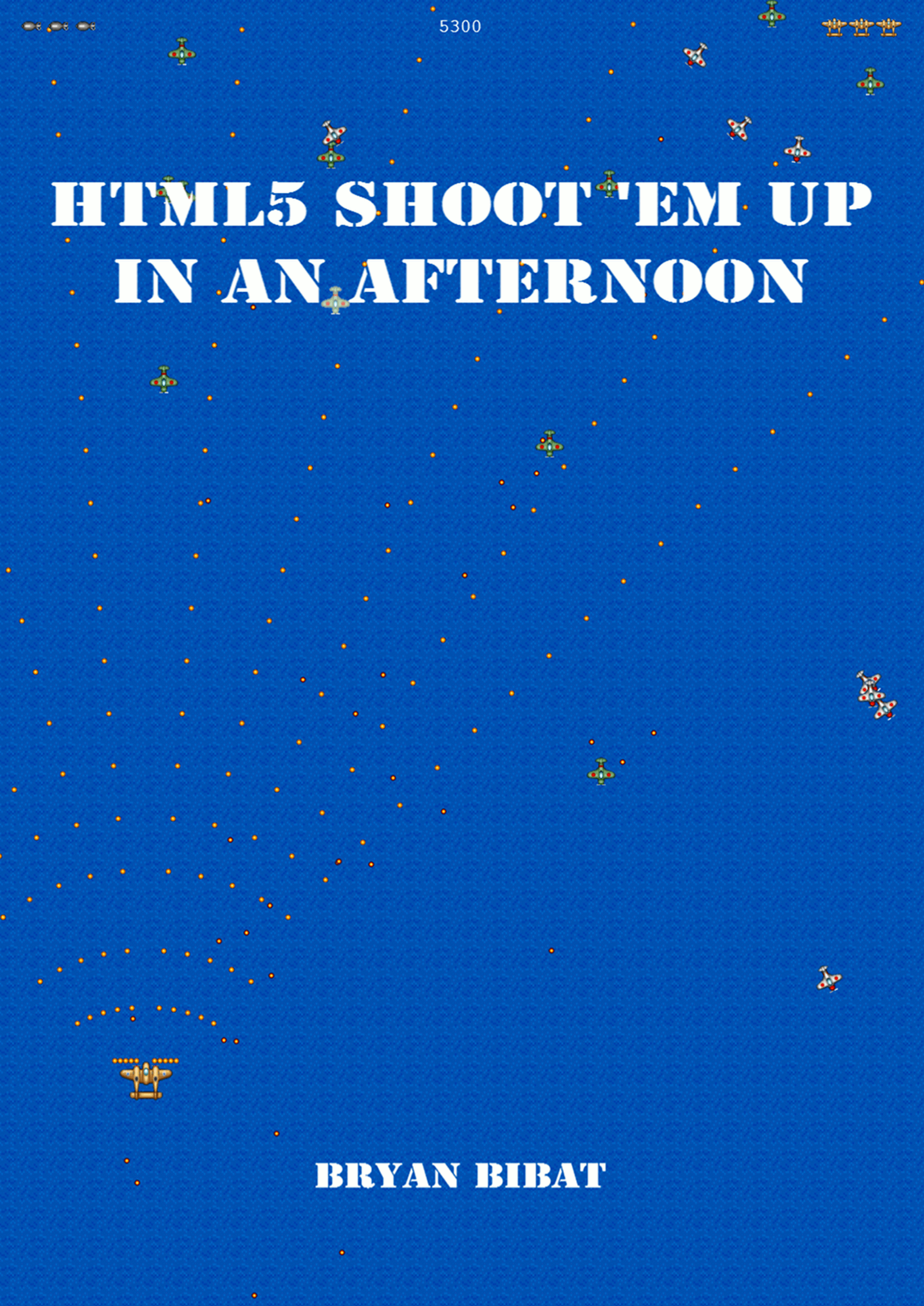


HTML5 SHOOT 'EM UP IN AN AFTERNOON

BRYAN BIBAT

HTML 5 Shoot 'em Up in an Afternoon

Learn (or teach) the basics of Game Programming with this free Phaser tutorial

Bryan Bibat

This book is for sale at http://leanpub.com/html5shootemupinanafternoon

This version was published on 2014-07-11

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

Contents

Preface . . . . . **i**
License i

Introduction **1**
Who is this book for? 1

Morning: Preparing for the Afternoon **2**
Introduce them to Shoot ‘em Ups 2
Technical Requirements: JavaScript and Math 2
Development Environment Setup 3
Other Suggested Prior Reading 3

Afternoon 0: Overview of the Starting Code **4**

Afternoon 1: Sprites, the Game Loop, and Basic Physics **7**
Sprite Basics 7
The Game Loop 13
Apply Physics 15

Afternoon 2: Player Actions **20**
Keyboard Movement 20
Mouse/Touch Movement 22
Firing Bullets 24

Afternoon 3: Object Groups **27**
Convert Bullets to Sprite Group 27
Enemy Sprite Group 29
Player Death 30
Convert Explosions to Sprite Group 32
Refactoring 35

Afternoon 4: Health, Score, and Win/Lose Conditions **39**
Enemy Health 39
Player Score 41
Player Lives 42
Win/Lose Conditions, Go back to Menu 44

Afternoon 5: Expanding the Game **48**
Harder Enemy 48
Power-up 54
Boss Battle 58

Sound Effects 63

Afternoon 6: Wrapping Up 66
Restore original game flow 66
Sharing your game 67

Evening: What Next? 70
Challenges 70
What we didn't cover 73

Appendix A: Environment Setup Tutorials 74
Basic Setup 74
Advanced Setup 77
Cloud IDE Setup 78

Appendix B: Expected Code Per Chapter 83

Preface

I'll be honest and get this out as early as possible: *I'm not a "professional" game developer*. Looking at my other Leanpub books will tell you that I'm more into web development. Heck, if you told me a few months ago that I would be putting out a game development book, I would've thought you're crazy.

This book was a result of three things that happened to occur around the same time:

First was the problem that came up with our HTML5 workshop. The original lecturer bailed at the last minute and we had problems with finding a replacement. We even considered the worst case, cutting out the hands-on portion leaving us with a morning "workshop" consisting only of talks from people in the local gaming industry.

Coincidentally, I was playing around with *Phaser* a few weeks before the event. While I am not a game developer, I had just enough knowledge to make a simple workshop to introduce basic game concepts via the said HTML5 game framework. In the end I volunteered to take over the workshop less than four days before the actual event.

Normally I would have prepared a hundred or so slides and go through them during the workshop. But earlier that week I had the rare opportunity to talk to the first person who gave me advice when I started out teaching, and one of the things we talked about the not-so-recent trend of lazy college professors making only slides leaving a big gap between them and textbooks. This convinced me to switch things up with the workshop – instead of giving the participants a link to SpeakerDeck, I would point them to Leanpub.

It took a few sleepless nights to write the original 36-page workbook, but it was worth it: I had a much easier time conducting the workshop than I would have if I went with slides.

The positive response from the participants also convinced me to spend some more time to improve this book and get it out there for anyone interested in learning the basics of game development.

License

Introduction

This is usually the part where books give a lengthy intro about HTML5 to increase their word count. This is not one of those books.

All you need to know about HTML5 is that it allows you to do stuff in your browser, regardless if it's on a desktop PC or a mobile phone, without the need for extra plugins. And that includes making games. If you want a better intro to HTML5, head over to Dive Into HTML5.

As the title and cover of the book implies, we will introduce you to both HTML5 and game development by guiding you in making a shoot-em-up game similar to the classic video game 1942.

There are a number of HTML5 libraries and frameworks out there right now. For this afternoon workshop, we'll be using **Phaser**, an open-source framework built on top of Pixi.js. It's a higher-level framework: it's bigger and may feel like you have much less control (i.e. *magical*) compared to other frameworks, but at the same time, you need far less code to get things done and this makes it suitable for a short workshop such as this one.

Who is this book for?

This book is for people who want to learn the basic concepts behind creating games. As a workshop manual, it is also for experienced developers interested in introducing those concepts to those people. With these in mind, here are some possible setups for using *HTML5 Shoot 'em Up in and Afternoon*:

- **Self-study** - AKA your run-of-the-mill tutorial where you just go through the book from cover-to-cover. Web developers with extensive experience in JavaScript will find the code in this book easy and fairly straightforward. Novice programmers might not get the same pleasant experience, especially those who have not yet coded in JS enough to understand its quirks.
- **Pair or Small-group Study** - Spend an afternoon teaching game programming to your daughter / cousin / nephew. It's recommended to go through the book once or twice beforehand to make sure things go smoothly. (Unless of course you want to expose the kid to the reality of "spend minutes or hours looking for the copy-paste typo" software development.)
- **Workshop** - What this book was originally written for. Gather a group of people interested in making games in HTML5 and go through the tutorial at a slower pace. An experienced instructor (i.e. worked with *Phaser* for some time, gone through the tutorial multiple times) can lead a workshop of 20 without a hitch, but for larger groups or groups with less programming experience, you may need to get a few extra mentors to help.

Morning: Preparing for the Afternoon

For instructors tutoring children or other individuals with little programming or even gaming experience, we recommended spending a few hours in the morning to make sure things go smoothly in the afternoon.

If you're the student, it's best you skip this part so as not to spoil what your teacher is going to ask you to do.

Introduce them to Shoot 'em Ups

It may sound weird for us who grew up in the '80s and '90s where shoot 'em ups were staple arcade games, but there is a very slight possibility that the person you're teaching may not be familiar with the genre.

If that's the case, then you need to let them play a few shoot 'em ups before starting the workshop. They must first understand the basic concepts around the genre, knowing what makes those games fun an challenging. At the worst case, finding out that they hate the genre will let you end the session early and spare you from an unproductive afternoon.

An obvious choice would be *1942*, as it has been ported and remade so many times that you can find one on pretty much any platform.

Then there are Flash games from sites like Newgrounds and Kongregate. As HTML5 is supposed to replace Flash, letting your student play these games will give them an idea on what they can make in the future.

Steam also has a good collection of shmups. Jamestown deserves special mention because it lets you play with your students via local co-op.

Technical Requirements: JavaScript and Math

Theoretically, you *can* conduct a workshop with students who have no prior knowledge of JavaScript. They will be at the mercy of the copy-paste gods however, and it's also safe to say that they won't retain much after this workshop is over.

For best results, students who aren't familiar with programming or JavaScript must take a crash course in the morning. You don't need to go all the way into advanced JavaScript - knowing how to make and use functions and objects as well as using browser's developer consoles for debugging should be enough for the workshop. MDN has a good list of JavaScript tutorials that you and your students can choose from for this purpose.

In addition to programming skills, students should know basic Trigonometry. *Phaser* already handles most of the calculation but knowing stuff like sine/cosine and polar coordinates will make it easier for them to visualize what's going on under the hood. They will also directly use those concepts at the latter part of the workshop where we rotate sprites and generate patterns for the boss battle.

While there are many online tutorials out there for trigonometry (Khan Academy comes to mind), I have yet to see one that is better than your usual high school trigonometry class while accessible to younger students. You might even say that the other way around, introducing kids to trigonometry through game concepts, would be a better approach[1].

[1] True story: I discovered sine and cosine as way to make things spin or bob up and down back when I was a kid playing around with BASIC, two years before I had trigonometry class.

If you still wish to quickly introduce basic trigonometry to your students before the workshop, look for visually impressive and interactive demos like *How to Fold a Julia Fractal*.

Development Environment Setup

All you need to code in *Phaser* is a browser that supports HTML5 (e.g. Chrome, Firefox), a web server, and the text editor of your choice.

You *have* to use a web server to test your game in this tutorial. The first part of *Getting Started With Phaser* explains why this is the case.

As for the text editor, any editor or IDE with JavaScript support (syntax highlighting, automatic indent/brackets) and can parse non-Windows line endings (i.e. not Notepad) will do. If your preferred editor does not fit these requirements, we suggest downloading the free trial of Sublime Text.

Once you have setup your web server and text editor, download the basic game template from Github, extract it to the folder served by the web server, and start coding.

More detailed information about setting up your development environment (like choosing a web server) can be found at *Appendix A: Environment Setup Tutorials*.

