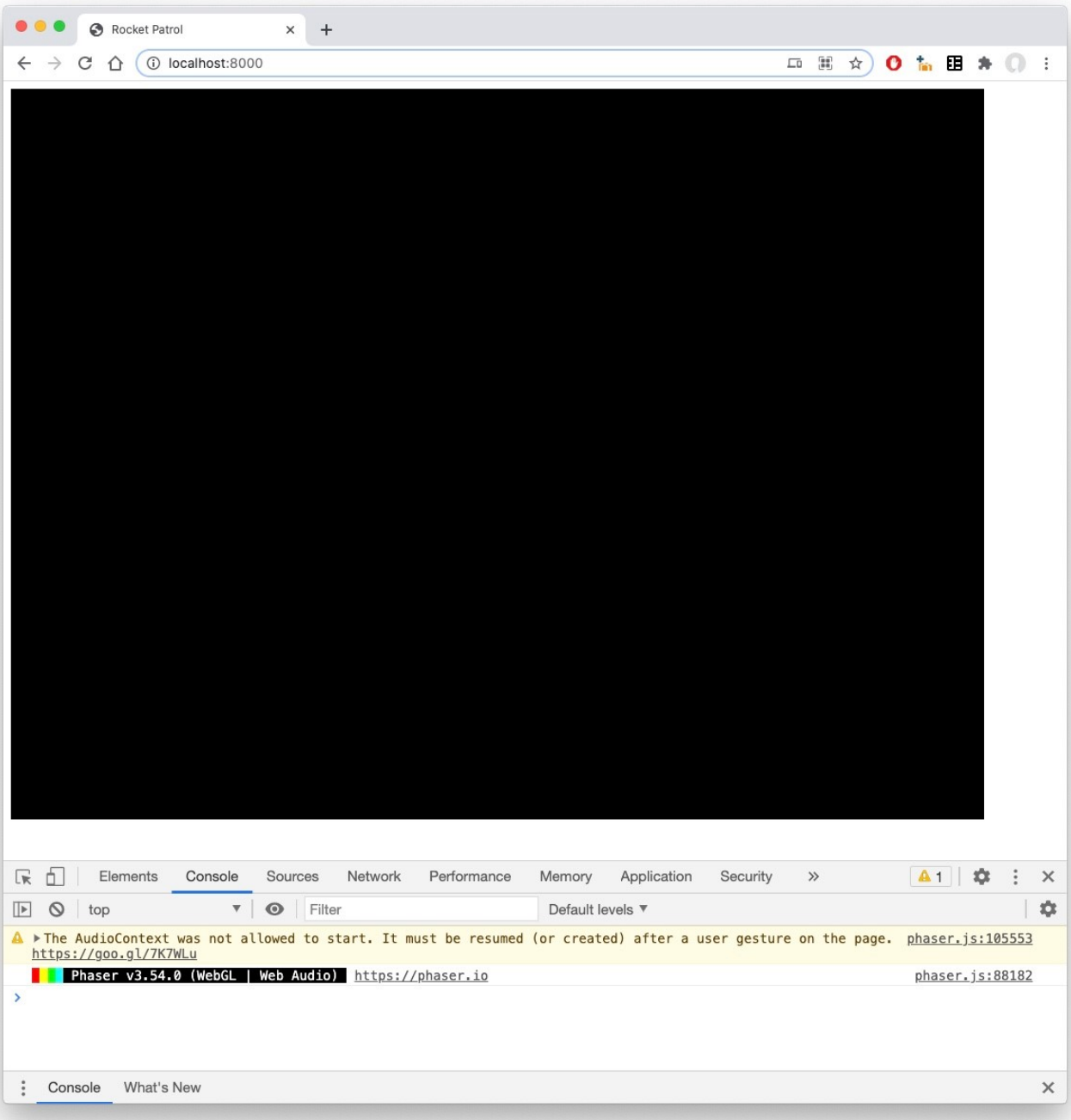
Testing Phaser

If everything worked correctly, return to VS Code, delete the `console.log()` statement, and type:

```
1 let game = new Phaser.Game();
```

Go back to the browser window and refresh the page to see the results.

You should see a black, rectangular box in the upper-left corner of the browser window and the Phaser version number beside a rainbow banner in the Console. (If you don't see those things, go back through the prior steps and check your work. Did you put the files in their proper folders? Did you reference all of the necessary files in `index.html`? Did you provide the proper file paths to those files? Did you type your code in properly, with no spelling mistakes or other errors? Did you invoke the proper arcane Gods of Programming?)



Note: You may a yellow warning message in the Console window that reads, “The AudioContext was not allowed to start. It must be resumed (or created) after a user gesture on the page.” Ignore this for now. There’s nothing wrong with your program. Google implemented a (really bad) policy a few years ago that requires direct user input before sound will play on a web page.

Creating a Game Configuration Object

Since we didn’t pass a parameter to the `Phaser.Game()` object, Phaser built us a game window with its default parameters (ie, 1024x768 pixels). However, we have full control of how our game window can look and behave, including its size, background color, position, and many other properties. To take control of these parameters, we need to pass Phaser a *configuration object* with the properties we

want to change.

In JavaScript, objects are lists of property:value pairs, much like a dictionary in other languages. We are going to construct an object literal, meaning we are going to provide a comma-delimited list of property names and their values and enclose them in curly braces ({}).

Type the following code *above* the game declaration statement we wrote before:

```
let config = {
  type: Phaser.CANVAS,
  width: 640,
  height: 480
}
```

Next, change the game declaration statement to:

```
let game = new Phaser.Game(config);
```

Once you save main.js and check the browser, you should see that our game window is now 640x480 pixels.

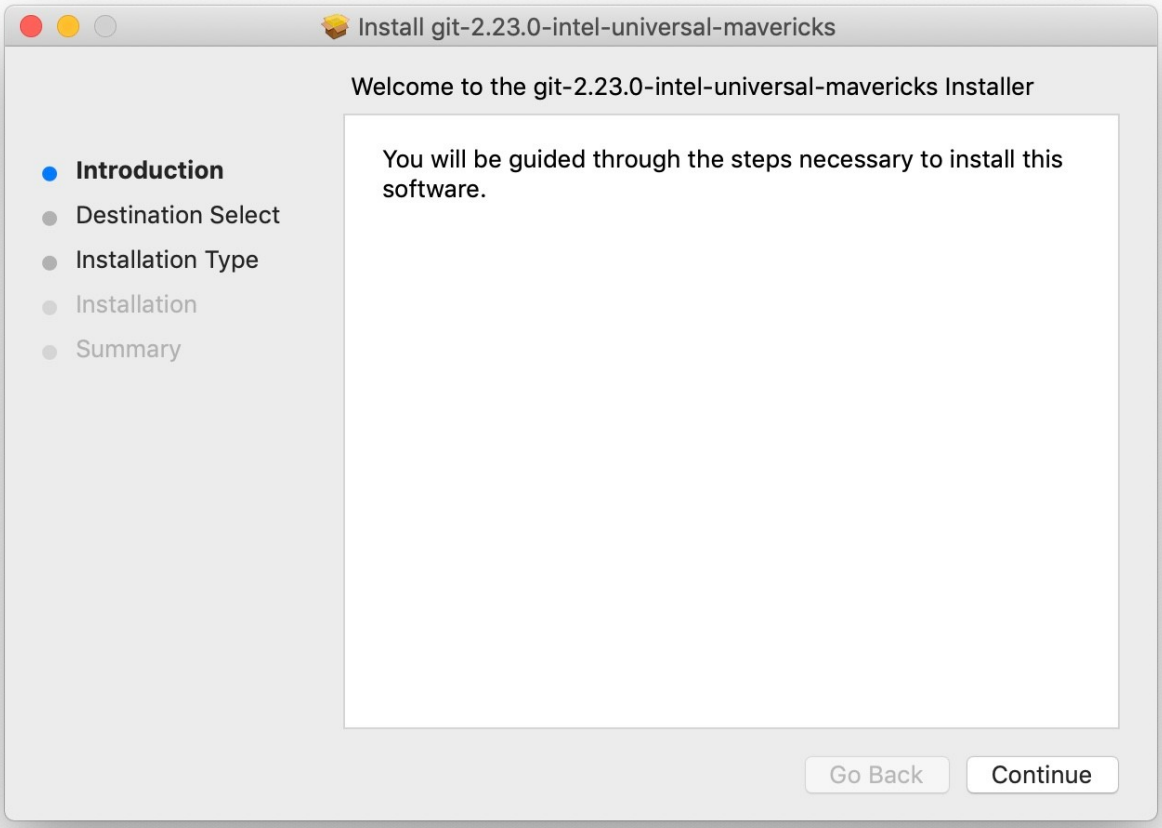
(Phaser gives us a lot of game configuration options. To see a comprehensive list, check out the official Phaser GameConfig documentation or Notes of Phaser 3.)

Setting Up Version Control

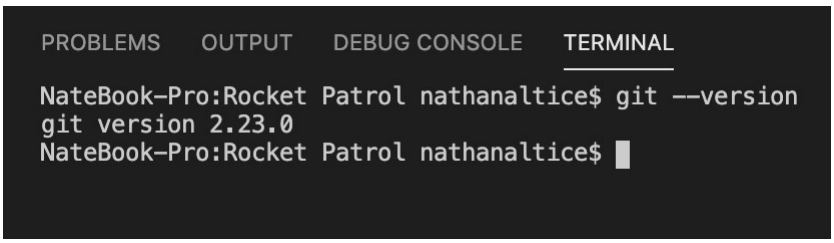
Version control tracks and manages your software project history, including collaboration with team members. There are many version control solutions you can use, but we're going to use git, a "free and open source distributed version control system," along with GitHub, a service that provides git repository hosting along with a social web interface for managing projects.

Getting git

If you don't already have git installed, close VS Code, then download and install the latest version of git (it's fine to just stick with the defaults during setup).



Now, re-open VS Code and open the Terminal window (Ctrl + Shift + ~) like you did above (also, if Code did not reopen your project folder, be sure to do so before you continue). In the Terminal window, verify your git install by typing: `git --version`. Terminal will output your git version number, similar to the screenshot below. If you get an error, something went wrong in the installation, so double-check your work.



Next, **before you do anything else**, set your git user name and email address by typing the following commands in the Terminal prompt:

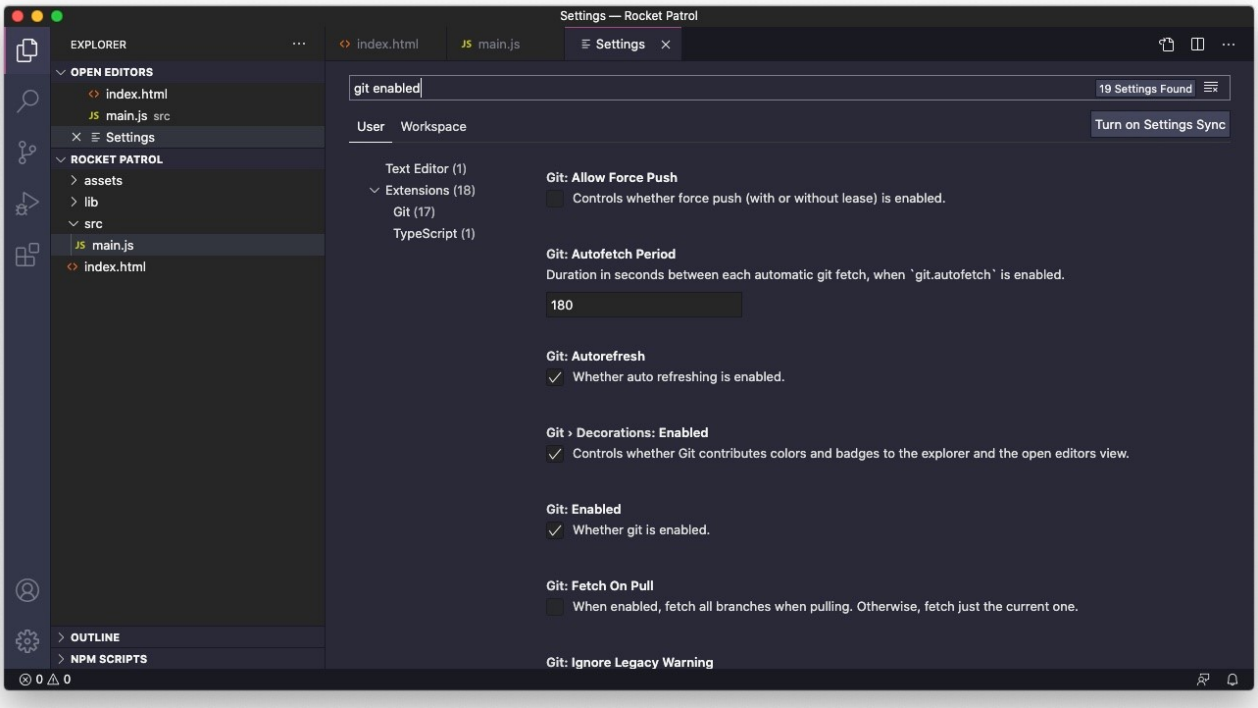
- `git config --global user.name "Your Name"`
- `git config --global user.email handle@email.com`

For more detailed info on git setup, see the following documentation page: "1.6 Getting Started - First-Time Git Setup."

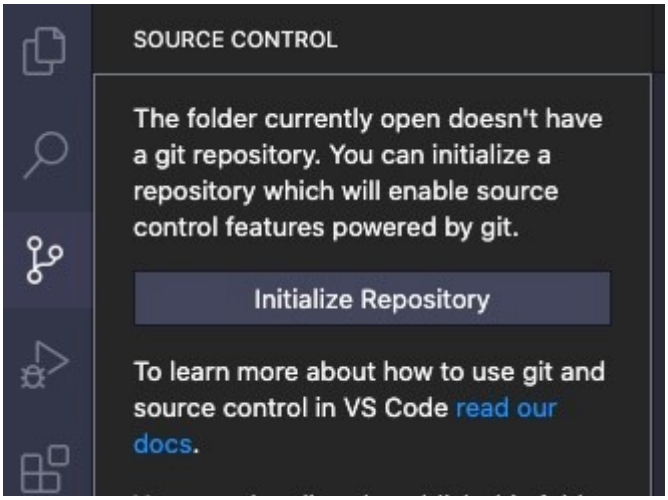
GitHub & VS Code

If you don't already have a GitHub account, set one up now. If you're a student, be sure to use the education portal so you can get a free pro account.

Once your GitHub account is set up, return to VS Code. If you're setting up version control integration in VS Code for the first time, be sure to click the Manage (⚙) icon at the lower left part of the window, choose Settings, and search for "git enabled" in the search bar. Then make sure the "Git: Enabled" option is checked, otherwise git won't work.



Once that's done, click the Source Control icon (it looks like three nodes connected by lines) on VS Code's far left panel.

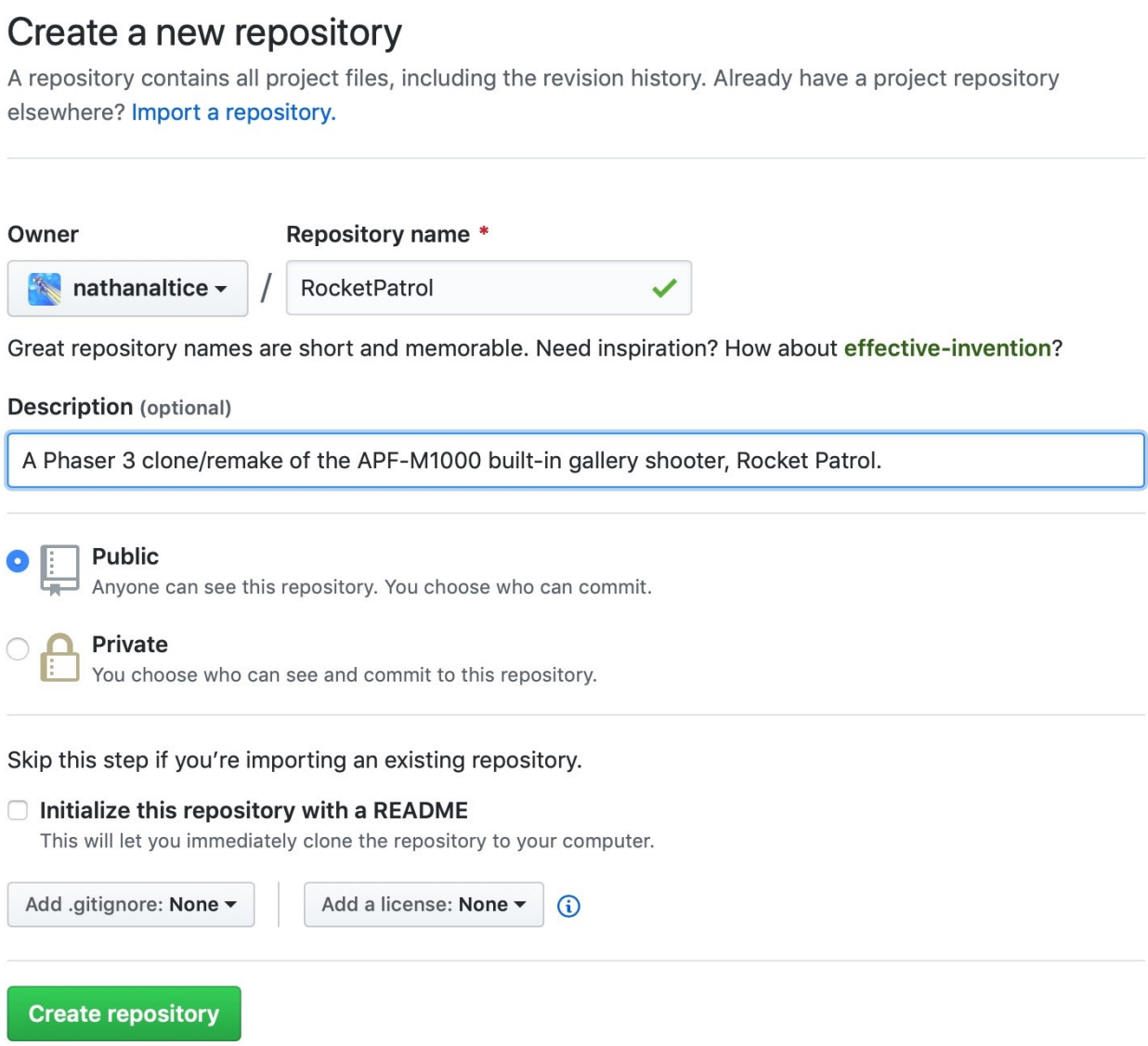


If there is no git repository set up, you'll see a big "Initialize Repository" button. CLICK DAT BUTTON. You should notice a few changes. First, the Source Control icon will now have a notification badge with the number of pending changes. Second, a list of files will appear, each with a green "U" beside them, indicating that they are "unstaged."

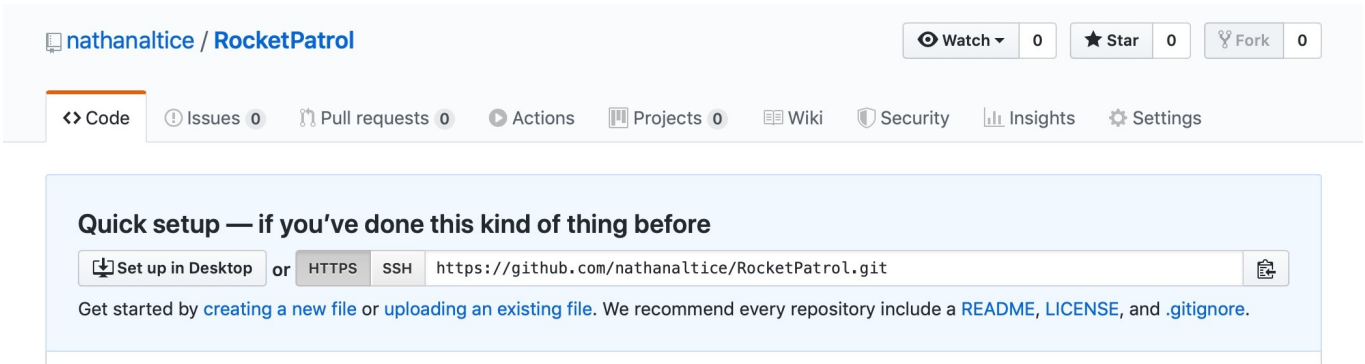
Type a *meaningful* commit message (eg, "Initial Commit") in the Message field and click the checkmark (✓) icon to Commit your files. (If this is your first time committing the files, you may see a message that says, "There are no staged changes to commit. Would you like to automatically stage all your changes and commit them directly?" You can choose to say "Yes" for a one-time stage/commit, or click "Always" to do this every time you commit.)

Note: If you forgot to set your git user name and email as instructed above, you will get a "Make sure you configure your 'user.name' and 'user.email' in git" message. GO BACK AND DO THAT.

Now that our files are committed, we can push them to GitHub. Log in to your GitHub account and click the green "New" button to create a new repository to store your project. Once you're in the "Create a new repository" screen (see image below), type in a name for your repository (e.g., "Rocket Patrol"), type in a description, then click the green "Create repository" button.



On the next screen, you'll see a Quick setup box with a URL for your repository. Copy that line by highlighting the URL or clicking the small clipboard icon to the right of the URL. We'll need that URL in VS Code.



Return to VS Code and select View -> Command Palette…, type "git remote," and choose "Git: Add Remote." Paste the URL of your repository (from above), press Enter again, then give your repository a name, and press Enter again. We've now connected our project in VS Code to the remote repository

("remote," as in, on GitHub).

Now we need to *push* our local code to the remote repository. To do so, we can either:

- click the `[...]` option in the SOURCE CONTROL menu and select "Push" **or**
- click the cloud icon at the bottom left of the VS Code screen (beside your current branch name, i.e., "master").

Note: The cloud icon is the preferred method, because it forces you to enter your GitHub account credentials if you haven't already.

Note: On subsequent pushes, the cloud icon will now be circular arrows (see image below) meaning "Synchronize Changes."

To check that your repository was pushed properly, go back to GitHub and refresh the page. You should see all the beautiful, glorious code you typed into VS Code.

Note: If there are any files you *don't* want to commit, you can right-click those files in the Source Control list and "Add to .gitignore."

Good git Habits

It's a good idea to get into the habit of making regular git pushes as your project develops. I push changes every time I hit a *working* milestone, meaning code that both achieves some significant goal *and* works without major errors. Other programmers choose their own milestones, like finishing a day's work or adding new files. Your version control schedule is your own (or your team's) choice, just be sure to develop good habits *now* so you can save yourself from headaches *later*.

Scene Architecture

One of Phaser's foundational architectural concepts is the Scene. Just like in theater or film, a scene is an organizing structure that describes a set of related actors, settings, actions, etc. For instance, in Act 1 Scene 5 of Hamlet, "The ghost of Old Hamlet reveals to his son that he was murdered by Claudius and demands that young Hamlet seek revenge." This scene involves particular characters in a particular place speaking about a particular topic with particular stage direction and particular props.

Similarly, in a game, the main menu might be a scene, an individual level might be a scene, a cutscene is *obviously* a scene, and so on. Each of these involves particular graphics, particular sounds, particular interactions, etc. Phaser does not pre-define Scenes for you. How you structure your individual scenes is up to you. You are the Shakespeare of your game!

Setting the Scene

We're now ready to build the basic scene architecture for our game, meaning that we're going to

create the file structure and some rudimentary code to support *Rocket Patrol*'s two scenes: the menu scene and our main play scene.

- In VS Code, create a new sub-folder in your "src" folder by clicking the New Folder icon while the "src" folder is highlighted. Name this new folder "scenes"
- Within that "scenes" folder, create two new files: `Menu.js` and `Play.js`. (We're using upper-case names b/c scenes in Phaser are JavaScript classes, and in JavaScript naming conventions, classes get capitalized.)
- In `Menu.js`, type the following:

```
1 class Menu extends Phaser.Scene {
2     constructor() {
3         super("menuScene");
4     }
5 }
```

We're creating a JavaScript class named "Menu" (the same as our file name) that *extends* (ie, becomes a child of) Phaser's predefined Scene object. The `constructor` (a special method for creating and initializing an object) uses the `super` keyword to call the constructor of the super class, ie `Phaser.Scene`. In other words, when we construct a Menu object, we will use the constructor of the parent Scene object to construct it.

The "`menuScene`" parameter we're passing via `super` is a string key that we will use to identify this particular scene in our code. (The key name you use is up to you, but like all keywords and variables, it's best to name it something logical, descriptive, and easy to read.)

Scenes in Phaser have a predefined flow of functions that happen in sequence, as follows: `init()` -> `preload()` -> `create()` -> `update()`. Each function has a different role to play in the "lifespan" of a scene.

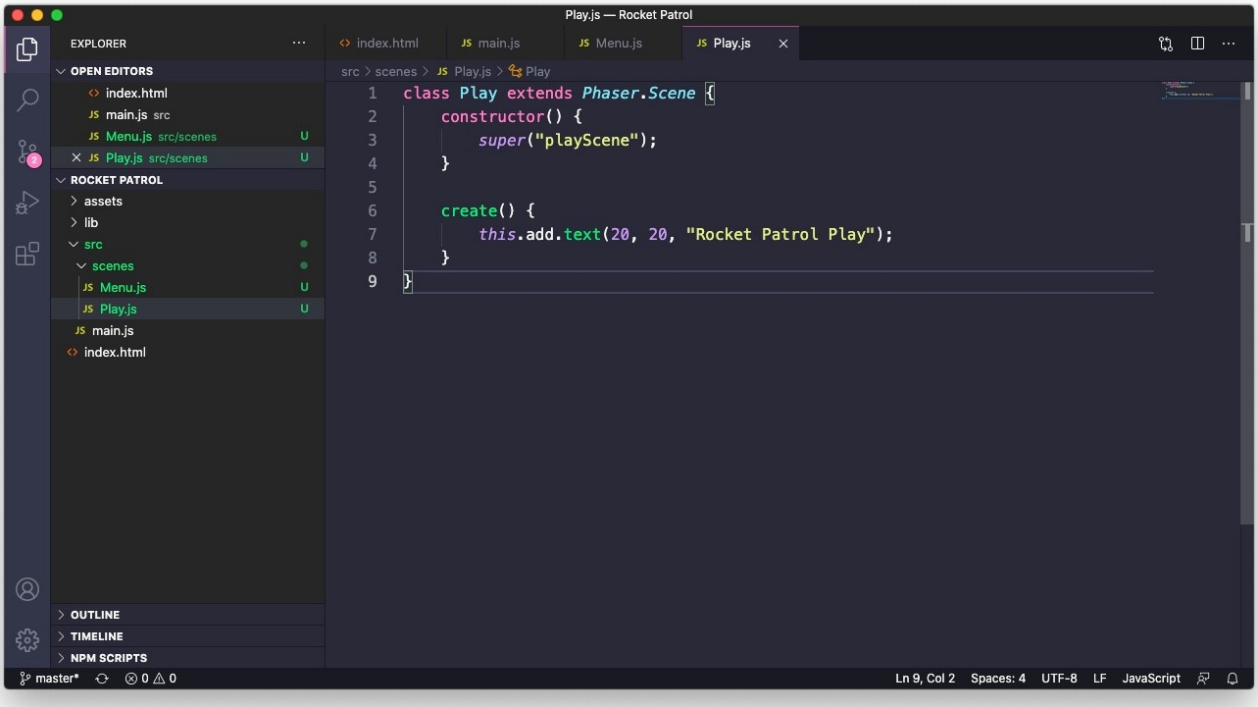
- `init()` prepares any data for the scene
- `preload()` prepares any assets we'll need for the scene
- `create()` adds objects to the scene
- `update()` is a loop that runs continuously and allows us to update game objects

However, Phaser Scenes only require *one* of those functions to work: `create()`.

In order to test our Menu scene, we're going to use the `create()` method to write some text on the screen. Below the `constructor()` method, add a new `create()` method so Menu.js now looks like this:

```
1 class Menu extends Phaser.Scene {
2   constructor() {
3     super("menuScene");
4   }
5
6   create() {
7     this.add.text(20, 20, "Rocket Patrol Menu");
8   }
9 }
```

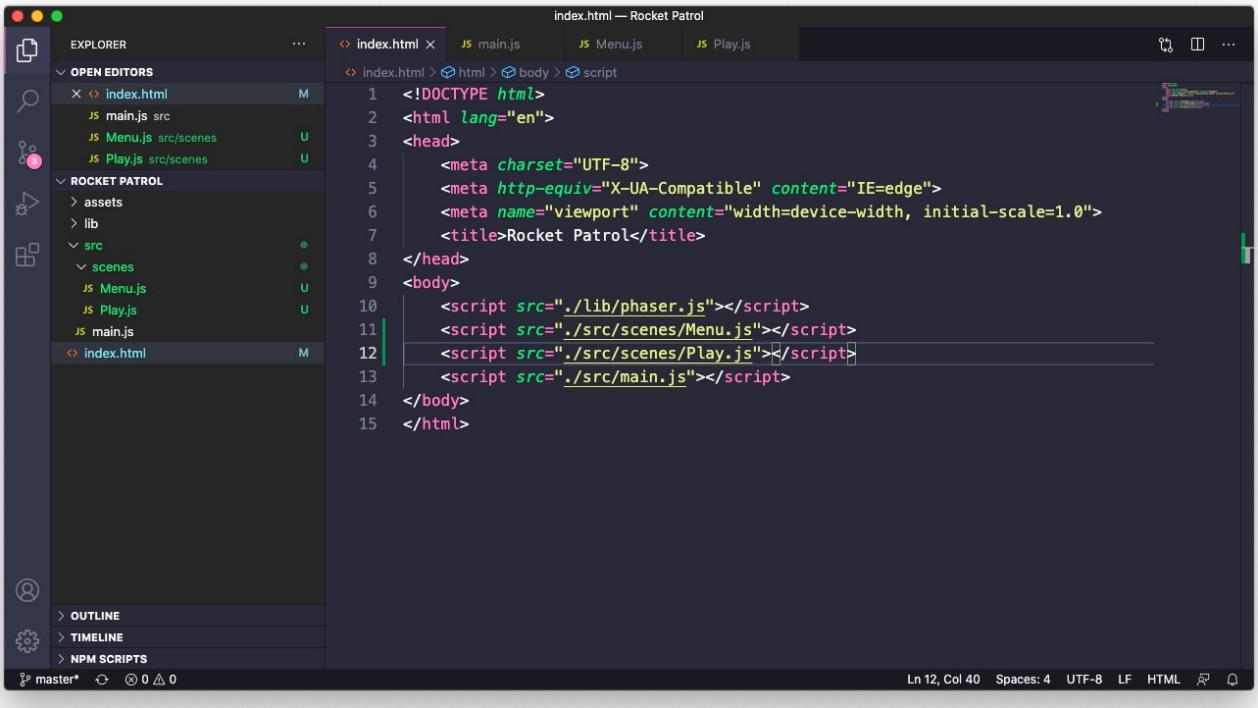
Now, copy all the code you typed in Menu.js, paste it into Play.js, and edit the text so it looks like the following:



If you're new to JavaScript, you might've noticed that odd keyword `this` while typing the above code. `this` is a confusing concept in JS, but the basic idea is that it's a special keyword bound to the current object *context*. In the example above, `this` references the `Scene` object. In other words, our statement is asking Phaser to add some text (the string "Rocket Patrol Menu") to *this* Scene object at the (x, y) coordinates (20, 20). The important thing to know for now is that `this` references different objects depending upon where you place it in your program.

If you save your file and go back to the browser, no text appears. That's because we haven't "hooked up" the scenes in our project. In other words, `index.html` can't "see" the new files in the "scenes" folder, and `main.js` doesn't know we've added any scenes. So let's make the proper hookups.

Back in `index.html`, reference the two new JavaScript files in the `<body>` tags *in between* the previous two `<script>` tags. The order of our JavaScript files is important, so make sure yours looks like the screenshot below:

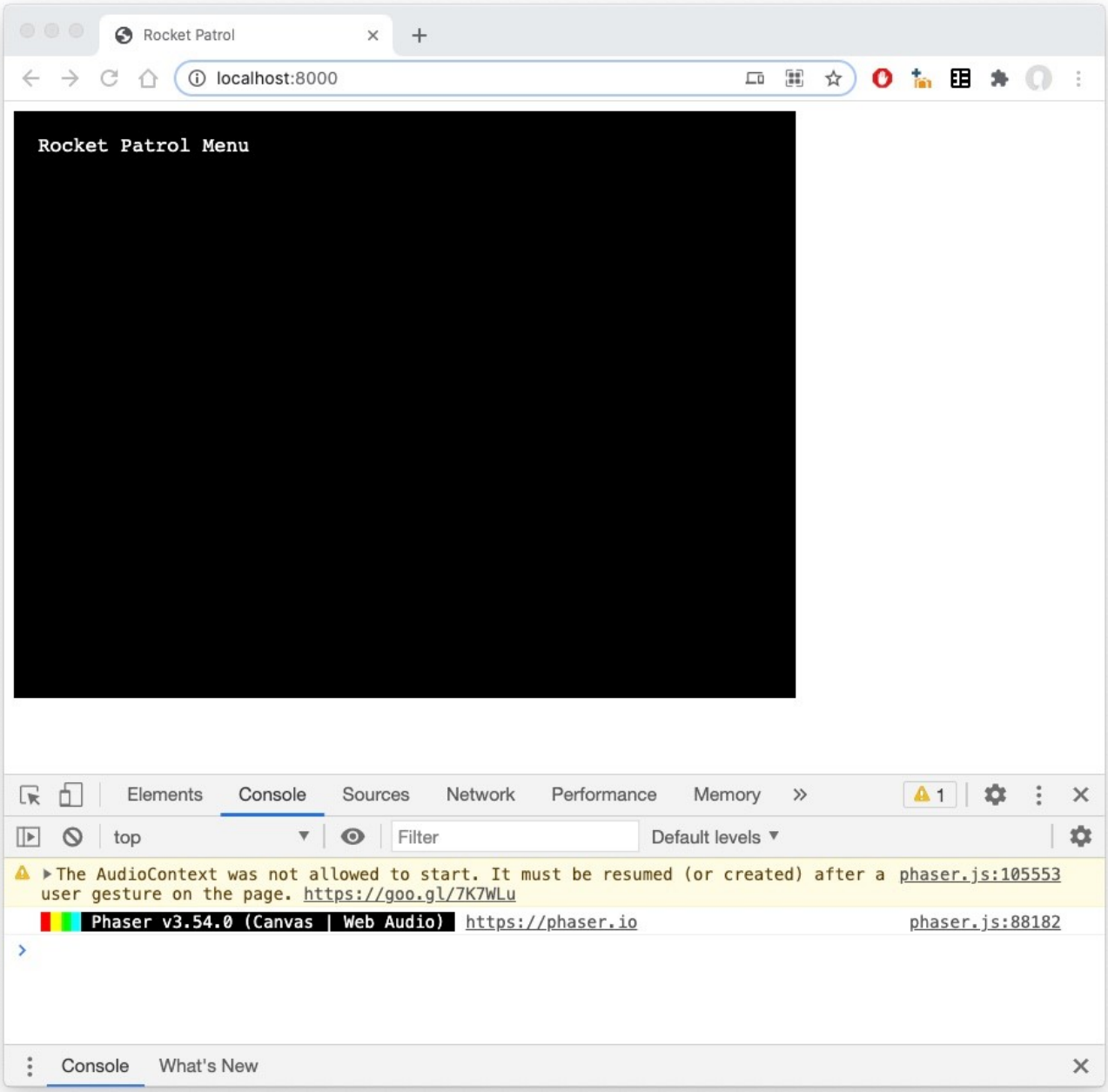


Next, go back to `main.js` and update the `config` object with a new line at the bottom:

```
let config = {
  type: Phaser.AUTO,
  width: 640,
  height: 480,
  scene: [ Menu, Play ]
}
```

The `scene` property expects an array with the object names of any Phaser scenes we've created.

If you now go back to the browser and reload, you should see the Menu text in the game window.