Other Suggested Prior Reading

Apart from JS and Math, we suggest that you at least skim through the following to give you an idea about what we are going to do:

- **Getting Started with Phaser** - Phaser's own guide setting up a development environment
- **Phaser Examples** - view demos of Phaser's features.
- **Phaser Documentation** - your typical API docs with link to source.

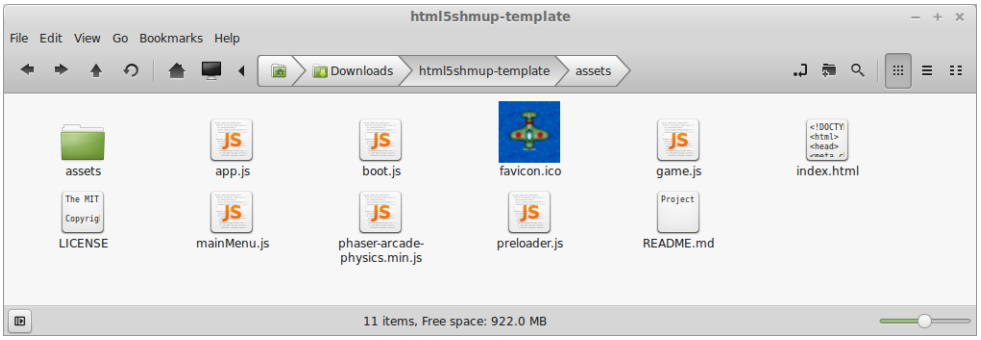
In addition to Phasers documentation, the following may give you insights on making games in Phaser:

- **Game Programming Patterns** - like many game frameworks, Phaser uses the *Game Loop* pattern at its core.
- **Game Mechanic Explorer** - a somewhat short list of game mechanics, all implemented in Phaser.

Afternoon 0: Overview of the Starting Code

By now you should have finished setting up your development environment, with your web server up and your editor open to the folder containing the base code for the tutorial. If not, please refer again to the Development Environment Setup in previous chapter.

Before we proceed to the actual tutorial, let's take a tour of the starter template:

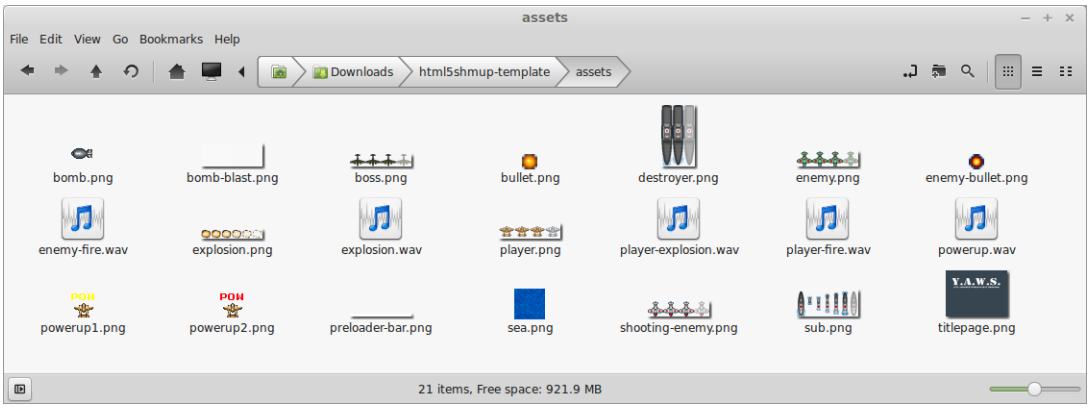


This template is based on the Basic Game Template found in the `resources/Project Templates` folder of the *Phaser* Git repository. We're using this because it follows a more modular approach compared to most of the *Phaser Examples* and therefore much closer to real-life apps.

Let's do a quick run-through of the files:

- `index.html` - our main HTML5 page that links all our files together. There's not much to say about this except for the `<div id="gameContainer"></div>` which *Phaser* will use to draw the Canvas on to.
- `phaser-arcade-physics.min.js` - *Phaser* stripped of 2 other physics engines (retaining only "Arcade" physics) and minified. You can replace this later on with the full version if you plan to use the other physics engines or if you want to the refer to the original code while developing.
- `app.js` - the code that kicks off the app. Creates the `Phaser.Game` object and adds the `States`.
- `boot.js`, `preloader.js`, `mainMenu.js`, `game.js` - the different states of our game, combined together by `app.js` to form the flow of our app:
  - **Boot** - The initial state. Sets up additional settings for the game. Also pre-loads the image for the pre-loader progress bar before passing the game to **Preloader**.
  - **Preloader** - Loads all assets before the actual game. Once that's done, the game proceeds to **MainMenu**.
  - **MainMenu** - The title screen and main menu before the actual game.
  - **Game** - The actual game.

Reading through the JS files and the comments within will give you a peek of what to expect from *Phaser*.



The template also includes all the necessary sprites and sounds for the basic game, saving you hours of looking for or making your own game assets. The sprites were taken from Ari Feldman's open-sourced sprite compilation SpriteLib while the sounds were from Devin Watson's OpenGameArt.org portfolio.

With the code tour out of the way, we can now move on to the tutorial.

Code Examples

You will see sample code throughout this manual. The text decoration in the code will tell you what you need to do to the existing code.

For example, let's modify `game.js` to make the background scroll vertically:

```
update: function () {
~~  // Honestly, just about anything could go here. It's YOUR game after all...~~
  this.sea.tilePosition.y += 0.2;
},
```

In the code example above, there is a strikethrough on the comment. Strikethrough means you need to delete those lines. On the other hand, the following line is in **boldface**, which means you need to insert those lines at that position.

Some examples for inserting functions will also have line numbers to tell you where to insert those functions. They will also give you an idea if you've properly added the code up to that point.

Skipping Main Menu

We'll be modifying our code many times throughout this tutorial. Skipping the boot, pre-loading, and main menu in order to go directly to our game, will save us a click after the refresh every time we make a change. To skip those states, change the starting state in `app.js`:

```
~~  game.state.start('Boot');~~
  game.state.start('Game');
```

And since we're skipping the `preloader.js`, we'll copy over the sea background asset loading to `game.js`:

```
BasicGame.Game.prototype = {
  preload: function () {
    this.load.image('sea', 'assets/sea.png');
  },

  create: function () {
```

Afternoon 1: Sprites, the Game Loop, and Basic Physics

In this first part, we'll go over how to draw and move objects on our game.

Sprite Basics

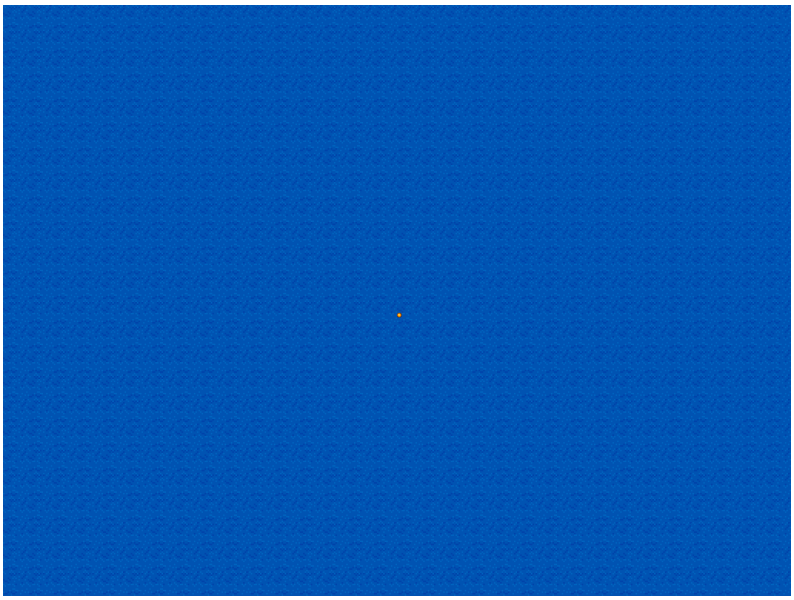
Draw Bullet Sprite

Let's start with something basic – drawing an object on the game stage. The most basic object in *Phaser* is the *Sprite*. So for our first piece of code, let's load then draw a bullet sprite on our game by making the following modifications to `game.js`. (All of the code examples in this tutorial refer to `game.js` unless otherwise noted.)

```
preload: function () {
  this.load.image('sea', 'assets/sea.png');
  this.load.image('bullet', 'assets/bullet.png');
},

create: function () {

  this.sea = this.add.tileSprite(0, 0, 1024, 768, 'sea');

  this.bullet = this.add.sprite(512, 400, 'bullet');

}
```

We called the following functions:

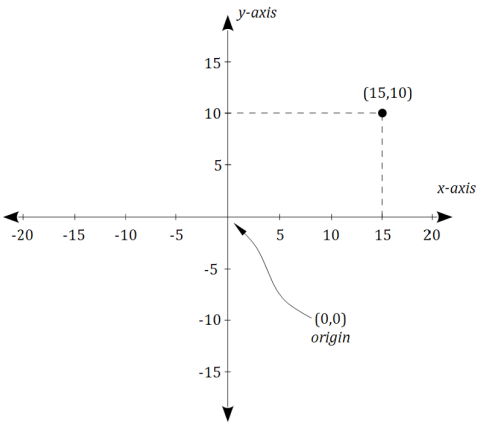
- `load.image()` - loads an image (e.g. `assets/bullet.png`) and assigns it a name (e.g. `bullet`) which we use later.
- `add.sprite()` - accepts the x-y coordinates of our sprite and the name of the sprite which we assigned in the `load.image()` function.



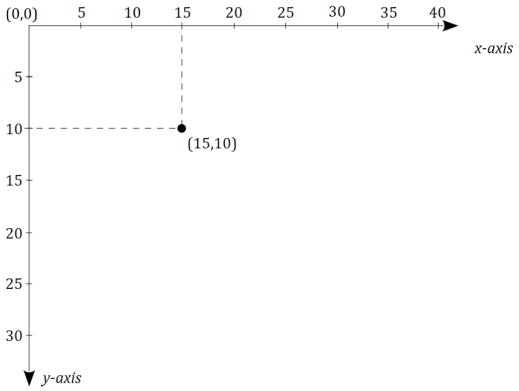
Bullet sprite added into our game

Screen Coordinates vs Cartesian Coordinates

At around middle school, children learn about the Cartesian coordinate system where points, defined by an ordered pair `(x, y)`, can be plotted on a plane. The center is `(0, 0)`, x-values increase as you go right, while y-values increase as you go up.

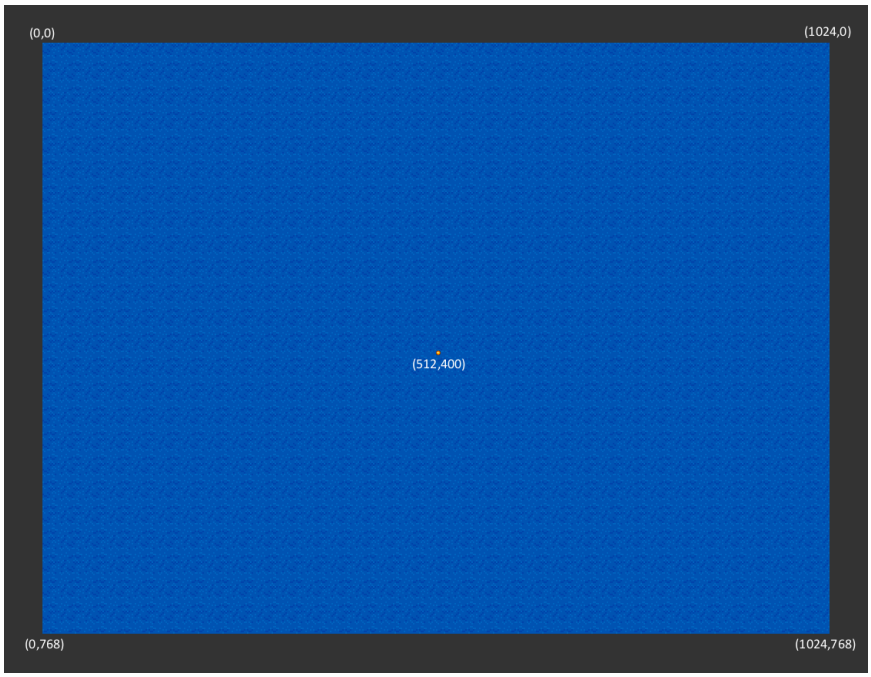


Cartesian coordinate system



screen coordinate system

However, computer displays do not use Cartesian coordinates *as is* but instead use a variation: instead of being at the center, `(0, 0)` represents the point at the top-left, and instead of decreasing, y-values increase as you go down. This picture illustrates the screen coordinate system in our game at the moment:


(0,0)
(1024,0)
(512,400)
(0,768)
(1024,768)

A note about the *Phaser Examples*

The biggest difference between the *Phaser Examples* and our game template is that the former uses global variables while we're adding States which encapsulate the logic of our game. This means that you can't copy the code from those examples directly. For example, *01 - load an image* uses the following syntax:

```
game.add.sprite(0, 0, 'einstein');
```

We don't have a `game` global variable within the scope in our `BasicGame.Game` state object. Instead, we have a `this.game` property so translating the code above into our template would be:

```
this.game.add.sprite(0, 0, 'einstein');
```

Adding `this.game` over and over in your code might make you wish for a global variable. Fortunately, *Phaser* also adds the other game properties into the state. You can see a list of this in the original game template:

```
BasicGame.Game = function (game) {

    // When a State is added to Phaser it automatically has the following
    //  properties set on it, even if they already exist:

    this.game;      // a reference to the currently running game
    this.add;       // used to add sprites, text, groups, etc
    this.camera;    // a reference to the game camera
    this.cache;     // the game cache
    this.input;     // the global input manager (you can access this.input.keyboard,
                    //   this.input.mouse, as well from it)
    this.load;      // for preloading assets
    this.math;      // lots of useful common math operations
    this.sound;     // the sound manager - add a sound, play one, set-up markers, etc
    this.stage;     // the game stage
    this.time;      // the clock
    this.tweens;    //  the tween manager
    this.state;     // the state manager
    this.world;     // the game world
    this.particles; // the particle manager
    this.physics;   // the physics manager
    this.rnd;       // the repeatable random number generator

    // You can use any of these from any function within this State.
    // But do consider them as being 'reserved words', i.e. don't create a property
    // for your own game called "world" or you'll over-write the world reference.

};
```

In other words, you only need to use `this.add` in place of `this.game.add`:

```
this.add.sprite(0, 0, 'einstein');
```

Which is exactly what we used for adding a sprite above.

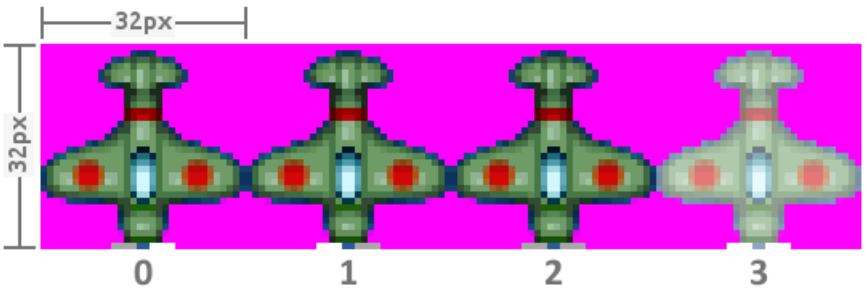
Draw Enemy Animation

Let's then proceed with something more complicated, an animated sprite.

We first load a sprite sheet, an image containing multiple frames, in the pre-loading function.

```
preload: function () {
  this.load.image('sea', 'assets/sea.png');
  this.load.image('bullet', 'assets/bullet.png');
  this.load.spritesheet('greenEnemy', 'assets/enemy.png', 32, 32);
},
```

Instead of `load.image()`, we used `load.spritesheet()` to load our sprite sheet. The two additional arguments are the width and height of the individual frames. Since we defined `32` for both width and height, *Phaser* will load the sprite sheet and divide it into individual frames like so:

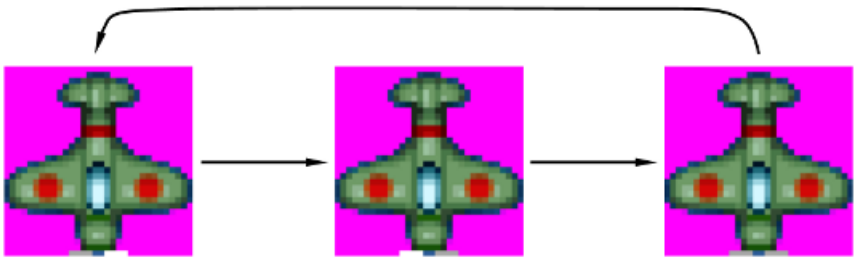


Enemy sprite sheet (magenta refers to the transparent parts of the image)

Now that the sprite sheet is loaded, we can now add it into our game:

```
create: function() {

  this.sea = this.add.tileSprite(0, 0, 1024, 768, 'sea');

  this.enemy = this.add.sprite(512, 300, 'greenEnemy');
  this.enemy.animations.add('fly', [ 0, 1, 2 ], 20, true);
  this.enemy.play('fly');

  this.bullet = this.add.sprite(512, 400, 'bullet');
}
```

The `animations.add()` function specified the animation: its name, followed by the sequence of frames in an array, followed by the speed of the animation (in frames per second), and a flag telling whether the animation loops or not. So in this piece of code, we defined the `fly` animation that loops the first 3 frames of the green enemy sprite sheet, an animation of the propeller spinning:



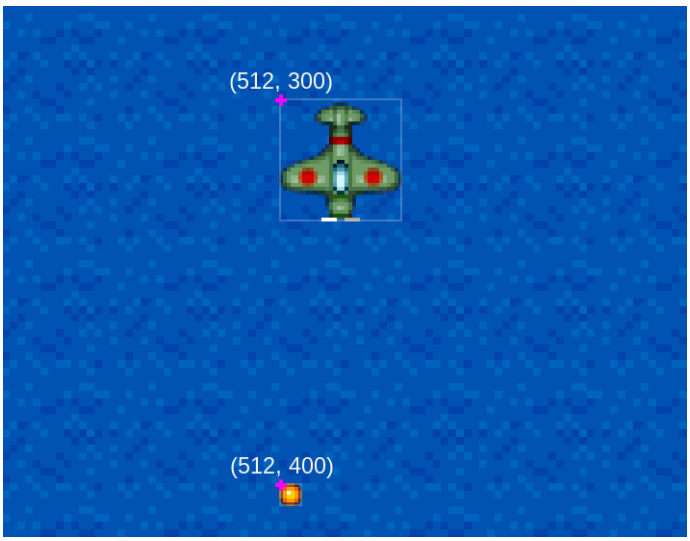
Ordering

Note how we added the bullet sprite after the enemy sprite. As we shall see later, this declaration will put the bullet sprite above the enemy sprite.

There are ways to rearrange the order of the sprites (e.g. top to bottom) but the simplest way is to create them already in a bottom-to-top order.

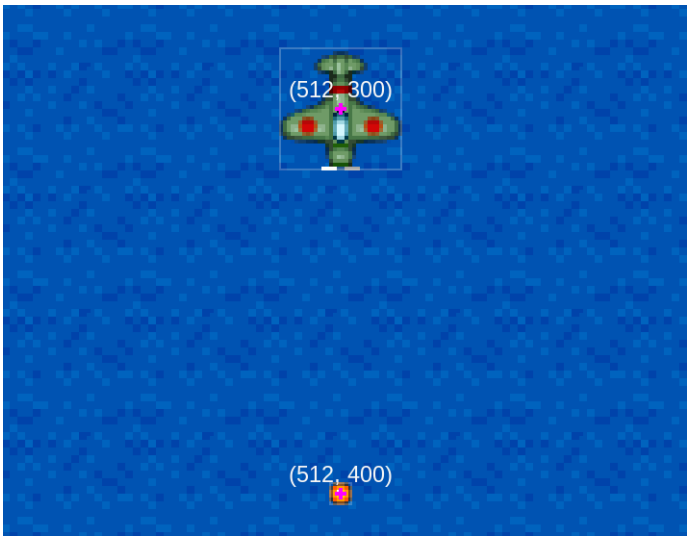
Set Object Anchor

The sprites share the same x-coordinate, so by default they are left-aligned.



For games, however, most of the time we want the x-y coordinates to be the center of the sprite. We can do that in *Phaser* by modifying the anchor settings:

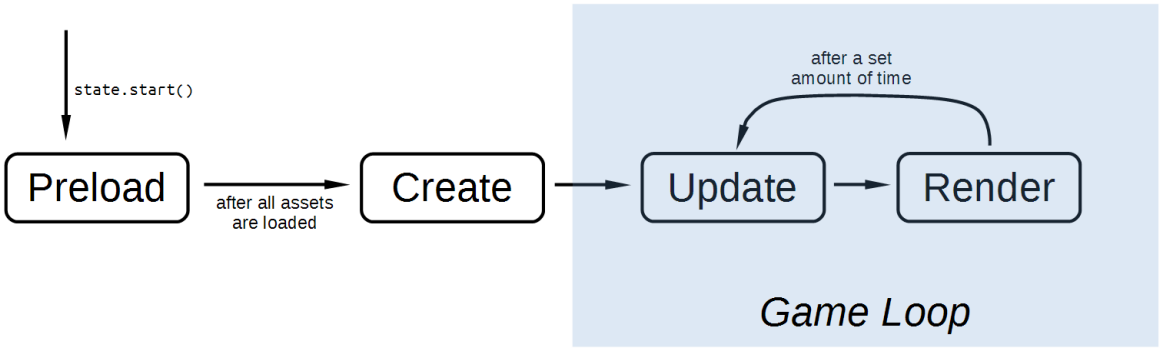
```
this.enemy = this.add.sprite(512, 300, 'greenEnemy');
this.enemy.animations.add('fly', [ 0, 1, 2 ], 20, true);
this.enemy.play('fly');
this.enemy.anchor.setTo(0.5, 0.5);

this.bullet = this.add.sprite(512, 400, 'bullet');
this.bullet.anchor.setTo(0.5, 0.5);
```



The `(0.5, 0.5)` centers the sprite. On the other hand, `(0, 0)` will mean that the x-y coordinate defines the top-left of the sprite. Similarly, `(1, 1)` put the x-y at the bottom right of the sprite.

The Game Loop

The following is an oversimplified diagram on what happens when *Phaser* games run:



- **Preload** - The game starts with a pre-load section where all assets are pre-loaded. Without pre-loading, the game will stutter or hang in the middle of gameplay because it has to load assets.
- **Create** - After pre-loading all assets, we can now setup the initial state of the game.
- **Update** - At a set interval (usually 60 times per second), this function is called to update the game state. All updates to the game are done here. For example, checking if the character has collided with the enemy, spawning an enemy at a random location, moving a character to the left because the player pressed the left arrow key, etc.
- **Render** - coming after Update, here is where the latest state of the game is drawn (rendered) to the screen.

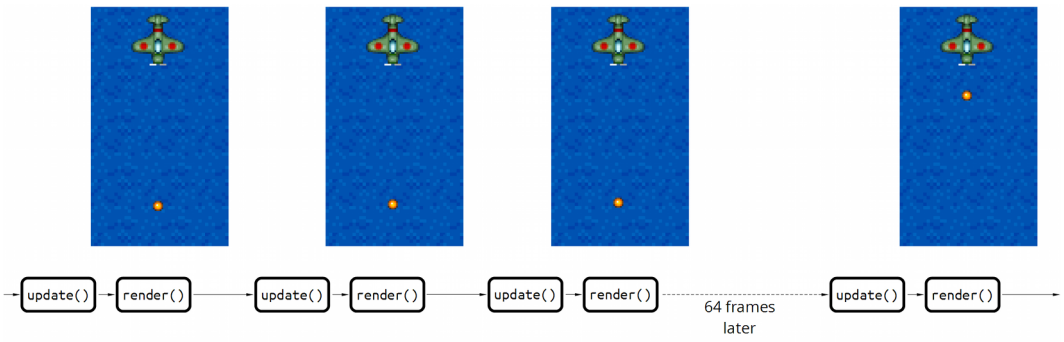
The update-render loop is what's called the ***Game Loop***, and is the heart of almost every computer game. You can read more about the Game Loop at the Game Programming Patterns site.