Changing Scenes

For now, we're going to leave the Menu scene for later and skip immediately to the Play scene. To do so, we're going to add one line of code to `Menu.js`, just below the `this.add.text` line in `create()`:

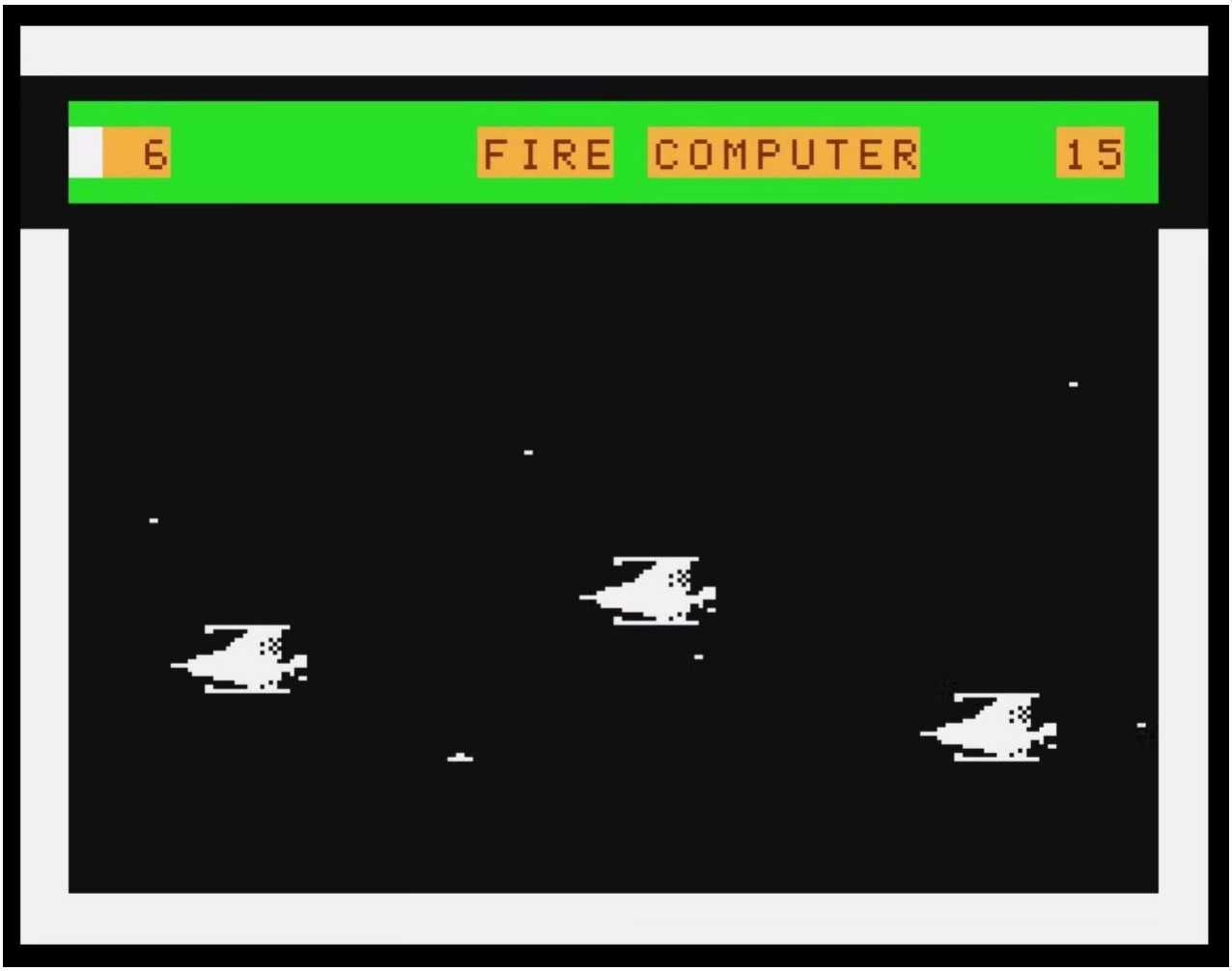
```
1 this.scene.start("playScene");
```

This bit of code tells the *current* scene to start a new scene, and the parameter we pass to the `start()` method is the string key we assigned to the Play scene.

Save, return to the browser, refresh, and you should see "Rocket Patrol Play" in the game window.

Constructing the Play Scene

Let's take a look at the original *Rocket Patrol*'s main play scene:



To start, we want to construct the white borders that surround the screen and the green bar that defines the UI section. To do so, we'll use some of Phaser's built-in shape primitives.

First, go to `main.js` and add the following code below the `game` definition:

```
// set UI sizes
let borderUISize = game.config.height / 15;
let borderPadding = borderUISize / 3;
```

Next, go to `Play.js` and in the `create()` method, delete the `this.add.text` line we typed before and replace it with the following code:

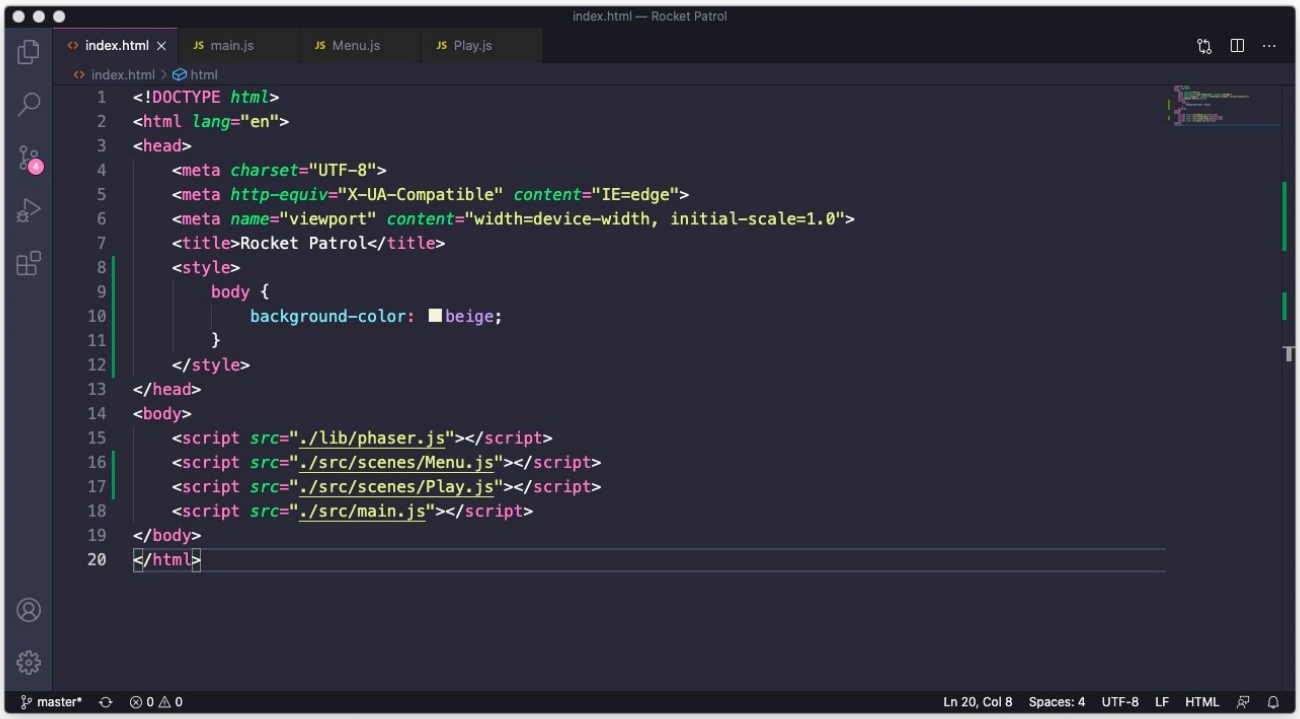
```
// green UI background
this.add.rectangle(0, borderUISize + borderPadding, game.config.width, borderUISize * 2,
0x00FF00).setOrigin(0, 0);
// white borders
this.add.rectangle(0, 0, game.config.width, borderUISize, 0xFFFFFF).setOrigin(0, 0);
```

```
this.add.rectangle(0, game.config.height - borderUISize, game.config.width, borderUISize, 0xFFFFFF).setOrigin(0, 0);
this.add.rectangle(0, 0, borderUISize, game.config.height, 0xFFFFFF).setOrigin(0, 0);
this.add.rectangle(game.config.width - borderUISize, 0, borderUISize, game.config.height, 0xFFFFFF).setOrigin(0, 0);
```

Wow, that's a lot! Let's review what we did. The `add.rectangle()` command tells Phaser to add a rectangle to our scene (referenced by `this`) with five parameters in a specific order: x-coordinate, y-coordinate, width, height, and color (in hexadecimal format). The `setOrigin` method chained to the end of each line tells Phaser to adjust the rectangle's origin—the point on the rectangle used to position it in coordinate space—from the default at its center to its upper left.

We've also done something *really smart* to help make our code more flexible. The size of the UI borders, their widths, their heights, and the padding between them are all defined by variables rather than being *hard-coded* to specific values. If, for instance, we decided to double the size of our game window to 1280x960, the UI would scale properly. Using dynamic values requires more typing and planning at the beginning, but it's worthwhile in the long run, especially for web games, where dimensions and scale may vary according to the player's device. Likewise, by placing them in main.js, we made them available in the global scope, so we can use these same variables throughout our program. This will be really handy later on.

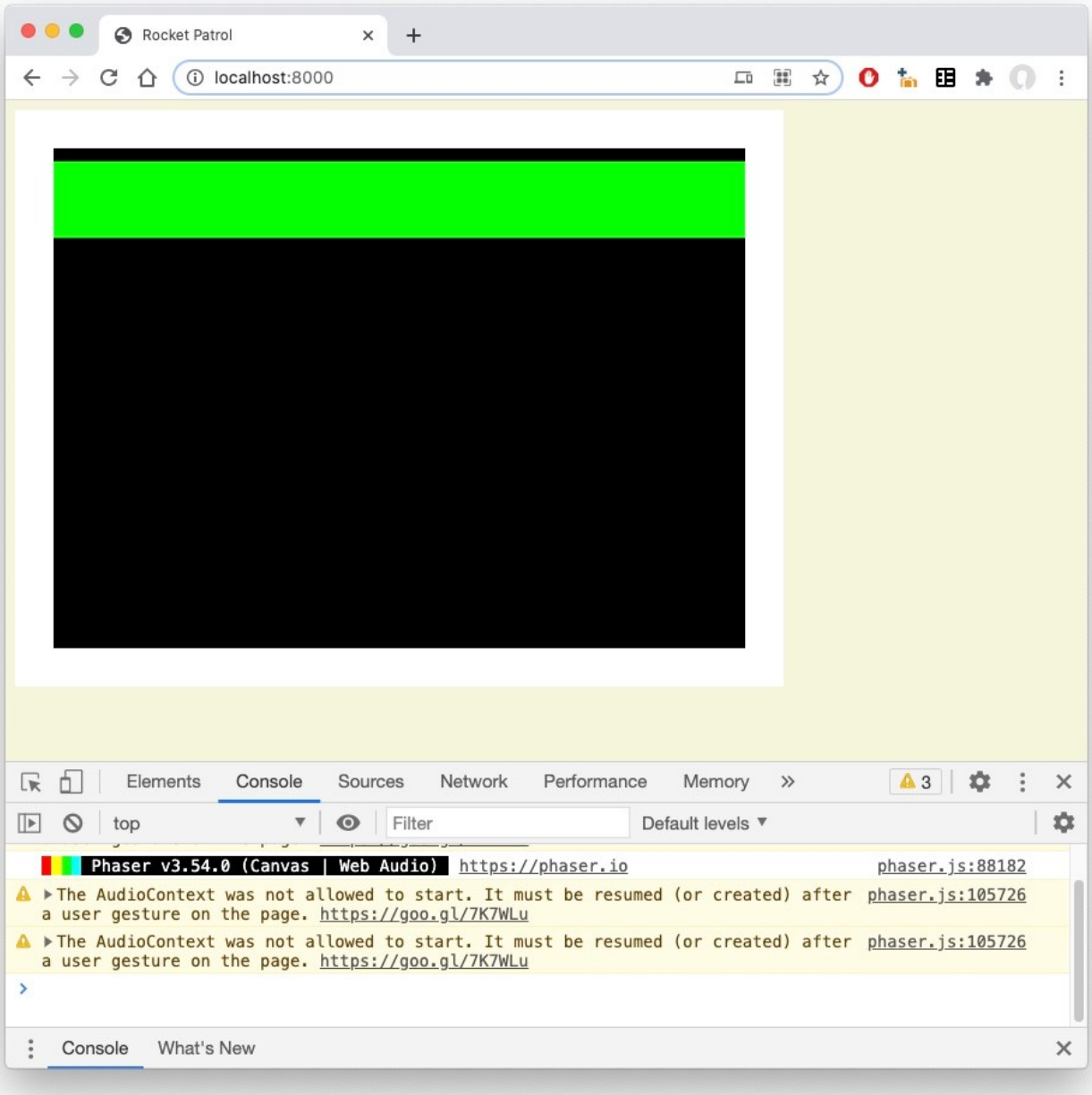
Before we check our work, let's add one last bit of decoration. Head back to `index.html` and type in some new `<style>` tag information as shown below:



Note: The little square to the left of the word `beige` is auto-added by VS Code as a tiny color swatch preview for the color value you've typed.

Since a web page's default background color is white, it makes it difficult to see *Rocket Patrol's* white

borders. What we've done is add three lines of CSS to help style our page. By changing the background color to beige, we're now able to see our pristine white game borders, like so:



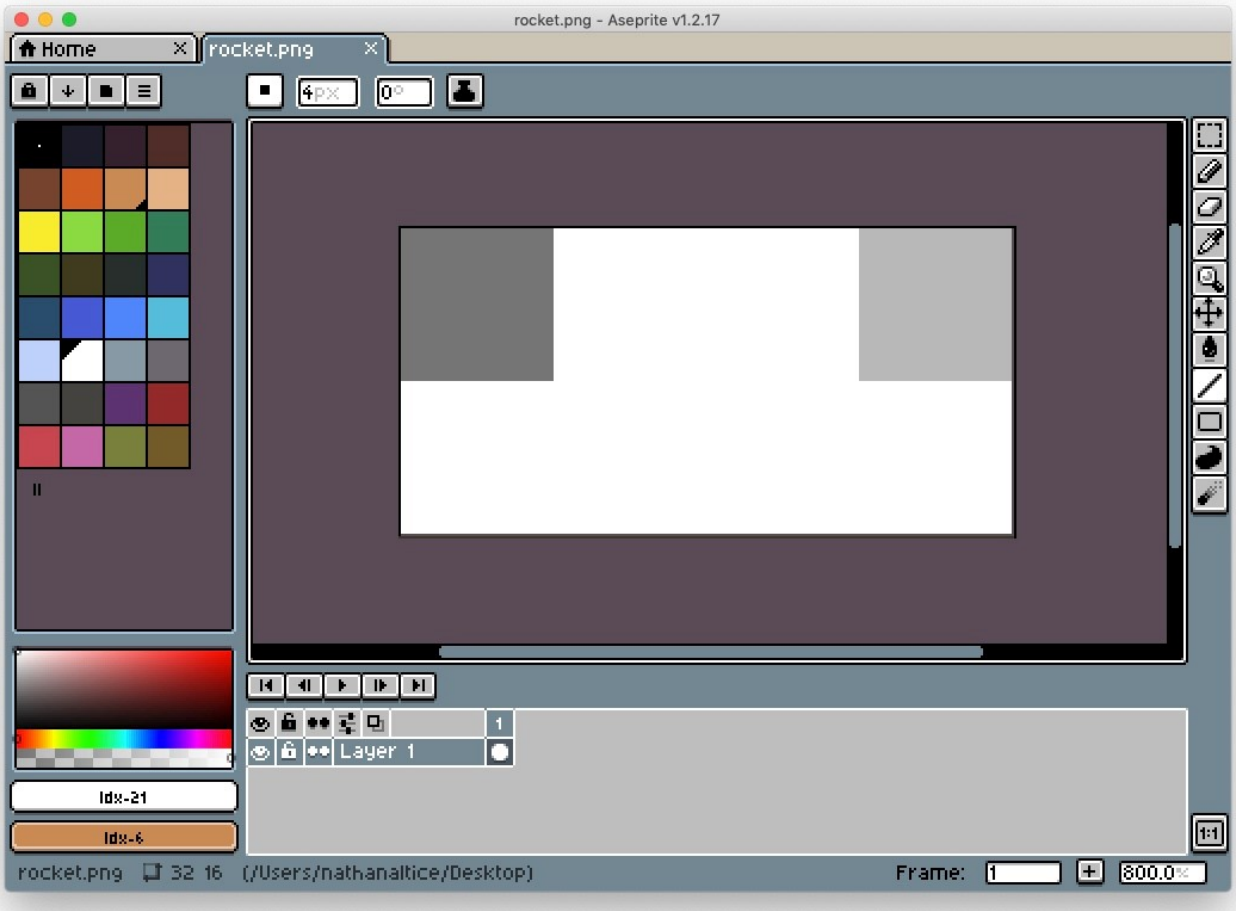
The end result isn't *exactly* like the original game, but it's close enough. Now we can start putting some moving objects on the screen. Let's start with the player-controlled rocket.

Creating & Adding Assets

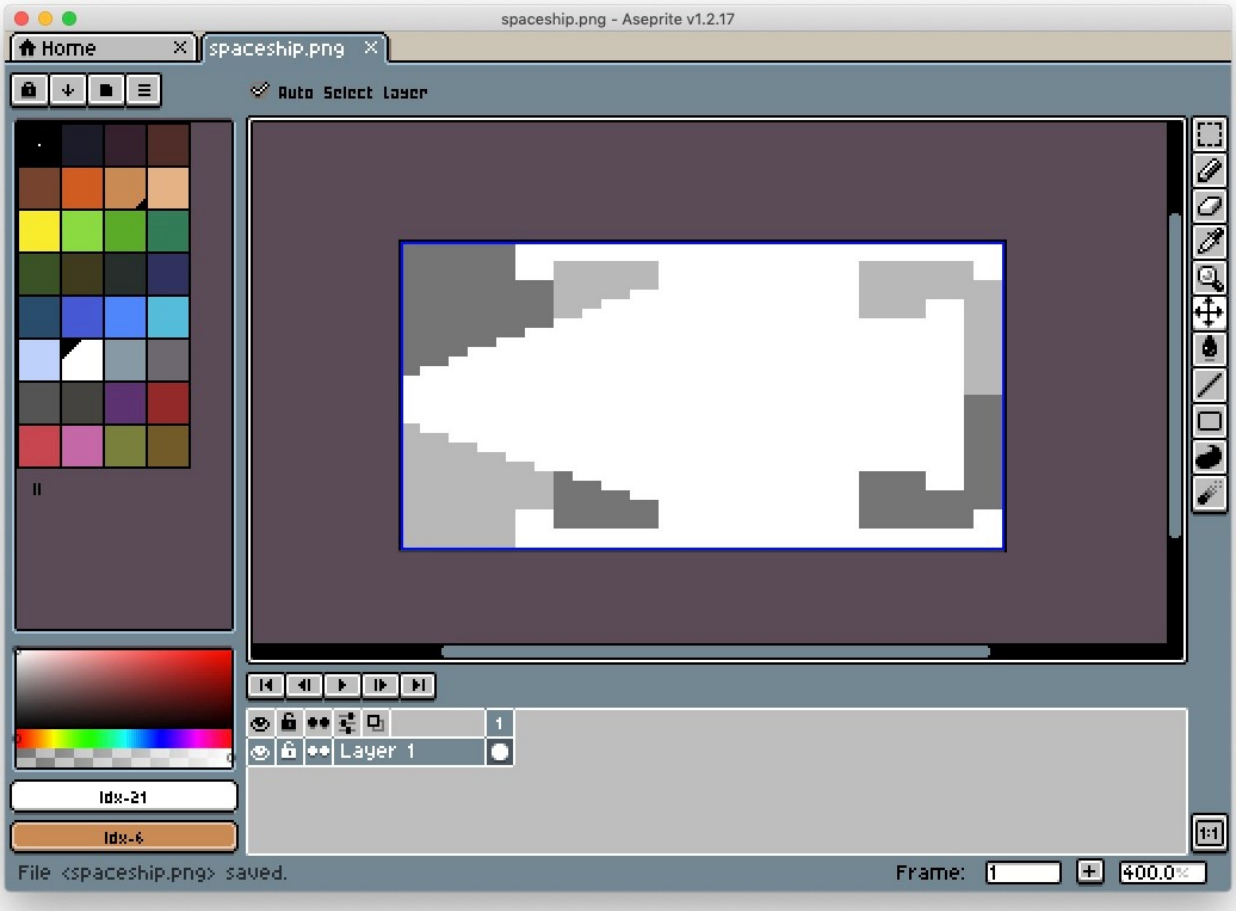
Note: If you don't want to create your own assets, you can download mine from the project's GitHub page.