Move Bullet via update()

Now that we know how the game loop is implemented in *Phaser*, let's move our bullet sprite vertically by reducing its y-coordinate in the `update()` function:

```
update: function () {
  this.sea.tilePosition.y += 0.2;
  this.bullet.y -= 1;
},
```

As mentioned above, *Phaser* will call the `update()` function at a regular interval, effectively moving the bullet upwards at a rate of around 60 pixels per second.



This is how you move sprites in most basic game libraries/frameworks. In *Phaser*, though, we can let the physics engine do almost all of the dirty work for us.

Missing render()?

Before we discuss how to use *Phaser*'s physics engines, let's explain why we still don't have a `render()` function and yet the game renders the game state on its own.

First off, as some might have noticed from the `app.js`, we're only coding a portion of the game called the `state` which, as the name implies, is a state of the game.

```
game.state.add('Boot', BasicGame.Boot);
game.state.add('Preloader', BasicGame.Preloader);
game.state.add('MainMenu', BasicGame.MainMenu);
game.state.add('Game', BasicGame.Game);
```

The state is just one of the many things updated and rendered in *Phaser*'s game loop. For instance, here's what the `Game` object calls on update (pre- and post-update hooks removed):

```
this.state.update();
this.stage.update();
this.tweens.update();
this.sound.update();
this.input.update();
this.physics.update();
this.particles.update();
this.plugins.update();
```

And here's the render section:

```
  this.renderer.render(this.stage);
  this.plugins.render();
  this.state.render();
  this.plugins.postRender();
```

We don't need to write code to render our sprites because that is already covered by the first line, `this.renderer.render(this.stage);`, with `this.stage` containing all of the sprites currently in the game.

We'll write some render code later for debugging purposes.

Apply Physics

Phaser comes with 3 physics systems, *Arcade*, *Ninja*, and *P2*. Arcade is the default and the simplest, and so we'll use that.

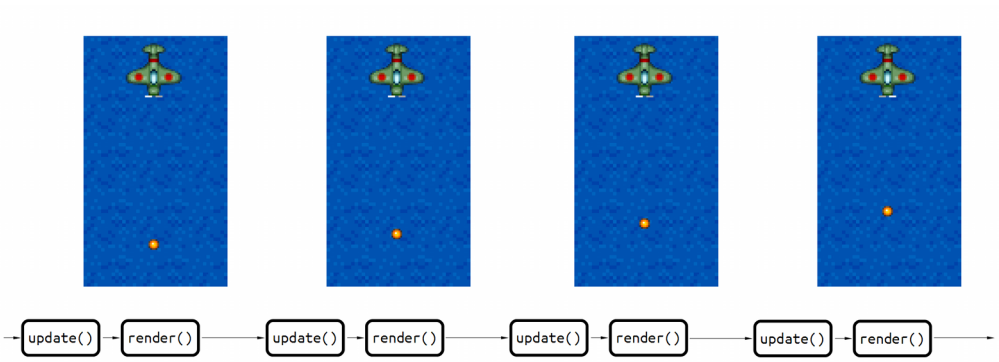
(And besides, the version of *Phaser* bundled with the basic template, `phaser-arcarde-physics.min.js`, contains *only* Arcade physics to reduce download file size.)

Velocity

Once we put our bullet into the Arcade physics system, we can now set its velocity and let the system handle all the other calculations (e.g. future position).

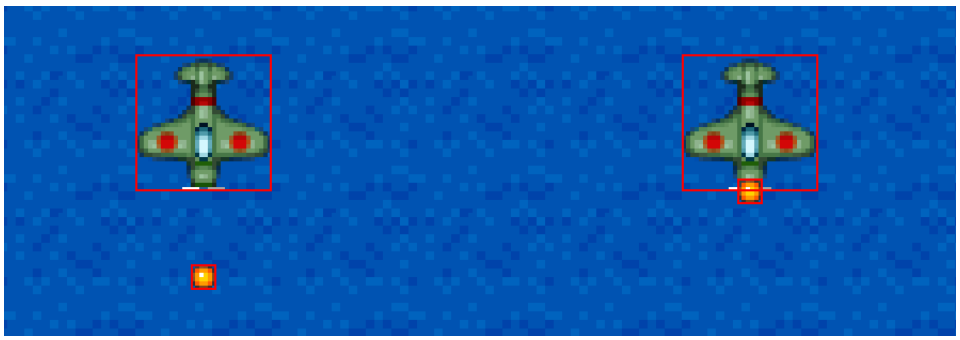
```
    this.bullet = this.add.sprite(512, 400, 'bullet');
    this.bullet.anchor.setTo(0.5, 0.5);
    this.physics.enable(this.bullet, Phaser.Physics.ARCADE);
    this.bullet.body.velocity.y = -400;

  },

  update: function () {
    this.sea.tilePosition.y += 0.2;
    this.bullet.y -= 1;
  },
```

With the physics enabled and velocity set, our sprite's coordinates will now be updated by the `this.physics.update();` call rather than our `update` code. In this case, "`velocity.y = -400`" is 400 pixels per second upward; at 60 frames per second, each `update` call will move the bullet up 6-7 pixels.



Show Body Debug

Arcade physics is limited to *axis-aligned bounding box* (AABB) collision checking only. In simpler terms, all objects under Arcade are rectangles.



bounding boxes (hitboxes) of the sprites outlined in red; the sprites to the right are colliding with each other

We can view these rectangles by rendering these areas with the debugger. First we add the enemy sprite to the physics system:

```
    this.enemy.play('fly');
    this.enemy.anchor.setTo(0.5, 0.5);
    this.physics.enable(this.enemy, Phaser.Physics.ARCADE);

    this.bullet = this.add.sprite(512, 400, 'bullet');
```

Then we add the debugging code under our currently nonexistent `render()` function:

```
30    update: function () {
31      this.sea.tilePosition.y += 0.2;
32    },
33
34    render: function() {
35      this.game.debug.body(this.bullet);
36      this.game.debug.body(this.enemy);
37    },
```

Collision

Once added to the physics system, checking collision and overlapping is only a matter of calling the right functions:

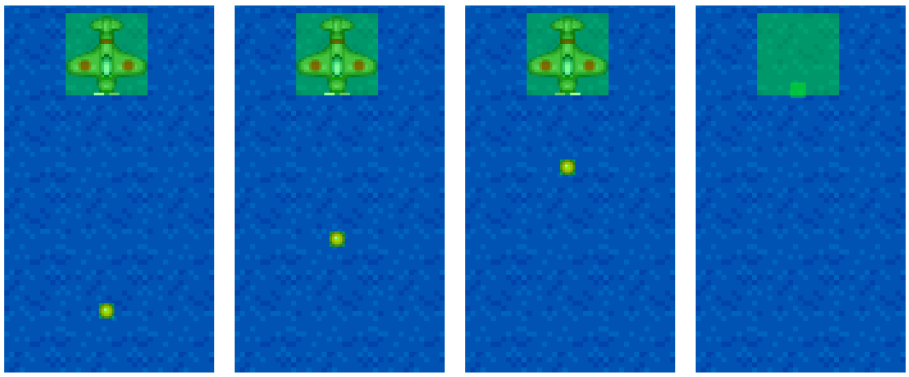
```
update: function () {
  this.sea.tilePosition.y += 0.2;
  this.physics.arcade.overlap(
    this.bullet, this.enemy, this.enemyHit, null, this
  );
},
```

The `overlap()` function requires a callback which will be called in case the objects overlap. Here's the `enemyHit()` function:

```
42  enemyHit: function (bullet, enemy) {
43    bullet.kill();
44    enemy.kill();
45  },
```

Being common situation in games, *Phaser* provides us with a `sprite.kill()` function for "killing" sprites. Calling this function both marks the sprite as dead and invisible, effectively removing the sprite from the game.

Here's the collision in action:



With debug on, we can see that the sprite is still at that location but it's invisible and the physics engine ignores it (i.e. it no longer moves).

Remove Debugging

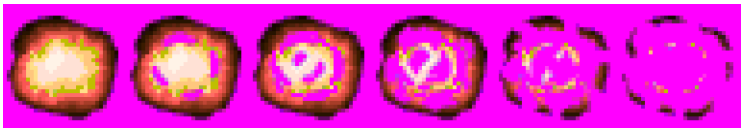
Debugging isn't really required in this workshop so you should probably remove or comment out the debugging code when you're done testing.

```
render: function() {
    this.game.debug.body(this.bullet);
    this.game.debug.body(this.enemy);
},
```

Explosion

Before we proceed to the next lesson, let's improve our collision handling by adding an explosion animation in the place of the enemy. Here's the animation pre-loading:

```
preload: function () {
  this.load.image('sea', 'assets/sea.png');
  this.load.image('bullet', 'assets/bullet.png');
  this.load.spritesheet('greenEnemy', 'assets/enemy.png', 32, 32);
  this.load.spritesheet('explosion', 'assets/explosion.png', 32, 32);
},
```



Then the actual explosion:

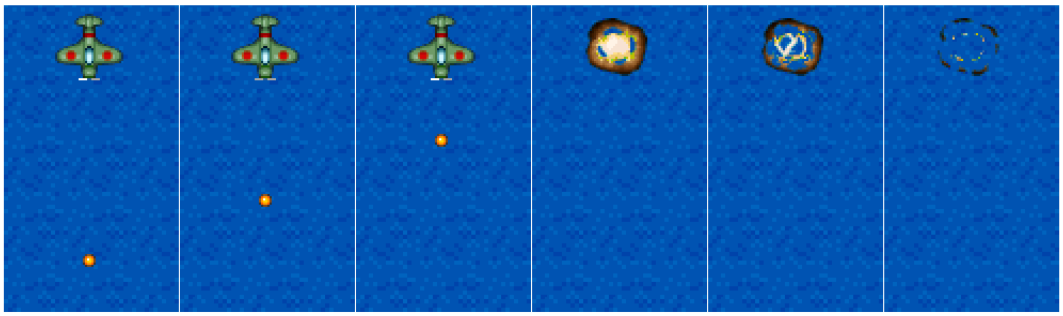
```
enemyHit: function (bullet, enemy) {
  bullet.kill();
  enemy.kill();
  var explosion = this.add.sprite(enemy.x, enemy.y, 'explosion');
  explosion.anchor.setTo(0.5, 0.5);
  explosion.animations.add('boom');
  explosion.play('boom', 15, false, true);
},
```

Here we used a different way to setup animations. This time we used `animations.add()` with only the name of the animation. Lacking the other arguments, the `boom` animation uses all frames of the sprite sheet, runs at 60 fps, and doesn't loop.

We want to tweak the settings of this animation, so we add them to the `explosion.play()` call as additional arguments:

- `15` - set the frames per second
- `false` - don't loop the animation
- `true` - kill the sprite at the end of the animation

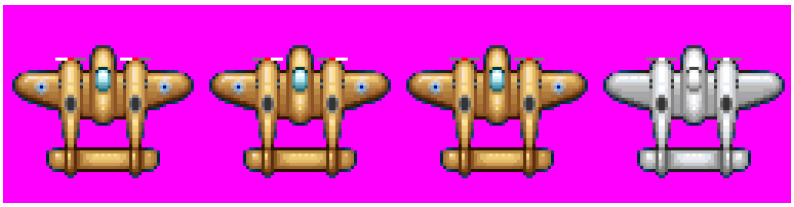
The last argument the most convenient to us; without it we'll need to register an event handler callback to perform the sprite killing, and event handling is a much later lesson. In the meantime, enjoy your improved "shooting down an enemy" animation:



Afternoon 2: Player Actions

Now that we're done with drawing and movement, let's move on to making an object that will represent us in the game. Load the player sprite in the `preload()` function:

```
  preload: function () {
    this.load.image('sea', 'assets/sea.png');
    this.load.image('bullet', 'assets/bullet.png');
    this.load.spritesheet('greenEnemy', 'assets/enemy.png', 32, 32);
    this.load.spritesheet('explosion', 'assets/explosion.png', 32, 32);
    this.load.spritesheet('player', 'assets/player.png', 64, 64);
  },
```



Add the following to the `create()` function before the enemy sprite to add our sprite into the game:

```
    this.sea = this.add.tileSprite(0, 0, 1024, 768, 'sea');

    this.player = this.add.sprite(400, 650, 'player');
    this.player.anchor.setTo(0.5, 0.5);
    this.player.animations.add('fly', [ 0, 1, 2 ], 20, true);
    this.player.play('fly');
    this.game.physics.enable(this.player, Phaser.Physics.ARCADE);

    this.enemy = this.add.sprite(512, 300, 'greenEnemy');
```

Keyboard Movement

Implementing keyboard-based input is straightforward in *Phaser*. Here we begin by using a convenience function which returns the four arrow keys.

```
    this.bullet.anchor.setTo(0.5, 0.5);
    this.physics.enable(this.bullet,  Phaser.Physics.ARCADE);
    this.bullet.body.velocity.y = -400;

    this.cursors = this.input.keyboard.createCursorKeys();
  },
```

Let's also set the player's initial speed speed as a property of the player object on create since we'll be using this value multiple times throughout our program:

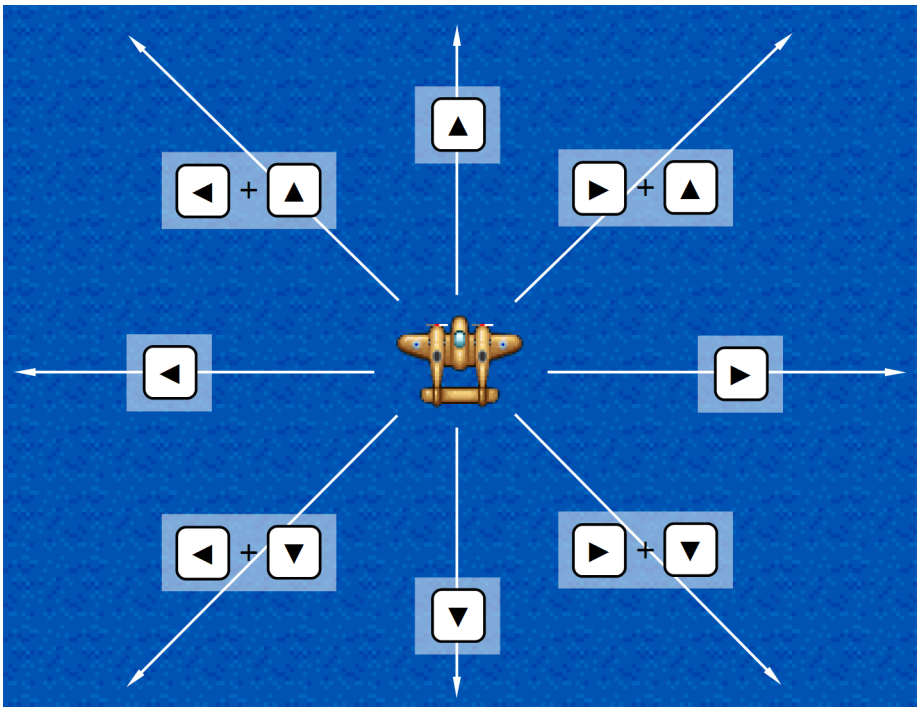
```
    this.player = this.add.sprite(400, 650, 'player');
    this.player.anchor.setTo(0.5, 0.5);
    this.player.animations.add('fly', [ 0, 1, 2 ], 20, true);
    this.player.play('fly');
    this.game.physics.enable(this.player, Phaser.Physics.ARCADE);
    this.player.speed = 300;

    this.enemy = this.add.sprite(512, 300, 'greenEnemy');
```

This will also allow us to have planes with different speeds or "speed up" type of power-ups later.

Once that's done, we can now set the velocity like so:

```
  update: function () {
    this.sea.tilePosition.y += 0.2;
    this.physics.arcade.overlap(
      this.bullet, this.enemy, this.enemyHit, null, this
    );

    this.player.body.velocity.x = 0;
    this.player.body.velocity.y = 0;

    if (this.cursors.left.isDown) {
      this.player.body.velocity.x = -this.player.speed;
    } else if (this.cursors.right.isDown) {
      this.player.body.velocity.x = this.player.speed;
    }

    if (this.cursors.up.isDown) {
      this.player.body.velocity.y = -this.player.speed;
    } else if (this.cursors.down.isDown) {
      this.player.body.velocity.y = this.player.speed;
    }
  },
```

Note that we set the velocity to zero so that the plane stops when the input stops. We also allow the player to input both vertical and horizontal movement at the same time.



Arcade physics also makes it easy to make the edges of the stage act like walls:

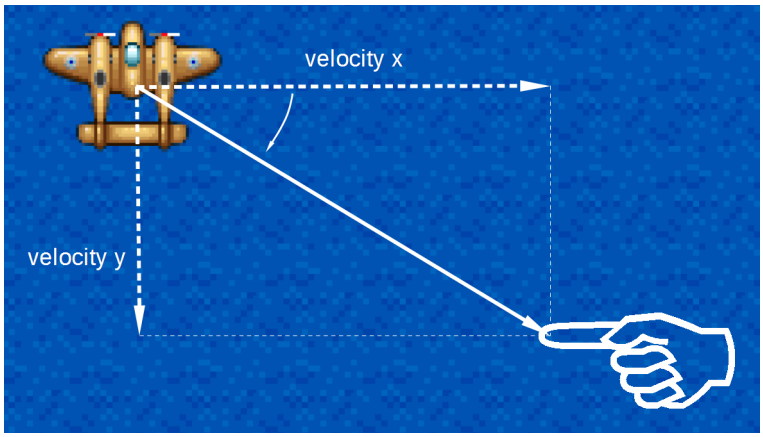
```
    this.game.physics.enable(this.player, Phaser.Physics.ARCADE);
    this.player.speed = 300;
    this.player.body.collideWorldBounds = true;

    this.enemy = this.add.sprite(512, 300, 'greenEnemy');
```

Mouse/Touch Movement

Point-based movement usually requires hand-rolling your mathematical calculations. Fortunately, *Phaser* already has functions which calculates the angle and velocity based on input points.

Here's how simple it is to move an object towards the pointer:

```
      this.player.body.velocity.y = this.player.speed;
    }

    if (this.game.input.activePointer.isDown) {
      this.game.physics.arcade.moveToPointer(this.player, this.player.speed);
    }
  },
```



Based on the object's location and a speed, the Arcade physics function `moveToPointer()` calculates the angle and velocities required to move towards the pointer at the input speed. Calling this function will already modify the x and y velocities of the object, which is exactly what we need in this situation.

This function will not rotate the sprite, though, so if you need to rotate the sprite accordingly, you can use the return value of the function which is the angle of rotation in radians. We shall see an example of this in a later lesson.

Just a word of warning, the movement in a frame may overshoot the target (i.e. move 5 pixels even though the pointer is 2 pixels away) causing your player sprite to tremble instead of staying put. The inaccurate coordinates given by a touch screen may also produce a similar effect. A crude way of getting over these is to stop movement at a certain distance from the pressed point, like so:

```
      this.player.body.velocity.y = this.player.speed;
    }

    if (this.game.input.activePointer.isDown) {
    if (this.game.input.activePointer.isDown &&
        this.game.physics.arcade.distanceToPointer(this.player) > 15) {
      this.game.physics.arcade.moveToPointer(this.player, this.player.speed);
    }
  },
```

If you need more precise input, you may be better off implementing an on-screen directional pad.

Firing Bullets

Let's remove our old bullet code and add new code for creating bullets on the fly.

```
  create: function () {
    ...
    this.bullet = this.add.sprite(512, 400, 'bullet');
    this.bullet.anchor.setTo(0.5, 0.5);
    this.physics.enable(this.bullet, Phaser.Physics.ARCADE);
    this.bullet.body.velocity.y = -400;
    this.bullets = [];
```

We set our fire button to Z or tapping/clicking the screen:

```
  update: function () {
    ...
      this.game.physics.arcade.moveToPointer(this.player, this.player.speed);
    }

    if (this.input.keyboard.isDown(Phaser.Keyboard.Z) ||
        this.input.activePointer.isDown) {
      this.fire();
    }
  },
```

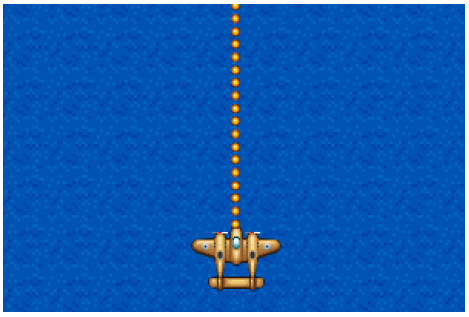
Then we create a new function that will fire a slightly faster bullet just above the nose of player's sprite:

```
82   fire: function() {
83     var bullet = this.add.sprite(this.player.x, this.player.y - 20, 'bullet');
84     bullet.anchor.setTo(0.5, 0.5);
85     this.physics.enable(bullet, Phaser.Physics.ARCADE);
86     bullet.body.velocity.y = -500;
87     this.bullets.push(bullet);
88   },
```

And finally we modify our collision detection code to iterate over the bullets:

```
  update: function () {
    this.sea.tilePosition.y += 0.2;
    this.physics.arcade.overlap(
      this.bullets, this.enemy, this.enemyHit, null, this
    );
    for (var i = 0; i < this.bullets.length; i++) {
      this.physics.arcade.overlap(
        this.bullets[i], this.enemy, this.enemyHit, null, this
      );
    }
```

Fire Rate



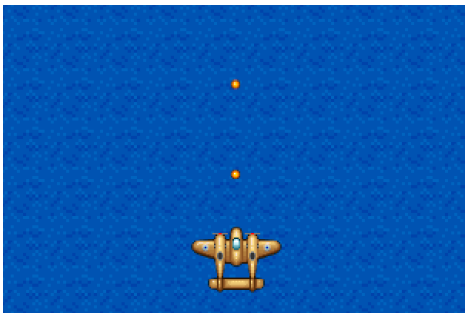
One obvious problem that you'll see as you test this new firing code is that the bullets come out at a very high rate. We can throttle this by storing a time value specifying the earliest time when the next bullet can be fired.

Add the variable `nextShotAt` and `shotDelay` (set to 100 milliseconds) to the `create()` function:

```
    this.bullets = [];
    this.nextShotAt = 0;
    this.shotDelay = 100;
```

Then modify the `fire()` function to check and eventually set the `nextShotAt` variable:

```
fire: function() {
  if (this.nextShotAt > this.time.now) {
    return;
  }

  this.nextShotAt = this.time.now + this.shotDelay;

  var bullet = this.add.sprite(this.player.x, this.player.y - 20, 'bullet');
  bullet.anchor.setTo(0.5, 0.5);
  this.physics.enable(bullet, Phaser.Physics.ARCADE);
  bullet.body.velocity.y = -500;
  this.bullets.push(bullet);
},
```



fire rate now down to 100 milliseconds per shot

How To Play message

We don't have time to code a help screen, so let's just flash the "how to play" instructions in the first 10 seconds of every session.

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

```
this.instructions = this.add.text( 510, 600,
  'Use Arrow Keys to Move, Press Z to Fire\n' +
  'Tapping/clicking does both',
  { font: '20px monospace', fill: '#fff', align: 'center' }
);
this.instructions.anchor.setTo(0.5, 0.5);
this.instExpire = this.time.now + 10000;
```

And the end of `update()` to make the text disappear after the time has elapsed:

```
if (this.instructions.exists && this.time.now > this.instExpire) {
  this.instructions.destroy();
}
```

Other problems

If you haven't noticed it yet, the *other* problem with our current bullet generation approach is it's essentially a memory leak. In the next chapter, we'll discuss one way of limiting the resources that our game will use.

Afternoon 3: Object Groups

Instead of creating objects on the fly, we can create *Groups* where we can use and re-use sprites over and over again.

Convert Bullets to Sprite Group

Bullets are best use case for groups in our game; they're constantly being generated and removed from play. Having a pool of available bullets will save our game time and memory.