We need some images to put on screen, so we're gonna have to do some artwork. I decided to stick with the basic pixel art aesthetic, so I used Aseprite to quickly replicate *Rocket Patrol*'s in-game

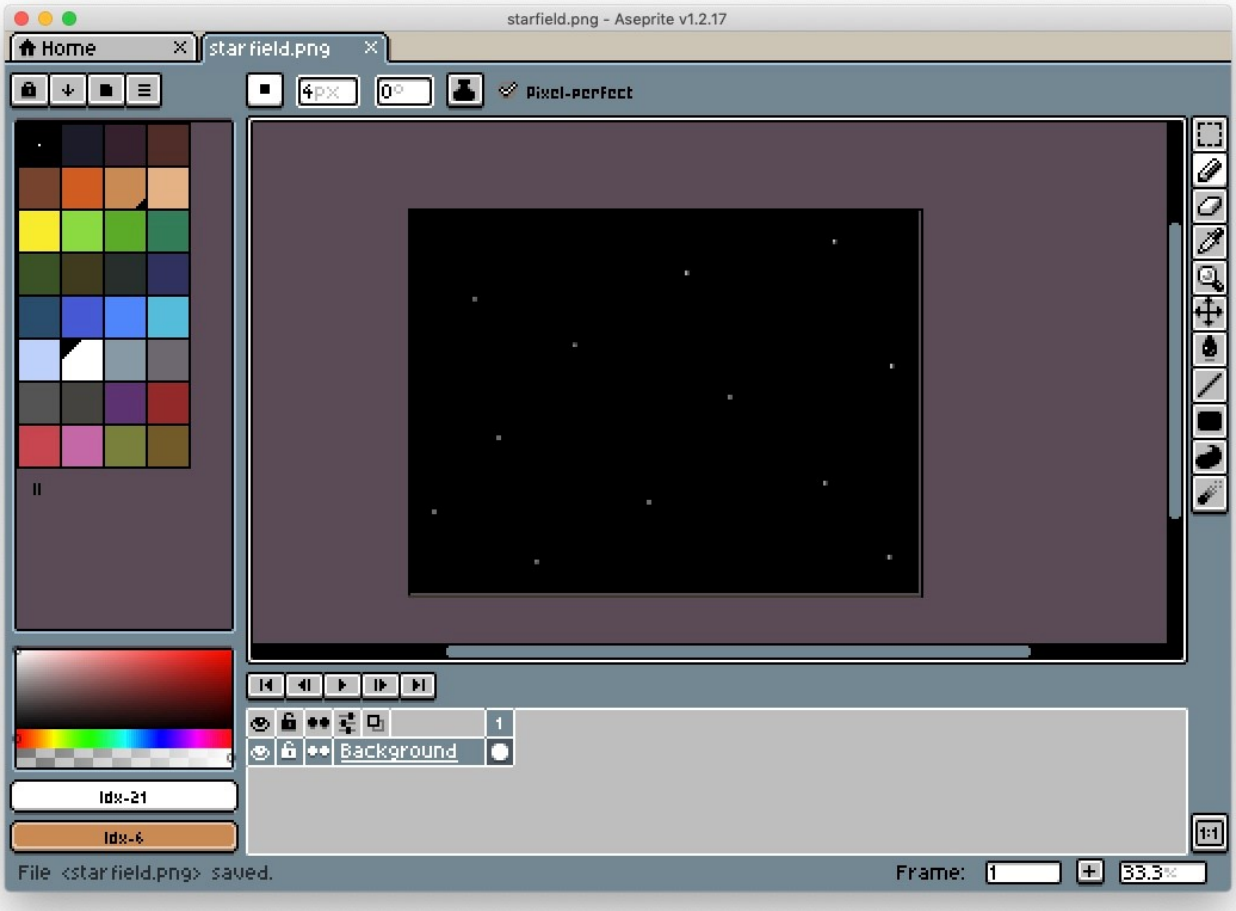
sprites. Since our game resolution is cleanly divisible by factors of 2, I chose to make the rocket sprite 16x8 pixels (rocket.png), approximating the “flattened” look of the original.



The spaceships are taller and wider than the rocket, so I chose a resolution of 64x32. I decided to make my ships (spaceship.png) slightly more symmetrical, but the focus was on passable rather than perfect. You can always update the graphics later if you end up not liking them.



Lastly, I made a 640x480 black background (starfield.png) with a few 4-pixel stars scattered about. We'll use this image for the scrolling background.



The only asset we're missing is the explosion, but we'll tackle that one later, since it requires animation.

Once you've finished creating the necessary graphics, drag them into the project folder's "assets" folder so we can access them in our game.

Preloading Assets

Before we can add our assets to the playfield, we need to preload them. If you remember the Scene methods breakdown from above, `preload()` occurs before `create()`, and allows us to—you guessed it—*preload* our graphics before we *create* them in our scene.

Go back to Play.js and add the follwing method *above* the create method:

```
1 preload() {
2   // load images/tile sprites
3   this.load.image('rocket', './assets/rocket.png');
4   this.load.image('spaceship', './assets/spaceship.png');
5   this.load.image('starfield', './assets/starfield.png');
6 }
```

The `load.image()` method expects two parameters: a string with the key name of the graphic you're

going to use (so you can reference it later in your program) and the URL for where your graphic is located.

Return to the browser and reload to make sure you have no errors. If you do have errors, double-check your spelling and file location paths. 99% of the time, preload errors result from mistyping a file name or the file path.

Implementing the Scrolling Starfield

Next we're going to place the starfield graphic and make it scroll. Fortunately, Phaser has an incredibly handy built-in object called a tile sprite. Tile sprites help simulate scrolling backgrounds by scrolling a sprite's *texture* without moving the position of the sprite itself.

In Play.js's `create()` method, add the following code *above* the code that places the rectangular borders:

```
1 // place tile sprite
2 this.starfield = this.add.tileSprite(0, 0, 640, 480, 'starfield').setOrigin(0, 0);
```

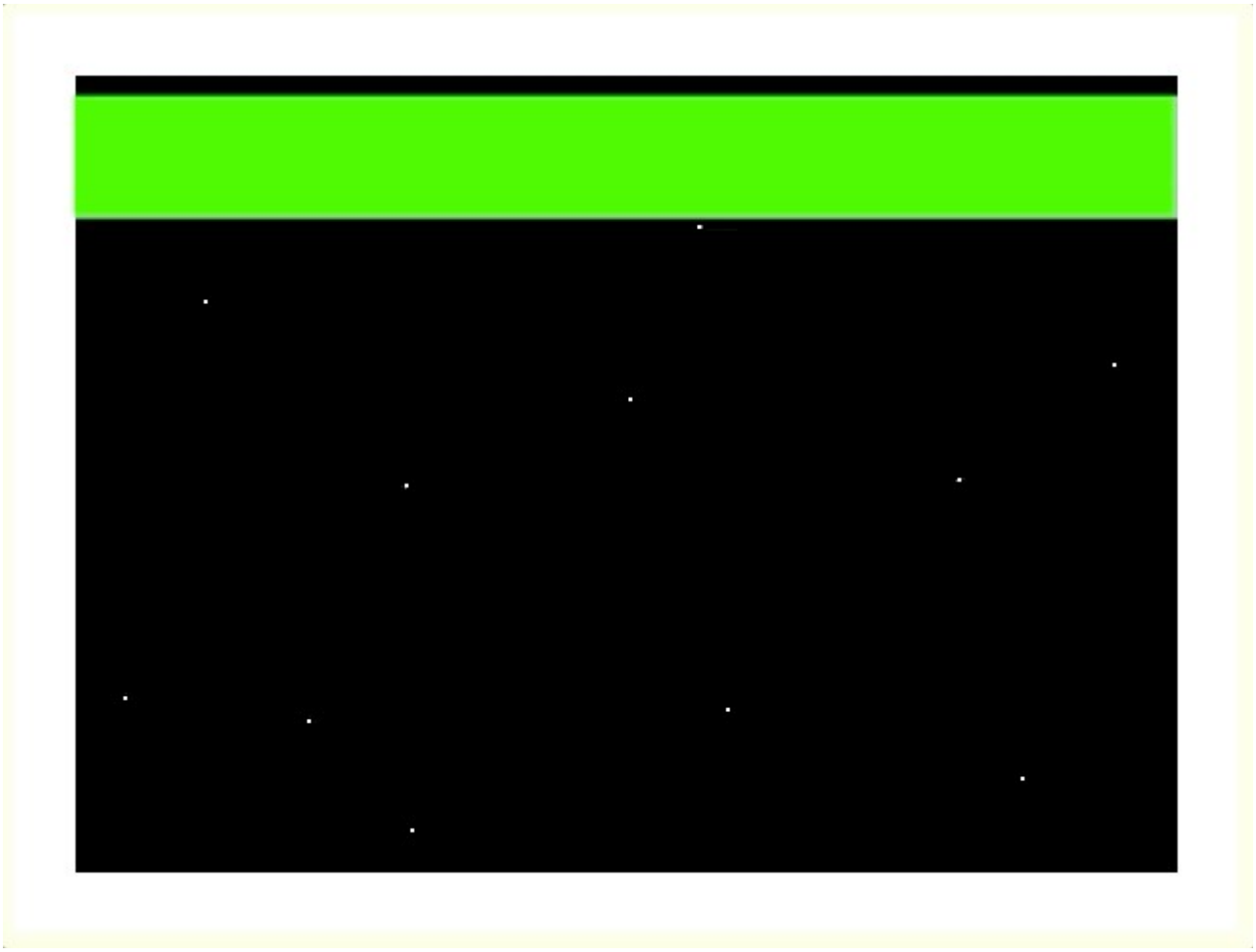
The `add.tileSprite()` method expects five parameters: x-position, y-position, width, height, and a key string that tells us which image to use. We're placing the sprite in the upper-left corner of the game window, but to do so, we need to set the sprite's origin to the upper-left, just as we did before. You might also notice that we added `this` to the front of our variable name. That's because we need the variable to be in scope *anywhere* in the scene, not just in the method in which it was defined. If we'd written `let starfield = this.add.tileSprite(…)`, the console would've thrown an error as soon as we tried to reference that variable outside of `create()` (which, coincidentally, we are just about to do).

Also, the reason we place this code before the white rectangles is because Phaser renders objects in the order in which they are called, from lowest to highest. So placing the tile sprite first means that other objects will render "on top" of the background, rather than behind it.

If you test your game now, you should see the tile sprite in the background, but it won't be moving. To do that, we need to *update* the tile sprite's texture position each frame, which means we need to add the `update()` method to our scene. Let's do that—add this code *below* your `create()` method (and note the use of `this` again):

```
1   update() {
2     this.starfield.tilePositionX -= 4;
3   }
```

Phaser's `update()` method (ideally) runs every frame, which means the texture of our tile sprite will move 4 horizontal pixels left every frame. You can pick any number you like, but 4 pixels feels fairly close to the scrolling speed of the original *Rocket Patrol*. (And yes, I realize I broke my hard-coded rule here, but we're only using this value in one place, so shut up, you know-it-all.)



The Rocket Class

The rocket is our “player,” meaning it’s the sprite we need to be able to control—i.e., move left and right and fire upward. This is more sophisticated behavior than any other object on the screen, so we want to architect our code to accommodate these additional features. Although it’s a bit of overkill for this game, we’re going to define our rocket as its own class. That way, if we decide to add some additional features later on (like two-player simultaneous play), we’ll have a good foundation to do so.

Creating the Class

Just like our other scene classes (`Menu.js` and `Play.js`), our rocket class will need its own file. Create a new folder in your “src” directory called “prefabs” and then, inside that folder, create a new file called `Rocket.js`. Inside `Rocket.js`, write the following code:

```
1 // Rocket prefab
2 class Rocket extends Phaser.GameObjects.Sprite {
3   constructor(scene, x, y, texture, frame) {
4     super(scene, x, y, texture, frame);
5
6     // add object to existing scene
7     scene.add.existing(this);
8   }
```

```
9 }
```

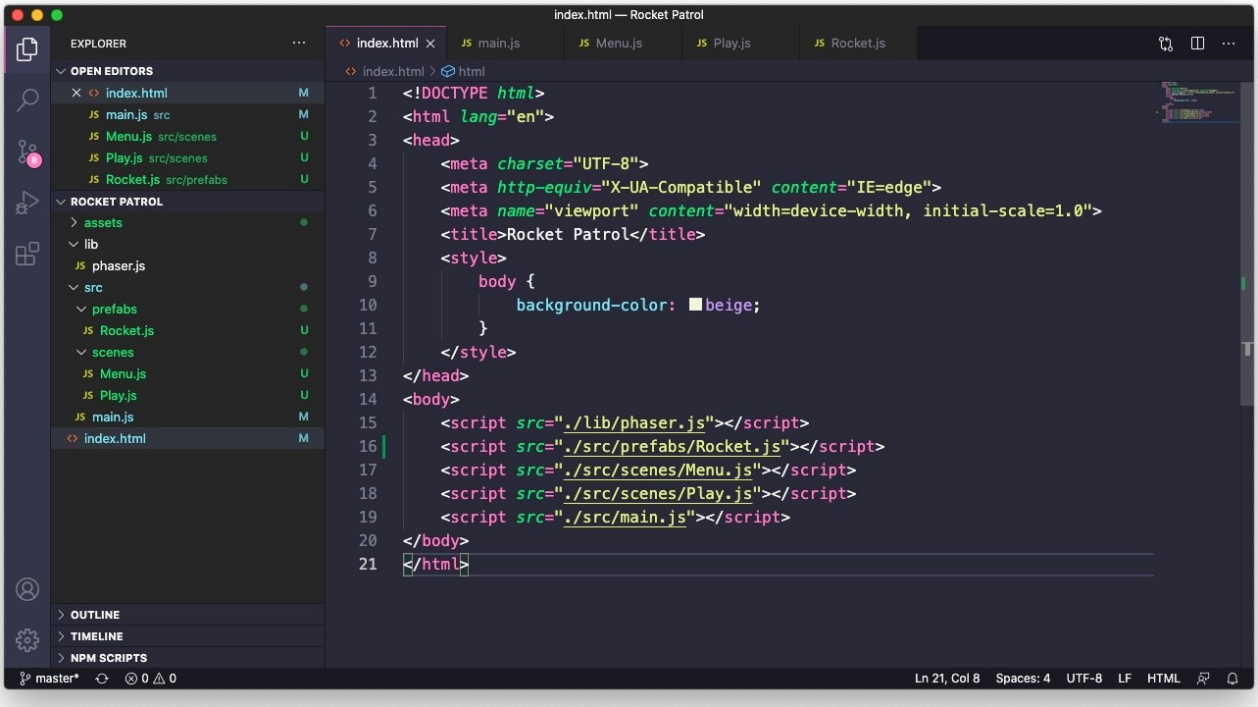
One key difference you'll see in this class versus our prior classes is the `scene.add.existing(this);` command. Since we're extending Phaser's base sprite class, it can't add our sprite to the scene automatically. Instead, we have to manually add `this`—whose current context is the Rocket object—to the scene that we pass in as a parameter. But I'm getting ahead of myself. Let's go ahead and create an *instance* of our Rocket class.

Go back to Play.js and add this line at the *end* of the `create()` method:

```
1 // add rocket (p1)
2 this.p1Rocket = new Rocket(this, game.config.width/2, game.config.height - borderUISize -
  borderPadding, 'rocket').setOrigin(0.5, 0);
```

Note once again that we're able to place our rocket dynamically based on dimensions we created previously. 💪

Lastly, don't forget to go back to `index.html` and add our new script file, just above the Menu.js script. Your full "stack" of scripts should now look like this:



When we reload our game, the rocket sprite should appear in the center of screen, resting just above the bottom rectangular border. Note that I've also added a `setOrigin` command to move the origin to the top *center* of the sprite. This isn't strictly necessary, but it will make collision-checking calculations a bit cleaner later on.

Keyboard Setup

In preparation for rocket movement, we need to set up some keyboard events so we can capture the player's input. And since we're going to use the keyboard throughout *all* of our game scenes, we want to create keyboard variables that are accessible in the global scope.

First, let's go back to `main.js` and add the following code below our main game object:

```
1 // reserve keyboard vars
2 let keyF, keyR, keyLEFT, keyRIGHT;
```

As the comment says, we're simply reserving those names for our keyboard variables. We won't give them values until we're in the scene.

Let's do that now. Head back to Play.js and add the following code at the bottom of the `create()` method:

```
1   // define keys
2   keyF = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.F);
3   keyR = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.R);
4   keyLEFT = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.LEFT);
5   keyRIGHT = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.RIGHT);
```

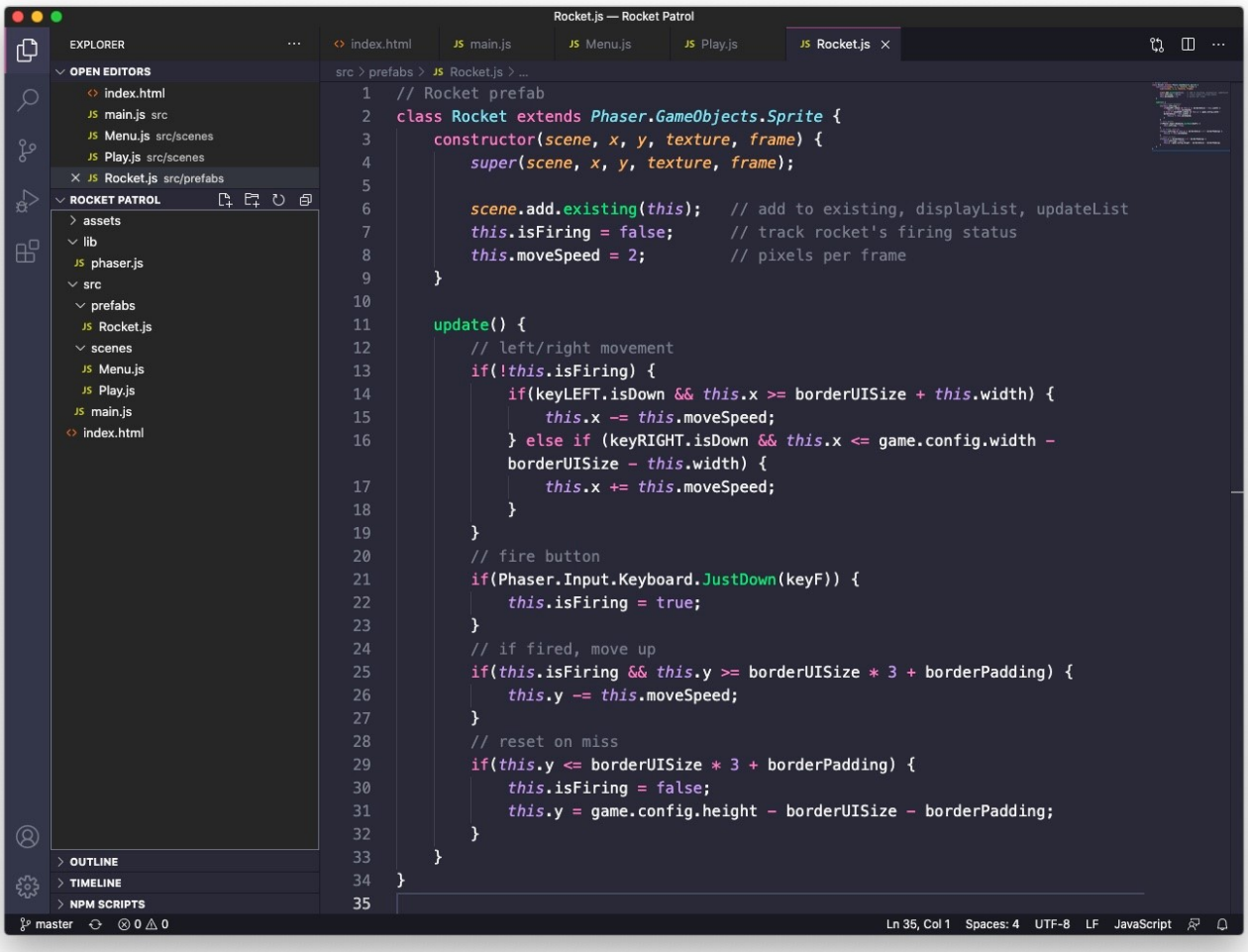
Adding keyboard keys is a pretty wordy in Phaser (handling keyboard input across web browsers is really messy, so it's not the Phaser devs' fault), but what we're essentially doing is attaching four keyboard keys—F, R, LEFT arrow, and RIGHT arrow—to the variables we reserved. With that done, we can "wire them up" to player movement.

Adding Movement

One advantage of using a separate class to handle our player rocket is that we can keep class-specific code within the class itself. As we've already seen, Phaser runs a core `update()` loop that allows all game movements, animations, physics, and so on to happen. For player input to work, we'll need to have our sprite update as well. And one clean way to do this is to connect our sprite to the core update loop.

In fact, we've done some preliminary work on this already. When we wrote the Rocket class, we had the following line in the constructor: `scene.add.existing(this);` In addition to adding our custom sprite object to the scene, it also adds the sprite to the Phaser `displayList` (which makes sure it can be seen) and the `updateList` (which allows our sprite to hook in to the Phaser update). (For more on how objects are added to Phaser, see the developer's post on Phaser Factories.)

To take advantage of the core update loop, we need to do two things. First, return to Rocket.js and add the new code shown below:



Most of our changes are in the new `update()` method, but be sure not to miss the `this.isFiring = false;` and `this.moveSpeed = 2;` statements shown on lines 7 + 8 above.

This is a lot of new code—so what exactly have we done? Well, if we look carefully at how the original *Rocket Patrol* works, we see that firing the rocket commits the player to that shot. In other words, once you shoot, you have to let the rocket ascend until it either (a) hits a spaceship or (b) reaches the top of the screen. That means the player can move the rocket left and right *only* when it's "on the ground." Once the shot is taken, horiztonal movement is disabled. Thus, we need a "toggle" to keep track of whether the rocket is firing or not. That's the `isFiring` variable.

As for the code itself, the first if statement checks to see if the rocket is firing. If **not**, left and right movement is allowed, once again by using an if/else statement. The variables you see in the if statements (once again reusing our dynamic UI sizes!) are doing simple bounds checking so the player can't move the rocket beyond the UI borders. Movement is handled by checking if keyLEFT or keyRIGHT `isDown`, meaning the player is holding down the key. If so, the rocket's x position is incremented or decremented by our defined `moveSpeed`, ie 2 pixels per frame. Fairly slow, but that's true to the original game.

The second if statement uses a slightly different `Phaser.Input` command to see if the F key is `JustDown`, rather than `isDown`. We do so because we only want the rocket to fire once. In contrast to

`isDown`, `JustDown` doesn't continue to register input every frame. Instead, it waits for the player to release the key and press it again before a new input is registered.

The third if statement moves the rocket up—ie, fires it—if `isFiring` is true (meaning the F key was pressed) and the rocket is below the green bar.

The final if statement resets our rocket by moving it back to the "ground" and toggling `isFiring` back to false.

All of this new code is great, but it won't do anything until we add one more thing. Return to Play.js and add the following line to your `update()` method:

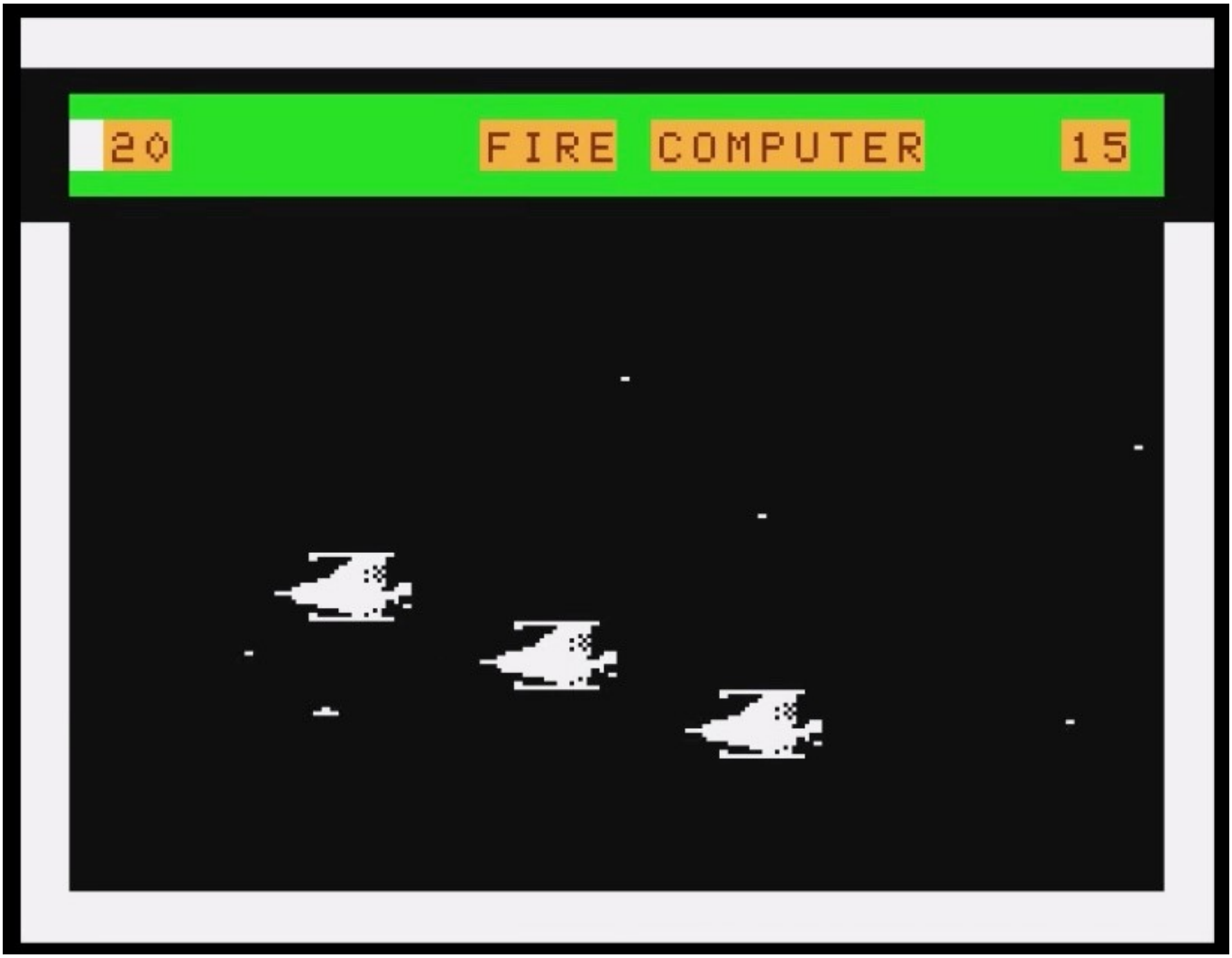
```
1 this.p1Rocket.update();
```

Doing so tells Phaser to run our custom class' `update()` method when it runs its own `update()` method.

We're done with our rocket for now—let's move on to the spaceships so we have something to shoot 🚀. (Psst…it might also be a good time to commit your changes and push them to GitHub. But I'm not your dad—I can't tell you how to live your life.)

The Spaceship Class

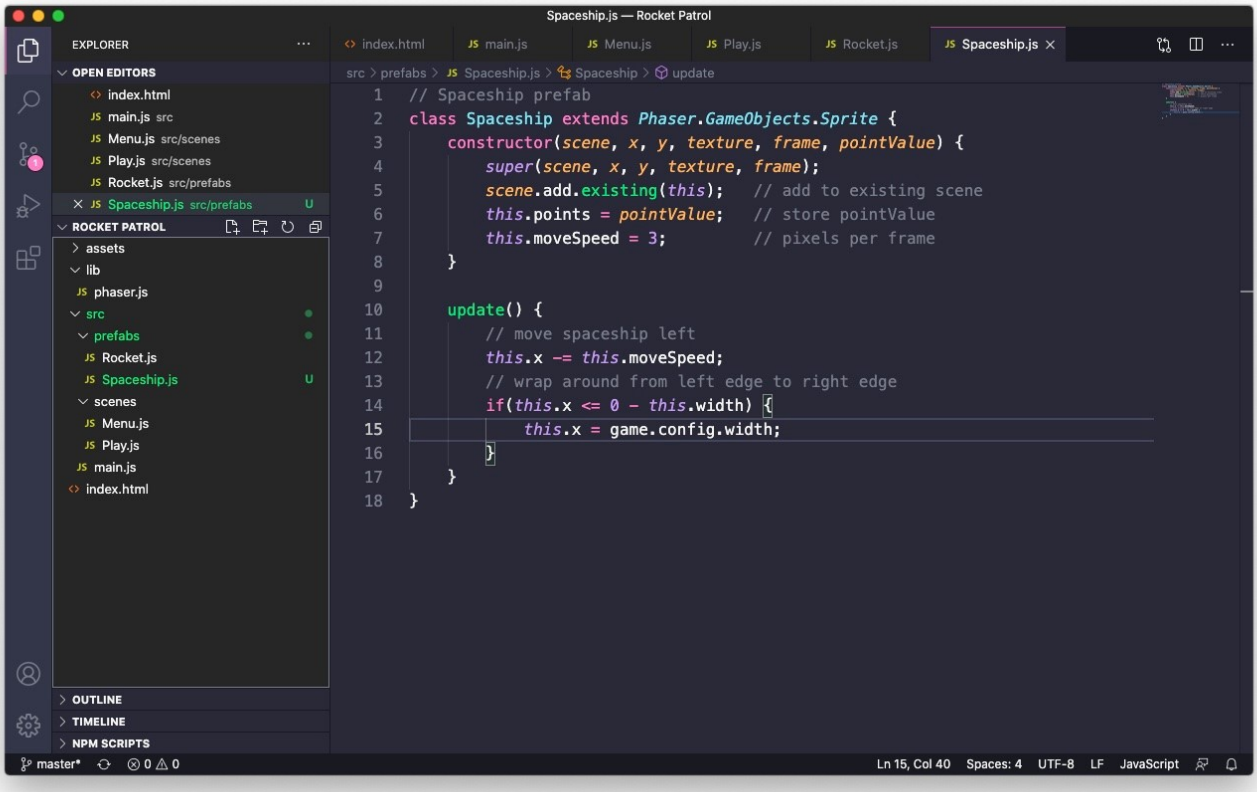
Before we add the spaceships, we need to go back to the original *Rocket Patrol* and observe how they work. If you watch carefully, there are only ever three spaceships on the screen at a time, and each spaceship stays in its designated "lane." The spaceships move right to left, and when they reach the left edge of the screen, they "wrap" back around to the right side. However, if the rocket collides with a spaceship, an explosion animation plays, and a new spaceship immediately spawns from the right edge, in the same lane.



This is pretty odd behavior, but remember that the programmer was working within the constraints of a platform from the 1970s. The APF MP-1000 had some severe limitations on the number of sprites it could display at once, so the “wraparound recycling” of objects you see was likely a function of those limitations. Constraints can make you creative!

We’re going to replicate this behavior in Phaser, but with some variations to make the game slightly more interesting.

Like the Rocket, we’re going to start by creating a new file in the “prefabs” folder called `Spaceship.js` and typing out our basic class structure in that new file. And you’ll notice that the code we use here is similar to what we typed in `Rocket.js`:



The first difference is an additional parameter passed to the constructor called `pointValue`. We're going to improve the original *Rocket Patrol* a bit by giving each spaceship a different point value, depending on its position. The higher the spaceship is, the more points it will be worth. The top ship will reward 30 points, the middle 20, and the lowest 10.

The second difference is the `update()` method, which is much simpler than Rocket.js. Since we need no player input, for now, we've written code to move the spaceship right to left and wraparound if it reaches the leftmost edge. To get the wraparound function to work, though, we'll need to change the spaceship's origin point when we create it. Let's do that now.

Return to `Play.js` and add the follow code below our `p1Rocket` definition in `create()`:

```
1   // add spaceships (x3)
2   this.ship01 = new Spaceship(this, game.config.width + borderUISize*6, borderUISize*4,
  'spaceship', 0, 30).setOrigin(0, 0);
3   this.ship02 = new Spaceship(this, game.config.width + borderUISize*3, borderUISize*5 +
  borderPadding*2, 'spaceship', 0, 20).setOrigin(0,0);
4   this.ship03 = new Spaceship(this, game.config.width, borderUISize*6 + borderPadding*4,
  'spaceship', 0, 10).setOrigin(0,0);
```

Take a moment to make sure you understand what this code does. First, we're binding three new Spaceship class instances as properties to our current scene context (eg, `this.ship01`), then passing *six* parameters to the Spaceship class constructor: the current scene (`this`), x-position, y-position, the key name for our graphics assets (as defined in `preload()`), the frame number (we only have one frame in our sprite, so we pass `0`, meaning the first frame), and then our custom parameter, `pointValue`. At the end, we chain the `setOrigin()` method to make sure the origin is on the upper left

of the sprite, so our screen-wrapping code from `Spaceship.js` will work.

Also note two things about the x- and y-coordinates. Phaser allows us to place things “off stage,” or beyond what the player can see. In other words, objects can exist outside of the game’s visual bounds. This is really handy for moving objects in and out of view without having to constantly create and destroy them. In the code above, we’re placing the lowest spaceship just outside of the game window’s right edge, then staggering the placement of the next two spaceships so they don’t all appear stacked atop one another. And once again, we’re doing all this spacing by using our dyanmic border UI variables from the beginning of this tutorial!

Next, scroll down to `update()` and add code that makes sure our spaceships actually move:

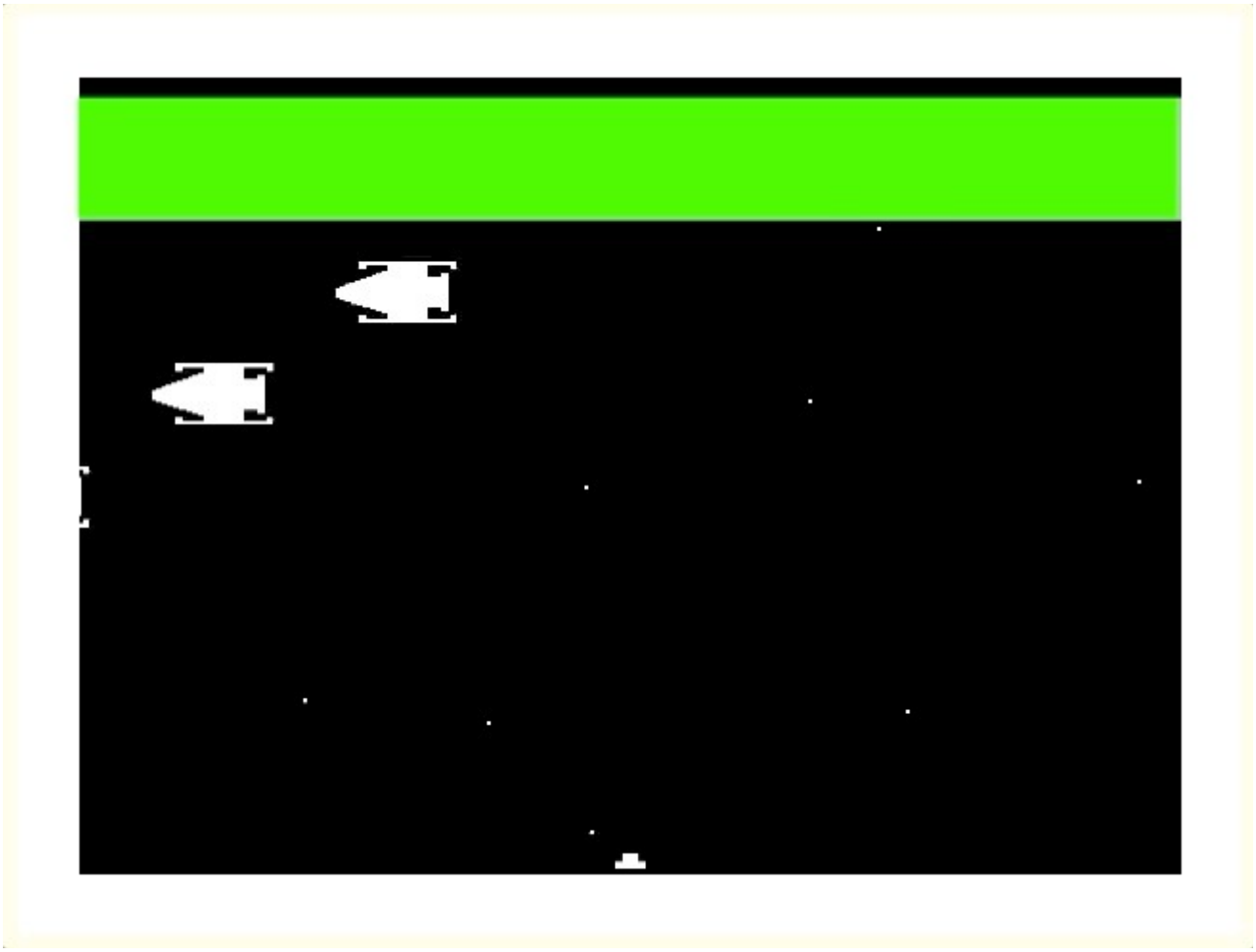
```
this.ship01.update();               // update spaceships (x3)
this.ship02.update();
this.ship03.update();
```

Those statements should look familiar, because we used the same code to update `p1Rocket`.