Let's begin by switching out our array with a sprite group. The comments below explain our new code.

```
  create: function () {
    ...
    this.bullets = [];
    // Add an empty sprite group into our game
    this.bulletPool = this.add.group();

    // Enable physics to the whole sprite group
    this.bulletPool.enableBody = true;
    this.bulletPool.physicsBodyType = Phaser.Physics.ARCADE;

    // Add 100 'bullet' sprites in the group.
    // By default this uses the first frame of the sprite sheet and
    //   sets the initial state as non-existing (i.e. killed/dead)
    this.bulletPool.createMultiple(100, 'bullet');

    // Sets anchors of all sprites
    this.bulletPool.setAll('anchor.x', 0.5);
    this.bulletPool.setAll('anchor.y', 0.5);

    // Automatically kill the bullet sprites when they go out of bounds
    this.bulletPool.setAll('outOfBoundsKill', true);
    this.bulletPool.setAll('checkWorldBounds', true);

    this.nextShotAt = 0;
```

Let's move on to the `fire()` function:

```
  fire: function() {
    if (this.nextShotAt > this.time.now) {
      return;
    }

    if (this.bulletPool.countDead() === 0) {
      return;
    }

    this.nextShotAt = this.time.now + this.shotDelay;

    var bullet = this.add.sprite(this.player.x, this.player.y - 20, 'bullet');
    bullet.anchor.setTo(0.5, 0.5);
    this.physics.enable(bullet, Phaser.Physics.ARCADE);
    bullet.body.velocity.y = -500;
    this.bullets.push(bullet);

    // Find the first dead bullet in the pool
    var bullet = this.bulletPool.getFirstExists(false);

    // Reset (revive) the sprite and place it in a new location
    bullet.reset(this.player.x, this.player.y - 20);

    bullet.body.velocity.y = -500;
  },
```

Here we replaced creating bullets on the fly with reviving dead bullets in our pool.

Update collision detection

Switching from array to group means we need to modify our collision checking code. Good news is that `overlap()` supports Group to Sprite collision checking.

```
  update: function () {
    this.sea.tilePosition.y += 0.2;
    for (var i = 0; i < this.bullets.length; i++) {
      this.physics.arcade.overlap(
        this.bullets[i], this.enemy, this.enemyHit, null, this
      );
    }
    this.physics.arcade.overlap(
      this.bulletPool, this.enemy, this.enemyHit, null, this
    );
```

Bad news is that there seems to be a bug when comparing “Groups to Sprites” (see if you can notice it) but not in “Sprite to Groups” or “Group to Groups”. This shouldn’t be a problem since we’re only doing the latter two.

Enemy Sprite Group

Our game would be boring if we only had one enemy. Let's make a sprite group so that we can generate a bunch more enemies so that they can start giving us a challenge:

```
    this.enemy = this.add.sprite(512, 300, 'greenEnemy');
    this.enemy.anchor.setTo(0.5, 0.5);
    this.enemy.animations.add('fly', [ 0, 1, 2 ], 20, true);
    this.enemy.play('fly');
    this.physics.enable(this.enemy, Phaser.Physics.ARCADE);
    this.enemyPool = this.add.group();
    this.enemyPool.enableBody = true;
    this.enemyPool.physicsBodyType = Phaser.Physics.ARCADE;
    this.enemyPool.createMultiple(50, 'greenEnemy');
    this.enemyPool.setAll('anchor.x', 0.5);
    this.enemyPool.setAll('anchor.y', 0.5);
    this.enemyPool.setAll('outOfBoundsKill', true);
    this.enemyPool.setAll('checkWorldBounds', true);

    // Set the animation for each sprite
    this.enemyPool.forEach(function (enemy) {
      enemy.animations.add('fly', [ 0, 1, 2 ], 20, true);
    });

    this.nextEnemyAt = 0;
    this.enemyDelay = 1000;
```

And again, modifying the collision code become Group to Group:

```
    this.physics.arcade.overlap(
      this.bulletPool, this.enemy, this.enemyHit, null, this
      this.bulletPool, this.enemyPool, this.enemyHit, null, this
    );
```

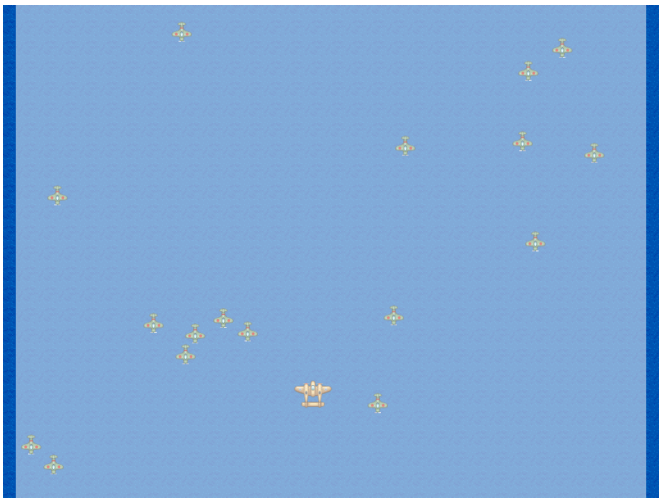
Randomize Enemy Spawn

Many games have enemies show up at scripted positions. We don't have time for that so we'll just randomize the spawning locations.

Add this to the `update()` function:

```
update: function () {
  this.sea.tilePosition.y += 0.2;
  this.physics.arcade.overlap(
    this.bulletPool, this.enemyPool, this.enemyHit, null, this
  );

  if (this.nextEnemyAt < this.time.now && this.enemyPool.countDead() > 0) {
    this.nextEnemyAt = this.time.now + this.enemyDelay;
    var enemy = this.enemyPool.getFirstExists(false);
    // spawn at a random location top of the screen
    enemy.reset(this.rnd.integerInRange(20, 1004), 0);
    // also randomize the speed
    enemy.body.velocity.y = this.rnd.integerInRange(30, 60);
    enemy.play('fly');
  }

  this.player.body.velocity.x = 0;
  this.player.body.velocity.y = 0;
```

Like our `bulletPool`, we also store the next time an enemy should spawn.



enemy spawn area and movement range in white

Note that we did not use `Math.random()` to set the random enemy spawn location and speed but instead used the built-in randomizing functions. Either way is fine, but we chose the built in random number generator because it has some additional features that may be useful later (e.g. seeds).

Player Death

Let's further increase the challenge by allowing our plane to blow up.

Let's first add the collision detection code:

```
  update: function () {
    this.sea.tilePosition.y += 0.2;
    this.physics.arcade.overlap(
      this.bulletPool, this.enemyPool, this.enemyHit, null, this
    );

    this.physics.arcade.overlap(
      this.player, this.enemyPool, this.playerHit, null, this
    );

    if (this.nextEnemyAt < this.time.now && this.enemyPool.countDead() > 0) {
```

Then the callback:

```
140   playerHit: function (player, enemy) {
141     enemy.kill();
142     var explosion = this.add.sprite(player.x, player.y, 'explosion');
143     explosion.anchor.setTo(0.5, 0.5);
144     explosion.animations.add('boom');
145     explosion.play('boom', 15, false, true);
146     player.kill();
147   },
```

You might notice that even though the plane blows up when we crash to another plane, we can still fire our guns. Let's fix that by checking the `alive` flag:

```
  fire: function() {
    if (this.nextShotAt > this.time.now) {
    if (!this.player.alive || this.nextShotAt > this.time.now) {
      return;
    }

    if (this.bullets.countDead() == 0) {
      return;
    }
```

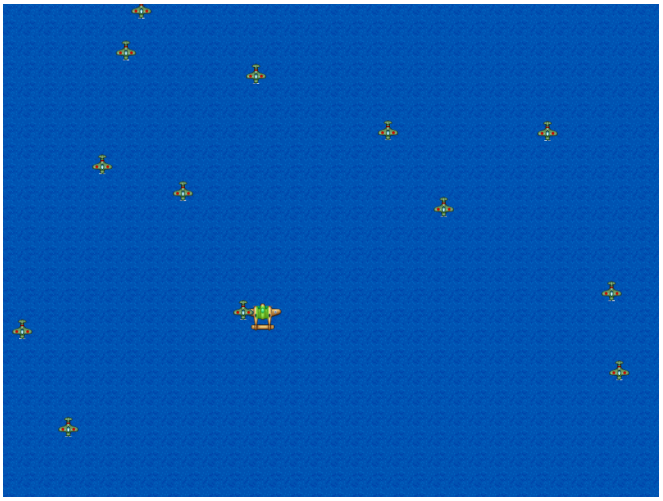
Another possible issue is that our hitbox is too big because of our sprite. Let's lower our hitbox accordingly:

```
    this.game.physics.enable(this.player, Phaser.Physics.ARCADE);
    this.player.body.collideWorldBounds = true;
    // 20 x 20 pixel hitbox, centered a little bit higher than the center
    this.player.body.setSize(20, 20, 0, -5);
```

This hitbox is pretty small, but it's still on par with other shoot em ups (some "bullet hell" type games even have a 1 pixel hitbox). Feel free to increase this if you want a challenge.

Use the debug body function if you need to see your sprite's actual hitbox size.

```
render: function() {
  this.game.debug.body(this.player);
}
```



smaller hitbox, but still fair gameplay-wise

Convert Explosions to Sprite Group

Our explosions are also a possible memory leak. Let's fix that and also do a bit of refactoring in the process.

Put this on the `create()` after all of the other sprites:

```
    this.shotDelay = 100;

    this.explosionPool = this.add.group();
    this.explosionPool.enableBody = true;
    this.explosionPool.physicsBodyType = Phaser.Physics.ARCADE;
    this.explosionPool.createMultiple(100, 'explosion');
    this.explosionPool.setAll('anchor.x', 0.5);
    this.explosionPool.setAll('anchor.y', 0.5);
    this.explosionPool.forEach(function (explosion) {
      explosion.animations.add('boom');
    });

    this.cursors = this.input.keyboard.createCursorKeys();
```

Then create a new function:

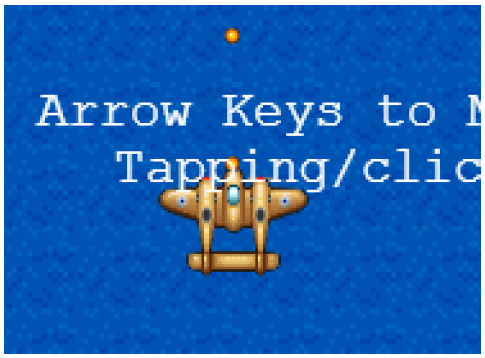
```
161   explode: function (sprite) {
162     if (this.explosionPool.countDead() === 0) {
163       return;
164     }
165     var explosion = this.explosionPool.getFirstExists(false);
166     explosion.reset(sprite.x, sprite.y);
167     explosion.play('boom', 15, false, true);
168     // add the original sprite's velocity to the explosion
169     explosion.body.velocity.x = sprite.body.velocity.x;
170     explosion.body.velocity.y = sprite.body.velocity.y;
171   },
```

And refactor the collision callbacks:

```
  enemyHit: function (bullet, enemy) {
    bullet.kill();
    this.explode(enemy);
    enemy.kill();
    var explosion = this.add.sprite(enemy.x, enemy.y, 'explosion');
    explosion.anchor.setTo(0.5, 0.5);
    explosion.animations.add('boom');
    explosion.play('boom', 15, false, true);
  },

  playerHit: function (player, enemy) {
    this.explode(enemy);
    enemy.kill();
    var explosion = this.add.sprite(player.x, player.y, 'explosion');
    explosion.anchor.setTo(0.5, 0.5);
    explosion.animations.add('boom');
    explosion.play('boom', 15, false, true);
    this.explode(player);
    player.kill();
  },
```

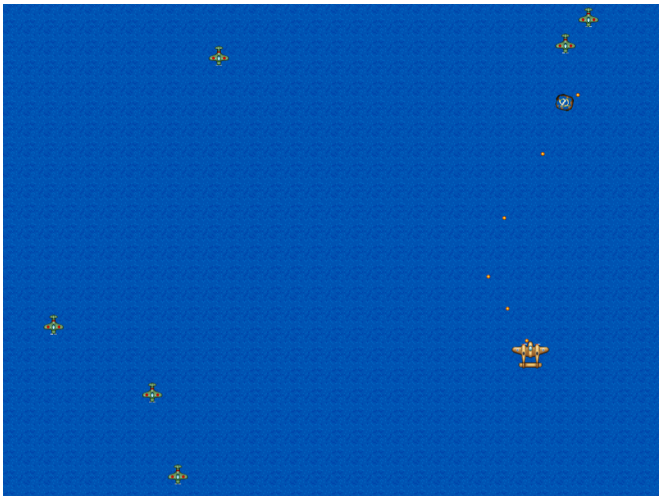
Sprite Ordering

We mentioned before that the ordering of sprites is determined by the time they are added into our game i.e. the first objects (sprites, text, etc) added are at the bottom while the later objects are at the top.