Finally, head back to `index.html` and reference our new class. In between the Rocket.js and Menu.js `<script>` tags, add the following:

```
<script src="./prefabs/Spaceship.js"></script>
```

Now we should be ready to test things out.



Looks good to me 😗

Shooting Spaceships

Now that our fundamental elements are working, we need to add some interaction. Without checking collisions, games can't do much. We need to know (a) when the rocket hits a spaceship (b) which spaceship it hits and (c) what to do when it hits. Phaser has lots of cool built-in mechanisms for handling collisions, but they require hooking into the physics system, and that's more than we need for this project.

For our purposes, we only need a style of collision checking called Axis-Aligned Bounding Boxes (AABB). Since our sprites are (a) rectangular and (b) don't rotate, we can pretend that our sprites have invisible rectangles around them, defined by the dimensions of the sprite, and check collisions by determining whether those rectangles overlap.

Collision Checking

Let's start with the code to check collisions. In `Play.js`, underneath our `update()` method, let's build a new custom function:

```
1 checkCollision(rocket, ship) {
2     // simple AABB checking
```

```
    if (rocket.x < ship.x + ship.width &&
        rocket.x + rocket.width > ship.x &&
        rocket.y < ship.y + ship.height &&
        rocket.height + rocket.y > ship. y) {
            return true;
    } else {
        return false;
    }
}
```

That may look complicated, but it's really one big if statement checking four different conditions to see if two rectangles overlap. If they do, the function returns `true`. If not, the function returns `false`. Also note that I've used the parameter names rocket and ship here, but that's just to make the function more readable. This function would work perfectly well with any two rectangles.

For collisions to work properly, we need to check our objects *every single frame*. Let's do that by adding the following code at the bottom of our `update()` method:

```
// check collisions
if(this.checkCollision(this.p1Rocket, this.ship03)) {
  console.log('kaboom ship 03');
}
if (this.checkCollision(this.p1Rocket, this.ship02)) {
  console.log('kaboom ship 02');
}
if (this.checkCollision(this.p1Rocket, this.ship01)) {
  console.log('kaboom ship 01');
}
```

Each frame, we're checking the rocket against each successive spaceship, and if the collision is true, we're logging to the console. The `console.log()` is just a placeholder action to check our work. We'll do more collision handling in the next section. For now, go back to the browser, shoot the spaceships, and see if your collisions register in the console.

Keep in mind that this won't be *perfect* collision detection. Because we're checking rectangular bounding boxes against one another, you can have cases where a collision registers, but it doesn't look like it should have happened. For example, the rocket can hit the left edge of a spaceship and make it disappear when it's clearly a "miss." But that's because our graphic asset has a triangular nose that doesn't fill the entire bounding box. But that's OK—close enough works for now.

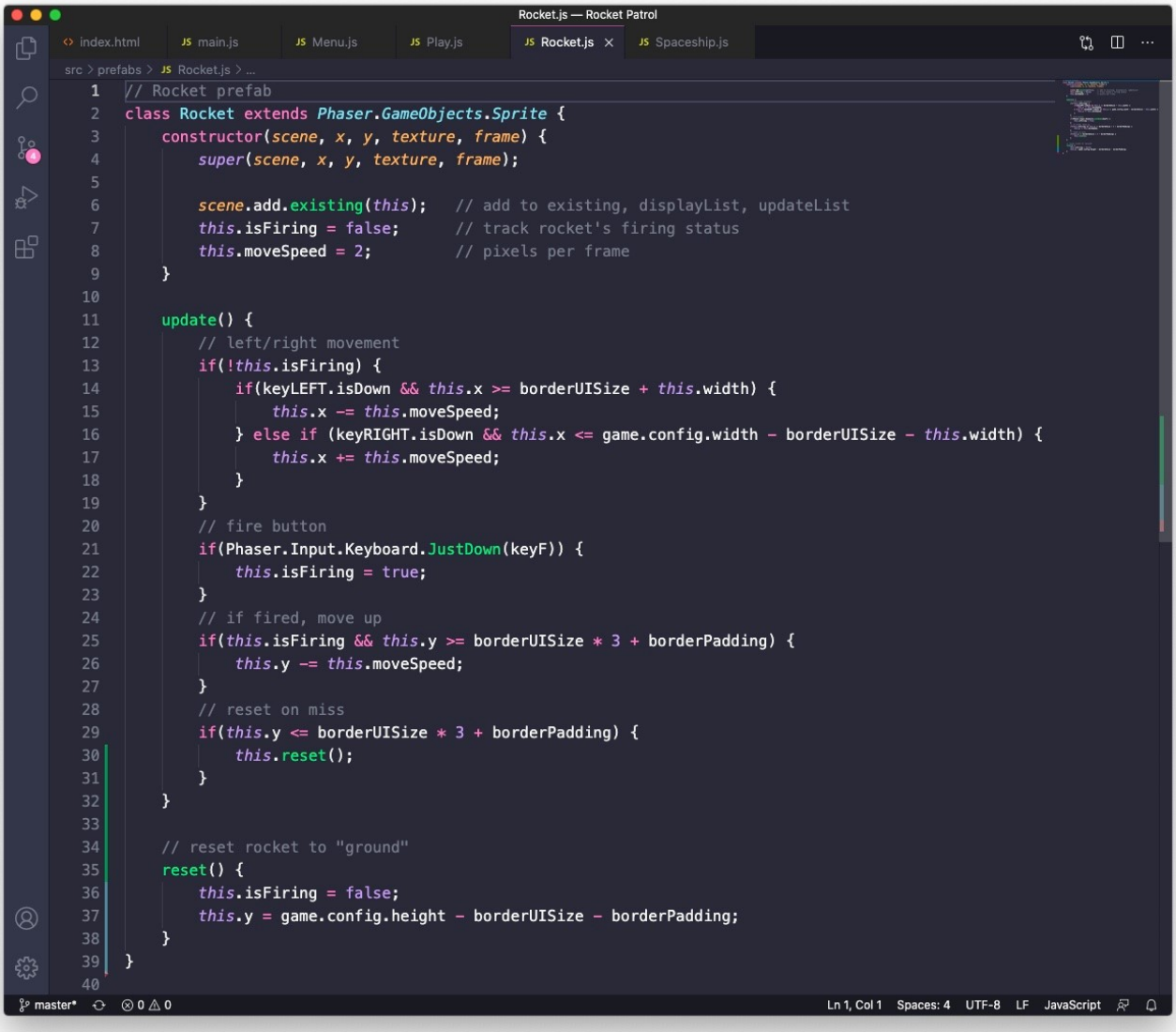
Collision Handling

Our first order of business is to handle the rocket-to-ship collisions. In the original game, several things happen:

- The spaceship sprite is removed
- An explosion animation plays
- The spaceship is reset to the right edge of the screen
- The rocket is reset to the bottom of the screen

Let's do the object resets first then circle back for the animation. Head back to `Rocket.js` and create a

new function called `reset()` below the `update()` method. If you'll remember back when we were programming the rocket behavior, we already created reset code for when a shot missed. Let's grab that code, put it in our new function, then replace what we deleted with a function call. Your finished code should look like the following:

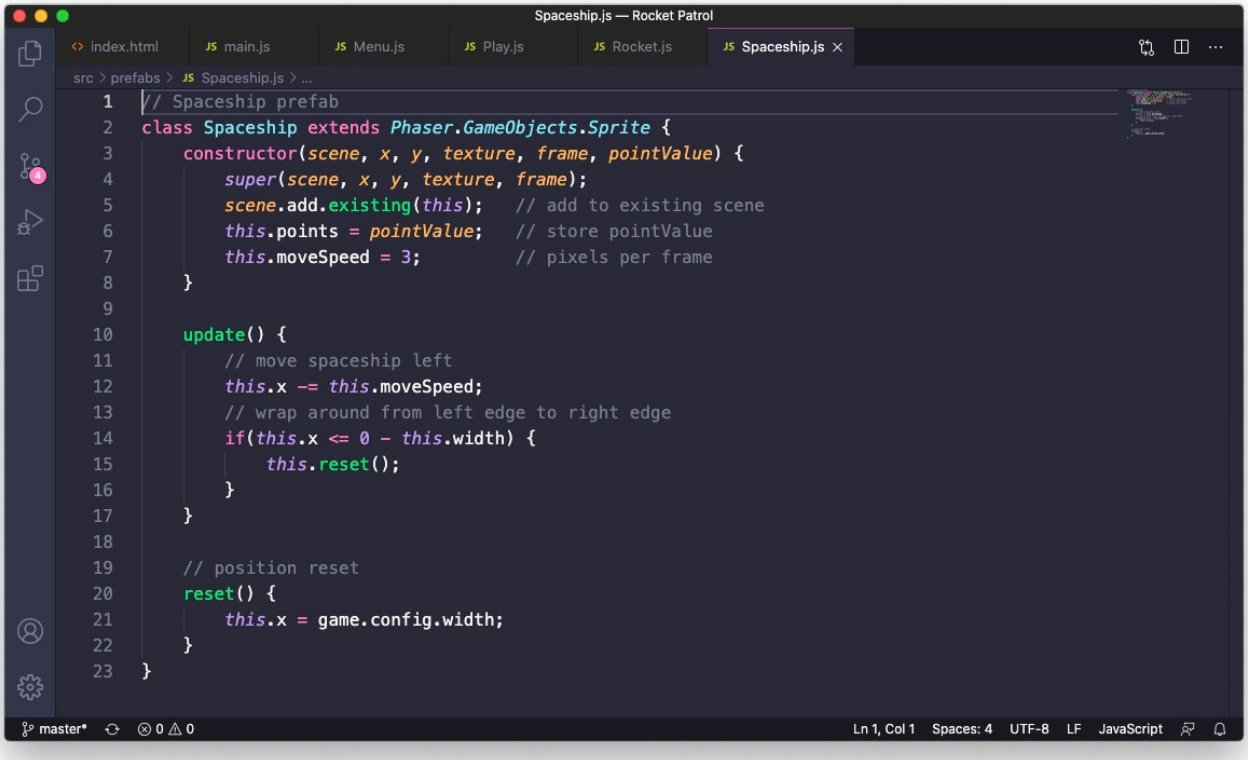


Now, go back to Play.js and replace each of the `console.log` statements from our collision checks with the object method we just created:

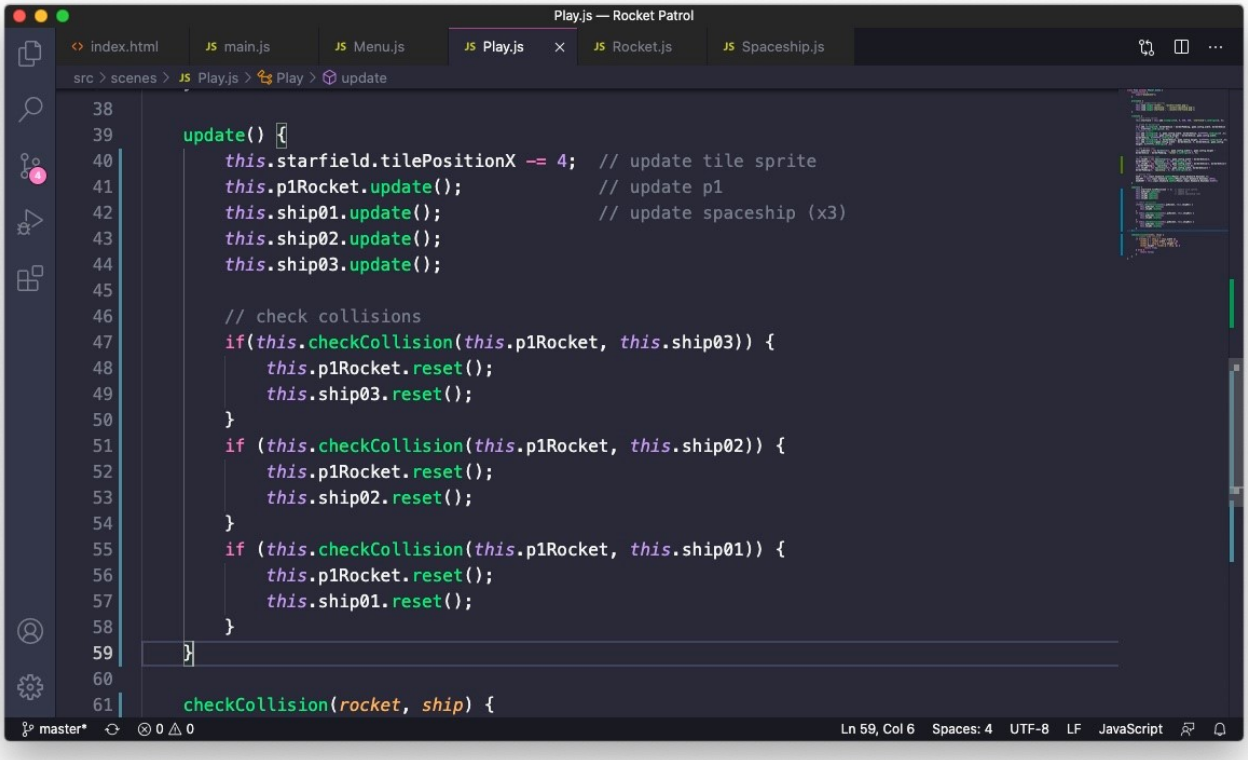
```
this.p1Rocket.reset();
```

Test the game again. Your rocket should reset to the bottom if THY AIM BE TRUE and it collides with a spaceship.

Return to `Spaceship.js`. Hopefully you guessed that we're going to do almost the exact same thing again. Make sure your code looks like the following:



And buddy you better believe we're going to add resets for each of our ships in `update()`:

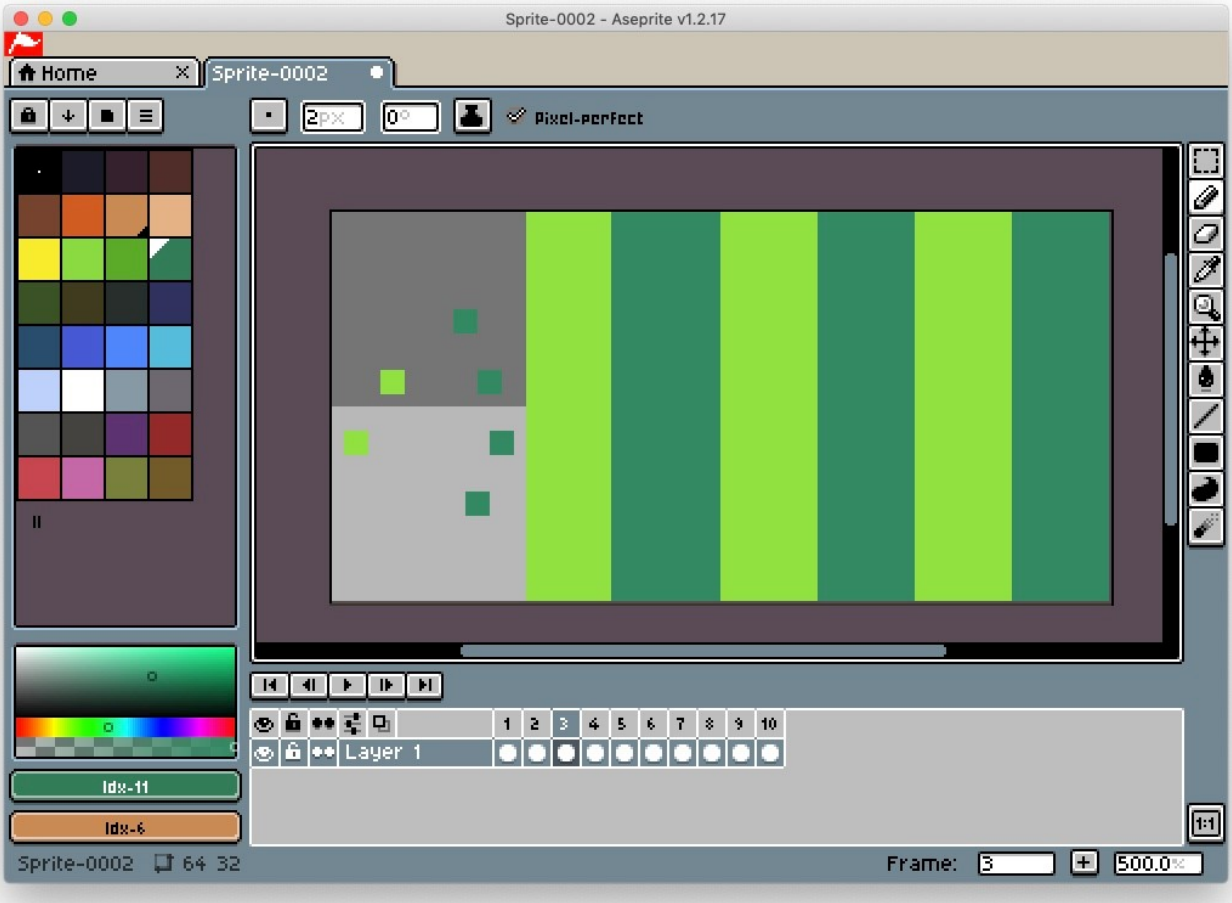


If all went well, you should see shots connecting and reseting sprites properly. If not, check back over your code and make sure you typed everything correctly. Sometimes a simple typo or missed line will make everything go haywire. And that's the "beauty" of programming. 🙃

Creating 💥Explosions💥 in the Sky

It's time to wreak mayhem. But first, we need an explosion animation. Let's make one.

I'm using Aseprite again because it allows me to create animation frames and export them as a single .png spritesheet, ready for import into Phaser.



Once again, I didn't do anything fancy. I tried to replicate the "disintegrating green forcefield" explosion from the original game, but I cleaned up the bars and gave it a little trailing particle animation. The final sprite sheet, titled explosion.png, looks like this:

With that done, drag the finished file into your "assets" folder.

As with the other graphics assets, we first need to load the spritesheet in the Play scene's `preload()` method. But using a spritesheet requires a slightly different syntax than we used before. Type the following below your existing image load statements:

```
1 // load spritesheet
2 this.load.spritesheet('explosion', './assets/explosion.png', {frameWidth: 64, frameHeight: 32,
  startFrame: 0, endFrame: 9});
```

First, instead of `this.load.image()`, we're using `this.load.spritesheet()`. Then, we provide three parameters: a key string to identify the spritesheet, the URL of its location, and a frame configuration object (note the curly braces). At a minimum, the frame configuration object needs the frame width, but we're providing additional data to be as descriptive as possible. Phaser assumes that spritesheets contain frames of equal sizes (sheets with variable sizes are called *texture atlases*, and they require a different setup), but the dimensions do not need to be symmetrical. In our case, the animation is 10 frames long (numbered 0–9), and each frame is 64x32 pixels (matching the dimensions of our spaceship, natch).

(For more info on Phaser's spritesheet loader, see the Phaser 3 documentation.)

Creating an Animation

Next we need to use Phaser's animation system to bind an animation to the current scene. At the bottom of the `create()` method, type the following code:

```
// animation config
this.anims.create({
    key: 'explode',
    frames: this.anims.generateFrameNumbers('explosion', { start: 0, end: 9, first: 0}),
    frameRate: 30
});
```

The `anims` property is used to access Phaser 3's Animation Manager. As Phaser developer Richard Davey explains "In Phaser 3 the Animation Manager is a global system. Animations created within it are globally available to all Game Objects. They share the base animation data while managing their own timelines. This allows you to define a single animation once and apply it to as many Game Objects as you require." So this code is using the global animation manager to *create* a new animation and bind it to the scene (via `this`).

As usual, there are many properties we can pass along in the configuration object, but here we're using three: a key string, the frames in our animation, and the frame rate. The frames property expects an array, but Phaser provides a handy method to `generateFrameNumbers` automatically, provided we pass along the spritesheet we're using, as well as the start, end, and first frame in the animation. This is incredibly useful is you have a spritesheet that contains *all* of the animations for, say, a player character that you don't want to chop up into individual animations. Phaser will do the chopping for you. 🔪

Now that the animation is setup, we need to call it. But unlike the spaceship or rocket, we don't add the actual explosion animation sprite in `create()`. Instead, we need the explosion to appear when a rocket/ship collision happens. To handle that event, we're going to construct a new function in our scene, called `shipExplode()`, directly below our `checkCollision()` function:

```
shipExplode(ship) {
  // temporarily hide ship
  ship.alpha = 0;
  // create explosion sprite at ship's position
  let boom = this.add.sprite(ship.x, ship.y, 'explosion').setOrigin(0, 0);
  boom.anims.play('explode');             // play explode animation
```

```
  boom.on('animationcomplete', () => {    // callback after anim completes
    ship.reset();                         // reset ship position
    ship.alpha = 1;                       // make ship visible again
    boom.destroy();                       // remove explosion sprite
  });
}
```

Next, we're going to change our collision checking code in `update()` to look like this:

```
// check collisions
if(this.checkCollision(this.p1Rocket, this.ship03)) {
  this.p1Rocket.reset();
  this.shipExplode(this.ship03);
}
if (this.checkCollision(this.p1Rocket, this.ship02)) {
  this.p1Rocket.reset();
  this.shipExplode(this.ship02);
}
if (this.checkCollision(this.p1Rocket, this.ship01)) {
  this.p1Rocket.reset();
  this.shipExplode(this.ship01);
}
```

That's a lot of new info, so let's break it down step by step.

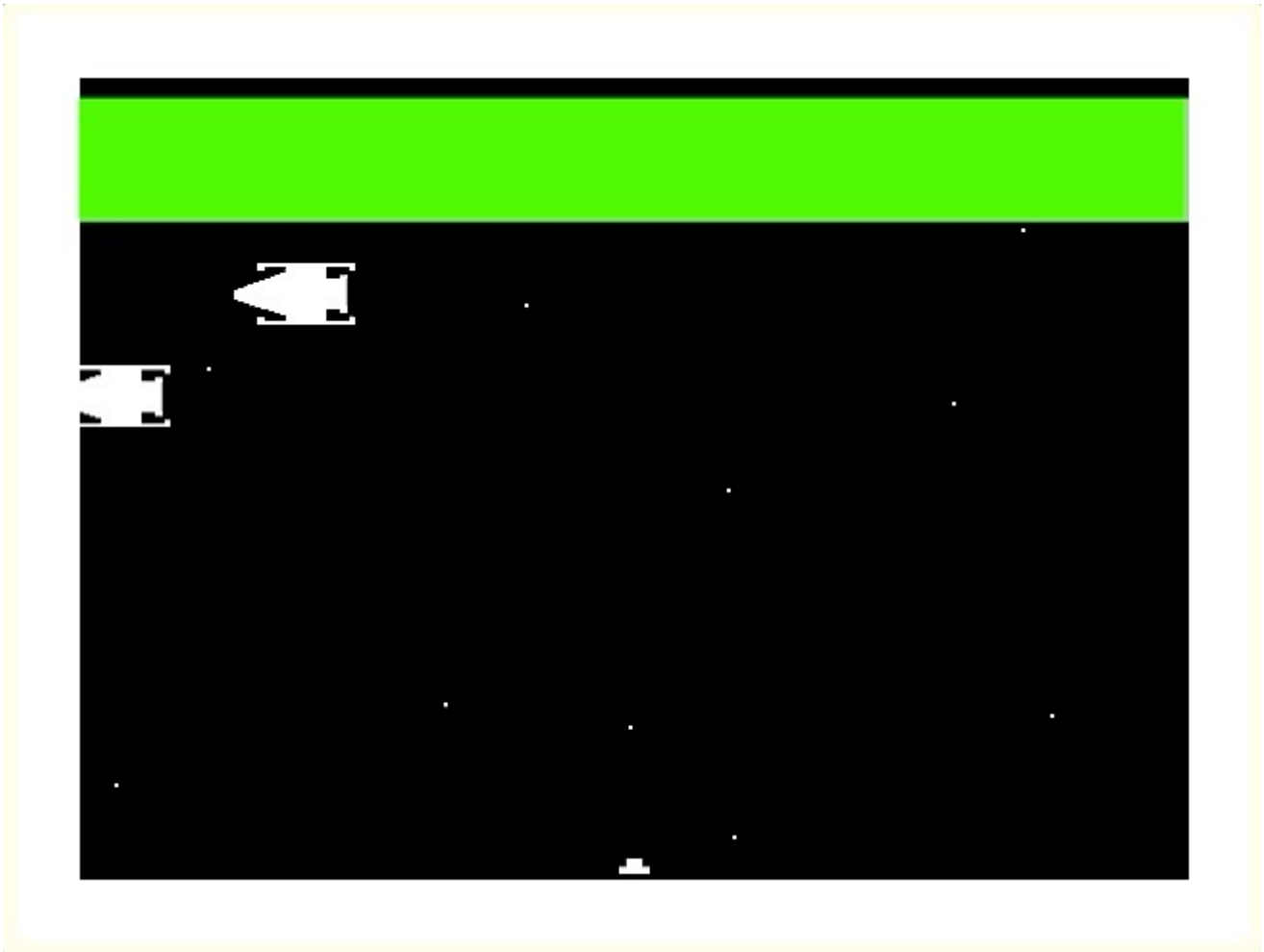
If you think through what needs to happen during a rocket/spaceship collision, there's a specific series of steps:

- Reset the rocket so it's ready to fire again
- Temporarily hide the spaceship that was hit
- Create the animation sprite at the hit spaceship's position
- When the animation finishes…
    - reset the spaceship position,
    - unhide the spaceship sprite,
    - and get rid of the animation sprite

And that's exactly what we've done in the code above. When a collision occurs, we first reset the rocket, then we call the `this.shipExplode()` function, passing along the collided spaceship as a parameter. Once inside the function, we use `ship.alpha = 0` to make the spaceship sprite transparent, ie, invisible. Phaser provides every sprite object with an `alpha` setting, so we can make any sprite more or less transparent at will. Then, we create a temporary `boom` variable to attach a new sprite to the scene, using the 'explosion' key. Then we play the 'explode' animation we created above via that sprite. The `boom.on()` function is an animation event that fires when a certain condition is met—in this case, it's when the animation completes. We then provide a callback function to trigger when the event happens. (The function might look a little odd, but it's simply a JavaScript arrow function, a shorthand way to call an anonymous function that helps make our code cleaner and more concise.) Finally, inside that callback function, we reset the spaceship position, make it opaque, then destroy the boom sprite.

Whew 😥

With that done, you can head back to the browser and test the game. If you got the code correct, collisions should now trigger explosion animations. WATCH THOSE SPACESHIPS VAPORIZE.



Keeping Score

With the basic elements of the game loop now in place, it's time to track the player's score. Doing so means we'll need a variable to store the score, a mechanism to update the score, and a way to display the score on the screen.

Return to `Play.js` and bind a score property to the scene at the end of the `create()` method:

```
1 // initialize score
2 this.p1Score = 0;
```

Note that we're also initializing the score to `0`. And we're doing this in the `create()` method, because we want the score initialization to happen *once*, when the scene is created, but before the update loop begins. (If we placed this statement in `update()`, the player's score would be set to `0` on every update frame, which would never allow the player to receive any points!)

Next, we're going to display the score so we're sure that it's updating properly. In the original game,

the player's score is displayed on the left-hand side of the green bar with an orange-ish background in brown (or red) type. We'll try to do the same.

Directly below the score variable definition, add the following code:

```
1   // display score
2   let scoreConfig = {
3     fontFamily: 'Courier',
4     fontSize: '28px',
5     backgroundColor: '#F3B141',
6     color: '#843605',
7     align: 'right',
8     padding: {
9       top: 5,
10      bottom: 5,
11    },
12    fixedWidth: 100
13  }
14  this.scoreLeft = this.add.text(borderUISize + borderPadding, borderUISize + borderPadding*2,
    this.p1Score, scoreConfig);
```

The only bit that's actually adding text to the screen is the last statement. In that line, we're binding a `scoreLeft` property to the scene (so we can update it elsewhere), then passing four parameters to `add.text()`: x-position, y-position, the text to display, and a configuration object. The configuration object is all the stuff *above* the final statement. It's just a big ol' object packed with properties that style our text. Phaser allows you to do a *lot* of text manipulation, so be sure to explore the documentation for a complete description of your options.

If you check back in the browser, you should see a big fat zero in an orange box. That's good, but we need a way to make the score dynamic, and that's going to involve our trusty `update()` loop and the `pointsValue` variable we so presciently squirreled away in our Spaceship class.

The good news is, we already have the logic we need to update our score with very little code overhead. We know that the only time our player's score will increase is when their rocket hits a spaceship, so we might as well add our score-tallying code in the `shipExplode` function. Update that function so it looks like the following (the only new code you'll need to add are lines 12–14 below):

```
1  shipExplode(ship) {
2    // temporarily hide ship
3    ship.alpha = 0;
4    // create explosion sprite at ship's position
5    let boom = this.add.sprite(ship.x, ship.y, 'explosion').setOrigin(0, 0);
6    boom.anims.play('explode');             // play explode animation
7    boom.on('animationcomplete', () => {    // callback after ani completes
8      ship.reset();                         // reset ship position
9      ship.alpha = 1;                       // make ship visible again
10     boom.destroy();                       // remove explosion sprite
11   });
12   // score add and repaint
13   this.p1Score += ship.points;
14   this.scoreLeft.text = this.p1Score;
15 }
```

Line 13 is updating the player's score with the value stored in the exploding spaceship's `points` property. Line 14 uses the `.text` property of our text object to replace its contents with the new value. In some games, you'll see the text replacement in `update()`, but that's overkill for this project. Repainting text objects can be processor intensive, and our score doesn't update frequently enough to repaint the score every frame.

All Games Must Come to an End

The final elements we need to complete our gameplay loop are a timer, an end condition, and some mechanism to reset our game. Let's tackle those in turn.

Game Time

The original *Rocket Patrol* limits the player's time to 60 seconds and makes a design decision to *not* show the timer to the player. Fortunately, Phaser has a robust time system that we can use to create our one-minute clock. In Phaser, timer's are essentially just an event that does something after a specified amount of time. Since our timer only needs to run once before the game ends, we can use a *one-shot* timer, meaning it will run once and only once.

In Play.js, add this bit of code at the bottom of `create()`:

```
1 // 60-second play clock
2 scoreConfig.fixedWidth = 0;
3 this.clock = this.time.delayedCall(60000, () => {
4     this.add.text(game.config.width/2, game.config.height/2, 'GAME OVER',
  scoreConfig).setOrigin(0.5);
5     this.add.text(game.config.width/2, game.config.height/2 + 64, 'Press (R) to Restart',
  scoreConfig).setOrigin(0.5);
6 }, null, this);
```

`time.delayedCall()` is Phaser's method for *calling* a function after a *delay*, ie, a one-shot timer. We're passing four parameters: the time that will elapse (in milliseconds) before the callback function fires, the callback function itself (which is another JS arrow function), any arguments we might want to pass to the callback (in this case, `null`), and the callback context (which is `this`, meaning the Play scene). The callback function itself adds text to the center of the screen that says "GAME OVER" and "Press (R) to Restart." Notice that we slightly altered the `scoreConfig` object to allow us to adopt the same text style as the score, but without the fixed width limitation. Efficient!

Note: As written, this will print the "GAME OVER" text after 60 seconds, but for testing, I'd recommend you change the timer to 1000 (1 second). You don't want to wait for one minute to elapse every time you test the timer!

End Condition

Our timer is in place, and it prints "GAME OVER," but it doesn't enforce the game ending in any way. In the original *Rocket Patrol*, once the timer ends, the player can no longer move, and the score is set. To do the same, we need to set a game over flag that "locks" player input.

Let's go back and revise our play clock with some new functionality (note that you'll only need to add lines 1, 2, and 9 below):

```
1 // GAME OVER flag
2 this.gameOver = false;
3
4 // 60-second play clock
5 scoreConfig.fixedWidth = 0;
6 this.clock = this.time.delayedCall(60000, () => {
7     this.add.text(game.config.width/2, game.config.height/2, 'GAME OVER',
  scoreConfig).setOrigin(0.5);
8     this.add.text(game.config.width/2, game.config.height/2 + 64, 'Press (R) to Restart',
  scoreConfig).setOrigin(0.5);
9     this.gameOver = true;
10 }, null, this);
```

There's only two new additions. First, we bind a new `gameOver` property to the scene and set it to `false`. Then, we set it to `true` in the timer's callback function. In other words, the game over flag will flip to `true` when the timer expires. But toggling the flag still doesn't do anything. Scroll down to your `update()` method and revise your sprite updates to the following:

```
1 if (!this.gameOver) {
2     this.p1Rocket.update();         // update rocket sprite
3     this.ship01.update();           // update spaceships (x3)
4     this.ship02.update();
5     this.ship03.update();
6 }
```

Wrapping the updates in an if statement tells the code to only update the rocket's/spaceships' positions if the game is **not** over. Go back to the browser and test the changes (and again, for your own sanity, reduce the timer to a smaller delay). Once the timer completes, the ships should stop updating.

Replay Value

The core gameplay loop is done, and the game ends when time is up, but we need to give the player more than one opportunity to play the game. Let's write some code to reset the scene.

At the top of your `update()` method, type the following:

```
1   // check key input for restart
2   if (this.gameOver && Phaser.Input.Keyboard.JustDown(keyR)) {
3       this.scene.restart();
4   }
```

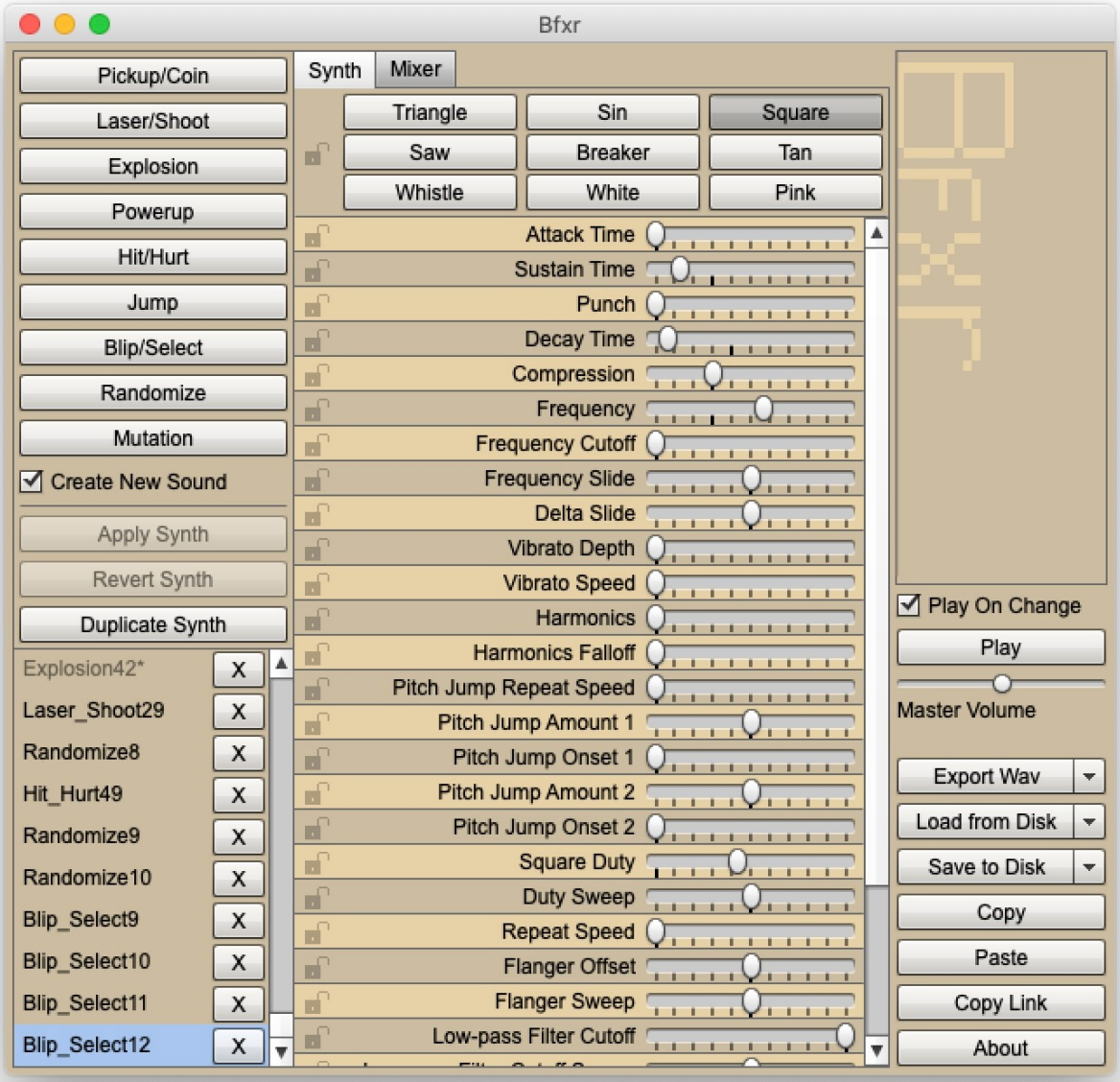
The `if()` loop is similar to the input checking we did in Rocket.js—we've just added a second condition (connected with the `&`) that makes sure the game is over before the scene can restart. And we restart the scene by using a slight variation of the `scene.start()` command we used way back at the beginning of the tutorial. Phaser's Scene Manager is really flexible. Scenes can move to other scenes, launch scenes in parallel, restart themselves, and even pass data to other scenes (and

themselves).