This is done through sprite groups: all objects (sprites, text, and even groups - groups can contain other groups) are added to the game's `World` by default, a special group in our game. Display order is then determined by iterating over the members of the `World`.

For example, the order of the contents of `World` in the following scene is:



- The `sea` tile sprite is at the bottom.
- The `player` sprite is next.
- The `greenEnemy` sprite group is on the next level. Only a few sprites from this group are visible (the rest are still dead).
- Next is the `bullet` sprite group. Same as the enemy group, only a few sprites from this group are visible.
- Next is the `explosion` sprite group.
- At the top is the instructions text. It's not visible anymore at this point in the game.

(`World` is also contained in the `Stage` but we won't be using the `Stage` directly so we won't cover it.)

Refactoring

Our `create()` and `update()` functions are getting bigger and it will be worse as we proceed with the workshop. We'll *refactor* them by splitting these large functions into smaller functions.

Here are the overhauled functions and the extracted ones:

```
15    create: function () {
16      this.sea = this.add.tileSprite(0, 0, 1024, 768, 'sea');
17
18      this.setupPlayer();
19      this.setupEnemies();
20      this.setupBullets();
21      this.setupExplosions();
22      this.setupText();
23
24      this.cursors = this.input.keyboard.createCursorKeys();
25    },
26
27    update: function () {
28      this.sea.tilePosition.y += 0.2;
29
30      this.checkCollisions();
31      this.spawnEnemies();
32      this.processPlayerInput();
33      this.processDelayedEffects();
34    },
35
36    // create()-related functions
37
38    setupPlayer: function () {
39      this.player = this.add.sprite(400, 650, 'player');
40      this.player.anchor.setTo(0.5, 0.5);
41      this.player.animations.add('fly', [ 0, 1, 2 ], 20, true);
42      this.player.play('fly');
43      this.game.physics.enable(this.player, Phaser.Physics.ARCADE);
44      this.player.speed = 300;
45      this.player.body.collideWorldBounds = true;
46      // 20 x 20 pixel hitbox, centered a little bit higher than the center
47      this.player.body.setSize(20, 20, 0, -5);
48    },
49
50    setupEnemies: function () {
51      this.enemyPool = this.add.group();
52      this.enemyPool.enableBody = true;
53      this.enemyPool.physicsBodyType = Phaser.Physics.ARCADE;
54      this.enemyPool.createMultiple(50, 'greenEnemy');
55      this.enemyPool.setAll('anchor.x', 0.5);
56      this.enemyPool.setAll('anchor.y', 0.5);
```

```
57      this.enemyPool.setAll('outOfBoundsKill', true);
58      this.enemyPool.setAll('checkWorldBounds', true);
59
60      // Set the animation for each sprite
61      this.enemyPool.forEach(function (enemy) {
62        enemy.animations.add('fly', [ 0, 1, 2 ], 20, true);
63      });
64
65      this.nextEnemyAt = 0;
66      this.enemyDelay = 1000;
67    },
68
69    setupBullets: function () {
70      // Add an empty sprite group into our game
71      this.bulletPool = this.add.group();
72
73      // Enable physics to the whole sprite group
74      this.bulletPool.enableBody = true;
75      this.bulletPool.physicsBodyType = Phaser.Physics.ARCADE;
76
77      // Add 100 'bullet' sprites in the group.
78      // By default this uses the first frame of the sprite sheet
79      //   and sets the initial state as non-existing (i.e. killed)
80      this.bulletPool.createMultiple(100, 'bullet');
81
82      // Sets anchors of all sprites
83      this.bulletPool.setAll('anchor.x', 0.5);
84      this.bulletPool.setAll('anchor.y', 0.5);
85
86      // Automatically kill the bullet sprites when they go out of bounds
87      this.bulletPool.setAll('outOfBoundsKill', true);
88      this.bulletPool.setAll('checkWorldBounds', true);
89
90      this.nextShotAt = 0;
91      this.shotDelay = 100;
92    },
93
94    setupExplosions: function () {
95      this.explosionPool = this.add.group();
96      this.explosionPool.enableBody = true;
97      this.explosionPool.physicsBodyType = Phaser.Physics.ARCADE;
98      this.explosionPool.createMultiple(100, 'explosion');
99      this.explosionPool.setAll('anchor.x', 0.5);
100     this.explosionPool.setAll('anchor.y', 0.5);
101     this.explosionPool.forEach(function (explosion) {
102       explosion.animations.add('boom');
103     });
104   },
105
```

```
106   setupText: function () {
107     this.instructions = this.add.text( 510, 600,
108       'Use Arrow Keys to Move, Press Z to Fire\n' +
109       'Tapping/clicking does both',
110       { font: '20px monospace', fill: '#fff', align: 'center' }
111     );
112     this.instructions.anchor.setTo(0.5, 0.5);
113     this.instExpire = this.time.now + 10000;
114   },
115
116   // update()-related functions
117
118   checkCollisions: function () {
119     this.physics.arcade.overlap(
120       this.bulletPool, this.enemyPool, this.enemyHit, null, this
121     );
122
123     this.physics.arcade.overlap(
124       this.player, this.enemyPool, this.playerHit, null, this
125     );
126   },
127
128   spawnEnemies: function () {
129     if (this.nextEnemyAt < this.time.now && this.enemyPool.countDead() > 0) {
130       this.nextEnemyAt = this.time.now + this.enemyDelay;
131       var enemy = this.enemyPool.getFirstExists(false);
132       enemy.reset(this.rnd.integerInRange(20, 1004), 0);
133       enemy.body.velocity.y = this.rnd.integerInRange(30, 60);
134       enemy.play('fly');
135     }
136   },
137
138   processPlayerInput: function () {
139     this.player.body.velocity.x = 0;
140     this.player.body.velocity.y = 0;
141
142     if (this.cursors.left.isDown) {
143       this.player.body.velocity.x = -this.player.speed;
144     } else if (this.cursors.right.isDown) {
145       this.player.body.velocity.x = this.player.speed;
146     }
147
148     if (this.cursors.up.isDown) {
149       this.player.body.velocity.y = -this.player.speed;
150     } else if (this.cursors.down.isDown) {
151       this.player.body.velocity.y = this.player.speed;
152     }
153
154     if (this.game.input.activePointer.isDown &&
```

```
155         this.game.physics.arcade.distanceToPointer(this.player) > 15) {
156       this.game.physics.arcade.moveToPointer(this.player, this.player.speed);
157     }
158
159     if (this.input.keyboard.isDown(Phaser.Keyboard.Z) ||
160         this.input.activePointer.isDown) {
161       this.fire();
162     }
163   },
164
165   processDelayedEffects: function () {
166     if (this.instructions.exists && this.time.now > this.instExpire) {
167       this.instructions.destroy();
168     }
169   },
```

Afternoon 4: Health, Score, and Win/Lose Conditions

Our game looks more like a real game now, but there's still a lot of room for improvement.

Enemy Health

Phaser makes modifying enemy toughness easy for us because it supports health and damage calculation.

Before we could implement health to our enemies, let's first add a `hit` animation (finally using the last frame of the sprite sheet):

```
  setupEnemies: function () {
...
    this.enemyPool.setAll('checkWorldBounds', true)

    // Set the animation for each sprite
    this.enemyPool.forEach(function (enemy) {
      enemy.animations.add('fly', [ 0, 1, 2 ], 20, true);
      enemy.animations.add('hit', [ 3, 1, 3, 2 ], 20, false);
      enemy.events.onAnimationComplete.add( function (e) {
        e.play('fly');
      }, this);
    });

    this.nextEnemyAt = 0;
```

The new animation is a very short non-looping blinking animation which goes back to the original `fly` animation once it ends.

Let's now add the health. Sprites in Phaser have a default health value of 1 but we can override it anytime:

```
  setupEnemies: function () {
...
    this.nextEnemyAt = 0;
    this.enemyDelay = 1000;
    this.enemyInitialHealth = 2;
  },

...

  spawnEnemies: function () {
    if (this.nextEnemyAt < this.time.now && this.enemyPool.countDead() > 0) {
      this.nextEnemyAt = this.time.now + this.enemyDelay;
```

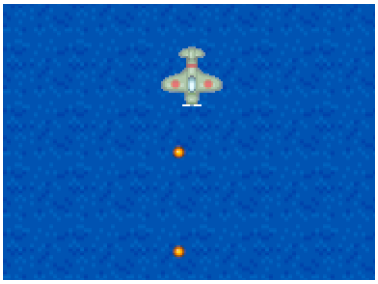
```
      var enemy = this.enemyPool.getFirstExists(false);
      enemy.reset(this.rnd.integerInRange(20, 1004), 0);
      enemy.reset(this.rnd.integerInRange(20, 1004), 0, this.enemyInitialHealth);
      enemy.body.velocity.y = this.rnd.integerInRange(30, 60);
      enemy.play('fly');
    }
  },
```

We could have used `enemy.health = this.enemyInitialHealth` but `reset()` already has an optional parameter that does the same.

And finally, let's create a new function to process the damage, centralizing the killing and explosion animation:

```
  enemyHit: function (bullet, enemy) {
    bullet.kill();
    this.explode(enemy);
    enemy.kill();
    this.damageEnemy(enemy, 1);
  },

  playerHit: function (player, enemy) {
    this.explode(enemy);
    enemy.kill();
    // crashing into an enemy only deals 5 damage
    this.damageEnemy(enemy, 5);
    this.explode(player);
    player.kill();
  },


  damageEnemy: function (enemy, damage) {
    enemy.damage(damage);
    if (enemy.alive) {
      enemy.play('hit');
    } else {
      this.explode(enemy);
    }
  },
```

Using `damage()` automatically `kill()`s the sprite once its health is reduced to zero.



Player Score

We don't need to explain how important it is to display the player's current score on the screen. Everyone just knows it.

First set the score rewarded on kill:

```
  setupEnemies: function () {
...
    this.enemyPool.setAll('outOfBoundsKill', true);
    this.enemyPool.setAll('checkWorldBounds', true);
    this.enemyPool.setAll('reward', 100, false, false, 0, true);

    // Set the animation for each sprite
    this.enemyPool.forEach(function (enemy) {
```

We used the full form of the `setAll()` function. The last four parameters are default, and we only change the last parameter to `true` which forces the function to set the `reward` property even though it isn't there.

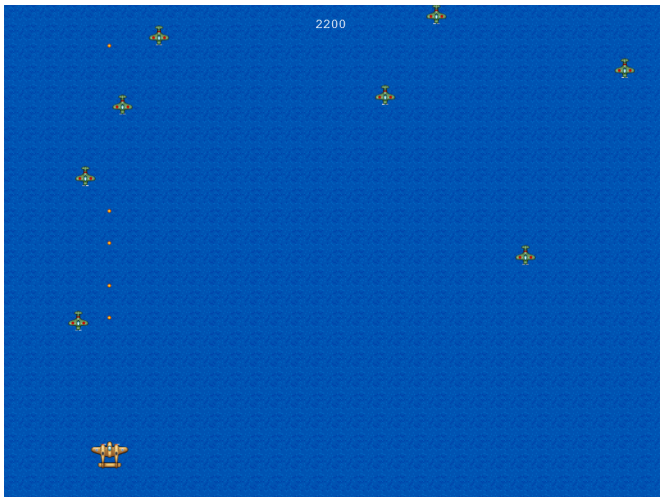
Next step is to add the `setupText()` code for displaying the starting score:

```
  setupText: function () {
    this.instructions = this.add.text( 510, 600,
      'Use Arrow Keys to Move, Press Z to Fire\n' +
      'Tapping/clicking does both',
      { font: '20px monospace', fill: '#fff', align: 'center' }
    );
    this.instructions.anchor.setTo(0.5, 0.5);
    this.instExpire = this.time.now + 10000;

    this.score = 0;
    this.scoreText = this.add.text(
      510, 30, '' + this.score,
      { font: '20px monospace', fill: '#fff', align: 'center' }
    );
    this.scoreText.anchor.setTo(0.5, 0.5);
  },
```

And then let's add it to our enemy damage/death handler:

```
  damageEnemy: function (enemy, damage) {
    enemy.damage(damage);
    if (enemy.alive) {
      enemy.play('hit');
    } else {
      this.explode(enemy);
      this.addToScore(enemy.reward);
    }
  },
```

```
209   addToScore: function (score) {
210     this.score += score;
211     this.scoreText.text = this.score;
212   },
```



Player Lives

Sudden death games are cool, but may be “unfun” for others. Most people are used to having lives and retries in their games.