Pew-Pew-Pew

Games feel lifeless without sound, so let's spruce up *Rocket Patrol*'s audio space with some sound effects.

Note: Once again, if you don't want to make your own SFX assets, you can download mine from the project's GitHub repository.



For quick, synthy sound generation, Bfxr (pictured above) is a great tool to have on hand. It has a number of "preset" buttons on the left side that generate endless, randomized 8-bit bleeps and blorps. Click around until you hear something you like, then export the .wav file. Remember, we need a rocket firing sound, a spaceship explosion sound, and a UI selection sound. Once you have all three, drag

those files into your “assets” folder.

Like our other assets, we need to preload them before Phaser can add them to a scene. But this time, we’re going to preload them in our Menu scene, so they’re ready to use anywhere else in the game. Return to `Menu.js`, and type the following between your `constructor()` method and the `create()` method (keeping in mind that your file names may be different than mine):

```
preload() {
  // load audio
  this.load.audio('sfx_select', './assets/blip_select12.wav');
  this.load.audio('sfx_explosion', './assets/explosion38.wav');
  this.load.audio('sfx_rocket', './assets/rocket_shot.wav');
}
```

Phaser has a number of ways to handle audio, and we’ll need to use two different techniques due to how we’ve structured our code. First, head back to Play.js, and scroll to the `shipExplode()` function. This is the perfect place to play our explosion sound effect, because the function is triggered immediately after a rocket-to-spaceship collision occurs. When you need to play a quick, one-off sound (instead of, for instance, music or a looping sound effect), Phaser has a simple means to do so. Type this line at the bottom of your `shipExplode()` function:

```
this.sound.play('sfx_explosion');
```

Head back to the browser and test the game. You should hear a tasty explosion every time you blast a spaceship.

The rocket sound is trickier, because we encapsulated the rocket-firing code in its own class. This will require a bit more setup but only a few lines of code. Return to `Rocket.js`, and add the following line *inside* your existing `constructor()` method:

```
this.sfxRocket = scene.sound.add('sfx_rocket'); // add rocket sfx
```

Unlike the previous `sound.play()` command, we’re now binding an audio object to the *scene* we passed in as a parameter via the object constructor and assigning it to a property in the current context (ie, the Rocket object). Now that sound will be accessible in any method within the object.

Next, revise your fire button code to look like the following:

```
  // fire button
  if (Phaser.Input.Keyboard.JustDown(keyF) && !this.isFiring) {
    this.isFiring = true;
    this.sfxRocket.play();  // play sfx
  }
```

We’ve only added two things: first, the `.play()` command that will play the audio when the rocket fires; and second, an additional condition (`!this.isFiring`) to make sure the sound effect will only play when the rocket first launches, just in case the player is spamming the fire button while the rocket is mid-flight.

Test your game again, and you should hear both firing and explosions. For our final sound, we’re

going to head back to the Menu scene.

Revisiting Menu.js

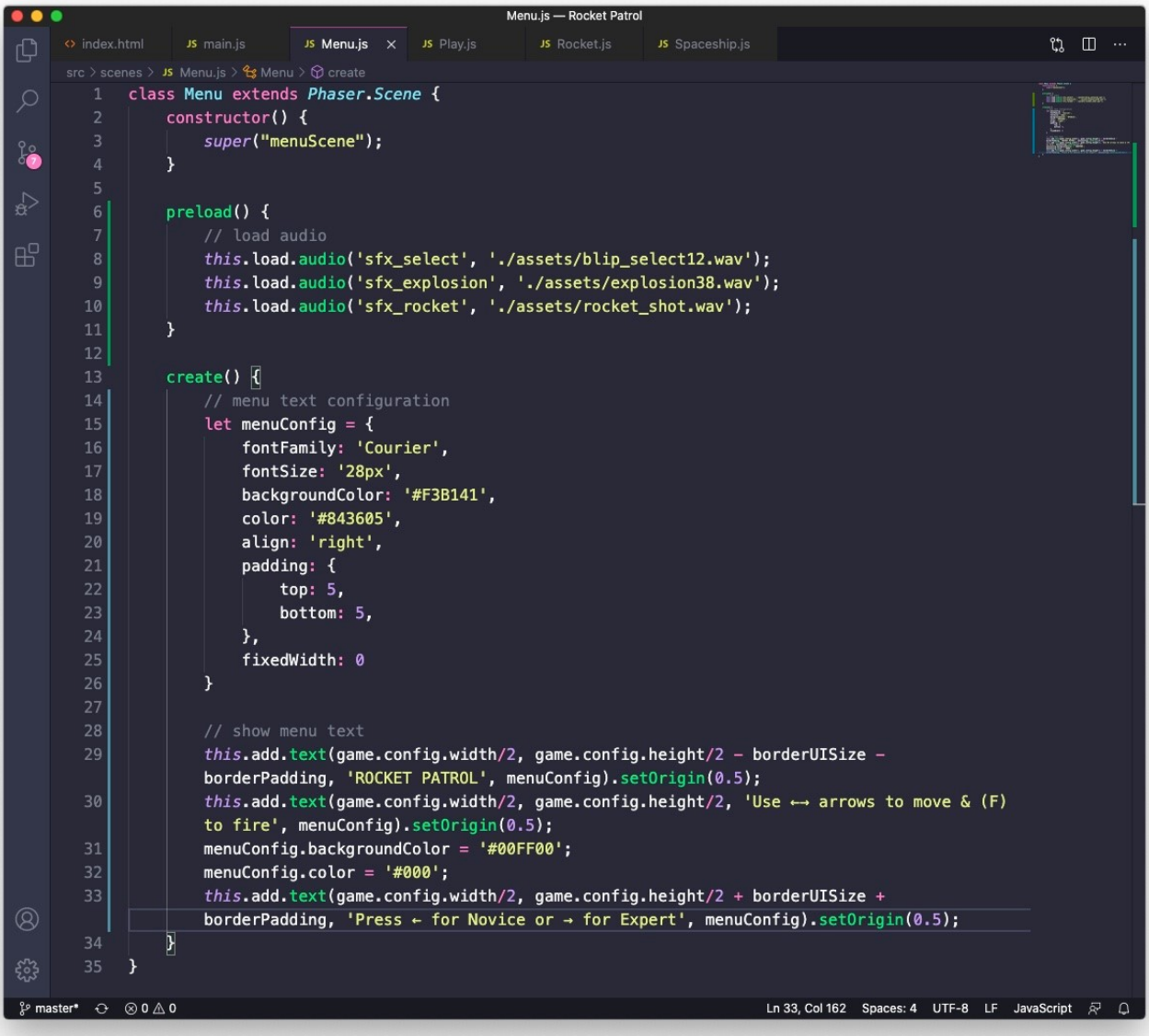
To complete the main menu (and save time), we're going to repurpose some code from other sections and edit some of our existing code. Remember, our game has had a menu scene since the beginning, but all it has done up to this point is (a) preload our audio (b) print text to the screen and (c) hand off control to the Play scene. However, steps (a) and (b) happen so fast that we never see them. As far as we're concerned, the scene transition is instantaneous.

Sprucing Up the Menu

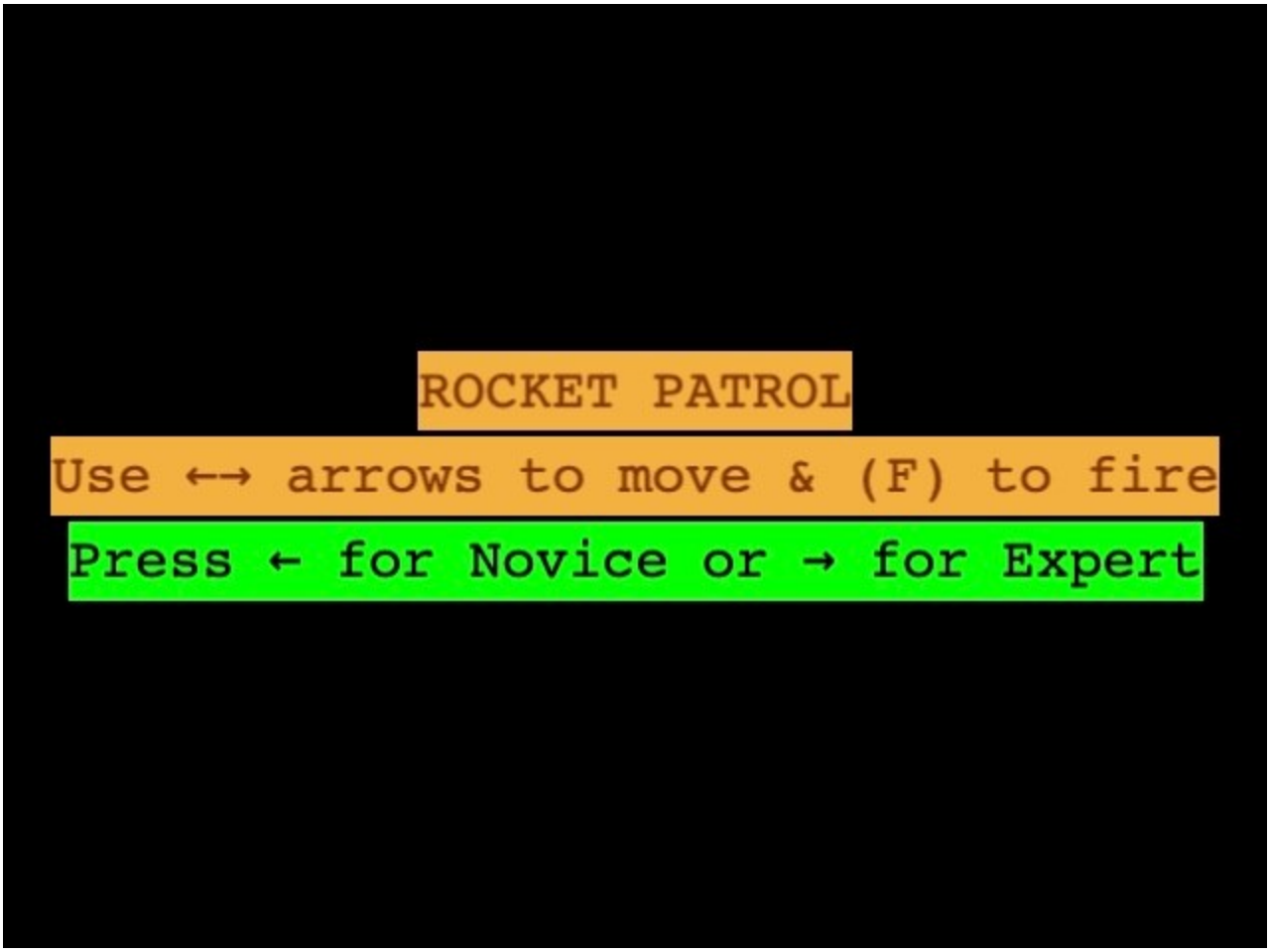
Before we add new code, return to Menu.js and delete the `this.add.text(20, 20, "Rocket Patrol Menu");` and `this.scene.start("playScene");` statements. For now, we want to remain on the menu so we can see our changes.

Now, go back to Play.js and copy the `scoreConfig` object declaration from `create()`. We're going to use the same style of text formatting in the main menu. Paste the code at the top of the `create()` method in Menu.js, change the object variable name to `menuConfig`, and change the `fixedWidth` property to `0`.

Next, we're going to display our menu text. Type in the following code:



This should look familiar from the “GAME OVER” text prompt we made earlier. The only difference is that I’m changing some properties in `menuConfig` to change the text and background colors. If you go back to the browser and test your game, you should see this:



It may be ugly and boring, but at least it's functional! Or wait, no it's not—what do the differing difficulties mean? Well, we haven't implemented them yet, so let's work on getting the menu working and implementing user-selectable difficulties.

First, let's borrow the key definition code from Play.js and insert the following at the end of the Menu.js create method:

```
// define keys
keyLEFT = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.LEFT);
keyRIGHT = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.RIGHT);
```

Difficulty Settings

Head back to Menu.js and create an `update()` method. Add the following code *below* the `create()` method:

```
update() {
  if (Phaser.Input.Keyboard.JustDown(keyLEFT)) {
    // easy mode
    game.settings = {
      spaceshipSpeed: 3,
      gameTimer: 60000
    }
    this.sound.play('sfx_select');
```

```
    this.scene.start('playScene');
  }
  if (Phaser.Input.Keyboard.JustDown(keyRIGHT)) {
    // hard mode
    game.settings = {
      spaceshipSpeed: 4,
      gameTimer: 45000
    }
    this.sound.play('sfx_select');
    this.scene.start('playScene');
  }
}
```

Test your game again, and make sure both arrow keys take you to the game (and that you hear the SFX when you press the key). Remember, the settings don't mean anything yet. Let's tackle those next.

Return to Spaceship.js and change the `this.moveSpeed = 3;` statement to:

```
this.moveSpeed = game.settings.spaceshipSpeed;
```

Next, hop over to Play.js, in the create method, in our play clock code, and update the hard-coded clock value in your play clock from `60000` to `game.settings.gameTimer`.

Now, if you test your game, selecting Expert mode should increase the spaceships' speed from 3 to 4 pixels/frame and decrease your timer from 60 to 45 seconds.

Returning to the Menu

One last bit of cleanup: once the player gets a game over, let's give them the chance to reset to the menu, just in case they want to change the difficulty. Doing so only requires two quick changes.

Return to Play.js and edit the '`Press (R) to Restart`' message to read '`Press (R) to Restart or ← for Menu`'. Then, in `update()`, below our existing key input check, add the following code:

```
        if (this.gameOver && Phaser.Input.Keyboard.JustDown(keyLEFT)) {
            this.scene.start("menuScene");
        }
```

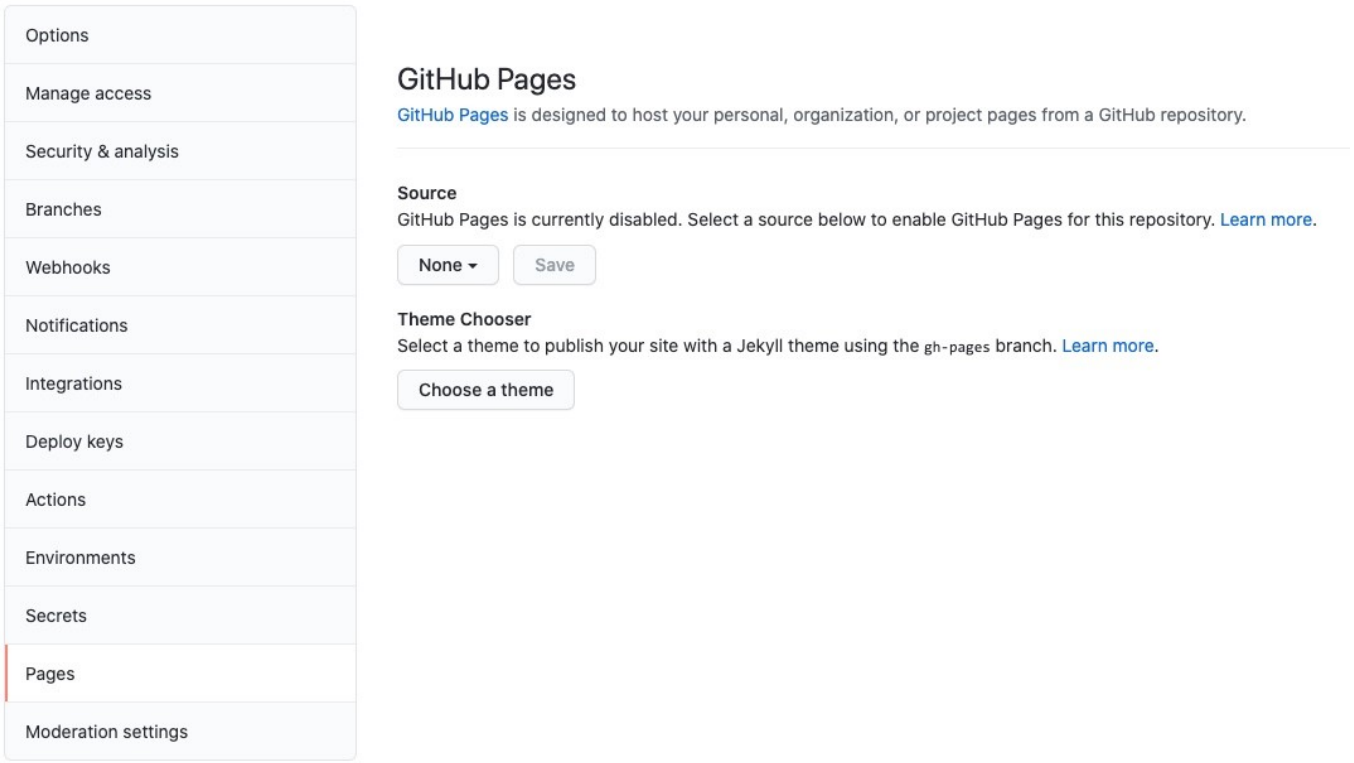
And that's it! If you return to the browser, you'll see that you've now completed a fully-functional game. Well done!

Do It Live!

Now that you've spent all this time creating a web game, why not actually *put it on the web*? The good news is, since we've been hosting our code on GitHub, there's a straightforward way to "publish" our game.

Return to your GitHub page, click on your Rocket Patrol repository if you're not there already, then

click the ⚙ Settings tab in the toolbar just below your repository name. Once there, you'll see a column to the left listing a number of subcategories. Click the Pages tab near the bottom of that column. You should arrive at a page that looks like this:



Under the **Source** heading, click the dropdown menu that says "None" and select the branch where your project resides. (This should be "master," unless you've been doing some extracurricular git experimentation.) Then, click the Save button.

And that's it! You should see a blue info bar appear that provides a live link to your hosted game. Click it, and revel in your game design prowess. Once you're done with your revelry, share your link with others so they can shower you with praises. 🙇

Note: If the published game doesn't look like you'd expect, make sure you saved, committed, and pushed your lastest changes to GitHub. GitHub Pages is serving the code stored on GitHub, *not* the code you're working on in VS Code. 👍

Wrapup

This tutorial is not meant to be an example of best practices for game programming. We used some shortcuts to get here, but sometimes shortcuts are fine. The game is functional, fun (imo), and *finished*.