First, let’s create a new sprite group representing our lives at the top right corner of the screen.

```
  create: function () {
    this.sea = this.add.tileSprite(0, 0, 1024, 768, 'sea')

    this.setupPlayer();
    this.setupEnemies();
    this.setupBullets();
    this.setupExplosions();
    this.setupPlayerIcons();
    this.setupText();
```

```
    this.cursors = this.input.keyboard.createCursorKeys();
  },

...

  setupPlayerIcons: function () {
    this.lives = this.add.group();
    for (var i = 0; i < 3; i++) {
      var life = this.lives.create(924 + (30 * i), 30, 'player');
      life.scale.setTo(0.5, 0.5);
      life.anchor.setTo(0.5, 0.5);
    }
  },

  setupText: function () {
```

For the life icons, we just used the player's sprite and scaled it down to half its size by modifying the `scale` property.

With the life tracking done, let's add the blinking ghost animation on player death:

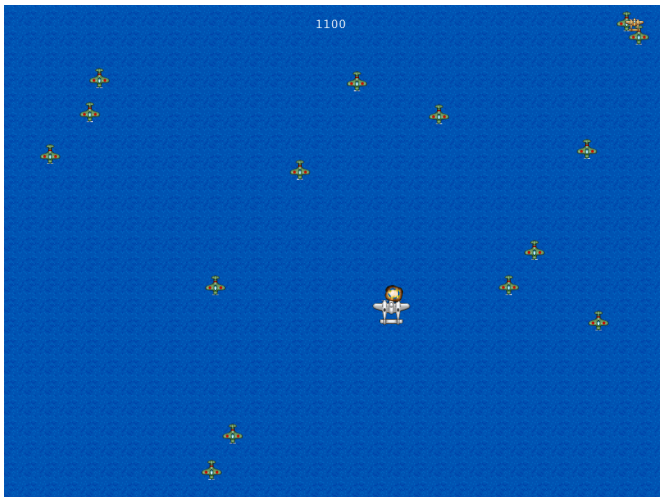
```
    this.player.animations.add('fly', [ 0, 1, 2 ], 20, true);
    this.player.animations.add('ghost', [ 3, 0, 3, 1 ], 20, true);
    this.player.play('fly');
```

Then let's modify `playerHit()` to activate "ghost mode" for 3 seconds and ignore everything around us while we're a ghost:

```
  playerHit: function (player, enemy) {
    // check first if this.ghostUntil is not not undefined or null
    if (this.ghostUntil && this.ghostUntil > this.time.now) {
      return;
    }
    // crashing into an enemy only deals 5 damage
    this.damageEnemy(enemy, 5);
    this.explode(player);
    player.kill();
    var life = this.lives.getFirstAlive();
    if (life) {
      life.kill();
      this.ghostUntil = this.time.now + 3000;
      this.player.play('ghost');
    } else {
      this.explode(player);
      player.kill();
    }
  },
```

And finally, we modify the `processDelayedEffects()` function to check if the ghost mode has already expired:

```
  processDelayedEffects: function () {
    if (this.instructions.exists && this.time.now > this.instExpire) {
      this.instructions.destroy();
    }

    if (this.ghostUntil && this.ghostUntil < this.time.now) {
      this.ghostUntil = null;
      this.player.play('fly');
    }
  },
```



Win/Lose Conditions, Go back to Menu

One of the last things we need to implement is a game ending condition. Currently, our player can die, but there's no explicit message whether the game is over or not. On the other hand, we also don't have a "win" condition.

Let's implement both to wrap up our prototype.

Create a new function to display the end game message:

```
241   displayEnd: function (win) {
242     // you can't win and lose at the same time
243     if (this.endText && this.endText.exists) {
244       return;
245     }
246
247     var msg = win ? 'You Win!!!' : 'Game Over!';
248     this.endText = this.add.text(
249       510, 320, msg,
250       { font: '72px serif', fill: '#fff' }
251     );
252     this.endText.anchor.setTo(0.5, 0);
253
```

```
254     this.showReturn = this.time.now + 2000;
255   },
```

Modify the `playerHit()` function to call the "Game Over!" message:

```
  playerHit: function (player, enemy) {
...
    } else {
      this.explode(player);
      player.kill();
      this.displayEnd(false);
    }
```

Do the same to the `addToScore()` function, but now to destroy all enemies (preventing accidental death and also stopping them from spawning) and display "You Win!!!" message upon reaching 2000 points:

```
  addToScore: function (score) {
    this.score += 100;
    this.scoreText.text = this.score;
    if (this.score >= 2000) {
      this.enemyPool.destroy();
      this.displayEnd(true);
    }
  },
```

Let's also display a "back to main menu" message a few seconds after the game ends. In `processDelayedEffects()`:

```
      this.player.play('fly');
    }

    if (this.showReturn && this.time.now > this.showReturn) {
      this.returnText = this.add.text(
        512, 400,
        'Press Z or Tap Game to go back to Main Menu',
        { font: '16px sans-serif', fill: '#fff'}
      );
      this.returnText.anchor.setTo(0.5, 0.5);
      this.showReturn = false;
    }
  },
```

Since our main menu button is the same action as firing bullets, we can modify `processPlayerInput()` function to allow us to quit the game:

```
    if (this.input.keyboard.isDown(Phaser.Keyboard.Z) ||
        this.input.activePointer.isDown) {
      this.fire();
      if (this.returnText && this.returnText.exists) {
        this.quitGame();
      } else {
        this.fire();
      }
    }
  },
```

Before going back to the main menu, let's destroy all objects in the world to allow us to play over and over again:

```
  quitGame: function (pointer) {

    //  Here you should destroy anything you no longer need.
    //  Stop music, delete sprites, purge caches, free resources, all that good stuff.
    this.sea.destroy();
    this.player.destroy();
    this.enemyPool.destroy();
    this.bulletPool.destroy();
    this.explosionPool.destroy();
    this.instructions.destroy();
    this.scoreText.destroy();
    this.endText.destroy();
    this.returnText.destroy();
    //  Then let's go back to the main menu.
    this.state.start('MainMenu');

  }
```

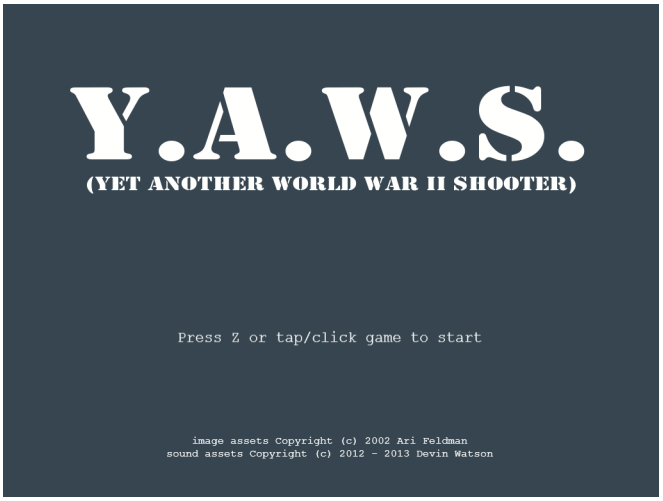
Going back to the main menu will display a black screen with text. This is because we skipped loading the title page image in `preloader.js`. To properly display the main menu, let's temporarily add the preloading in `mainMenu.js`:

```
BasicGame.MainMenu.prototype = {

  preload: function () {
    this.load.image('titlepage', 'assets/titlepage.png');
  },

  create: function () {
```

Enjoy playing your prototype game!

Game Over screen

Win screen



pressing Z returns you to the Main Menu

Afternoon 5: Expanding the Game

Let's flesh out the game by adding an additional enemy, a power-up, a boss battle, and sounds.

Harder Enemy

The green enemy fighters in our current game pose no real threat to our players. To make our game more difficult, our next enemy type will be able to shoot at our player while also being faster and tougher.

Enemy Setup

First load the sprite sheet in the pre-loader:

```
  preload: function () {
    this.load.image('sea', 'assets/sea.png');
    this.load.image('bullet', 'assets/bullet.png');
    this.load.spritesheet('greenEnemy', 'assets/enemy.png', 32, 32);
    this.load.spritesheet('whiteEnemy', 'assets/shooting-enemy.png', 32, 32);
    this.load.spritesheet('explosion', 'assets/explosion.png', 32, 32);
    this.load.spritesheet('player', 'assets/player.png', 64, 64);
  },
```

Then create the group for the sprites (which we will call "shooters" from now on):

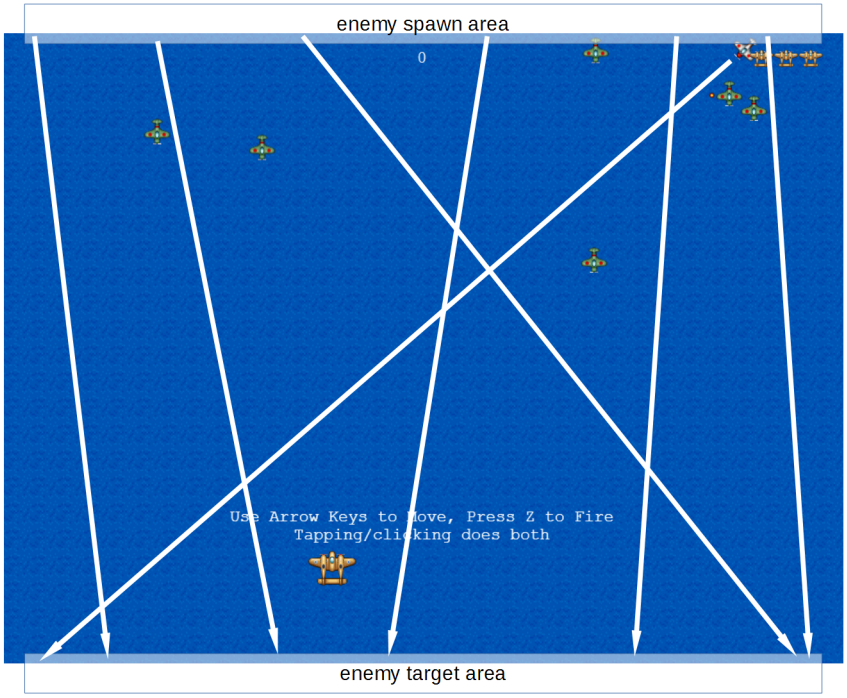
```
  setupEnemies: function () {

...
    this.enemyDelay = 1000;
    this.enemyInitialHealth = 2;

    this.shooterPool = this.add.group();
    this.shooterPool.enableBody = true;
    this.shooterPool.physicsBodyType = Phaser.Physics.ARCADE;
    this.shooterPool.createMultiple(20, 'whiteEnemy');
    this.shooterPool.setAll('anchor.x', 0.5);
    this.shooterPool.setAll('anchor.y', 0.5);
    this.shooterPool.setAll('outOfBoundsKill', true);
    this.shooterPool.setAll('checkWorldBounds', true);
    this.shooterPool.setAll('reward', 400, false, false, 0, true);

    // Set the animation for each sprite
    this.shooterPool.forEach(function (enemy) {
      enemy.animations.add('fly', [ 0, 1, 2 ], 20, true);
      enemy.animations.add('hit', [ 3, 1, 3, 2 ], 20, false);
```

```
    enemy.events.onAnimationComplete.add( function (e) {
      e.play('fly');
    }, this);
  });

  // start spawning 5 seconds into the game
  this.nextShooterAt = this.time.now + 5000;
  this.shooterDelay = 3000;
  this.shooterShotDelay = 2000;
  this.shooterInitialHealth = 5;
},
```

Diagonal Movement

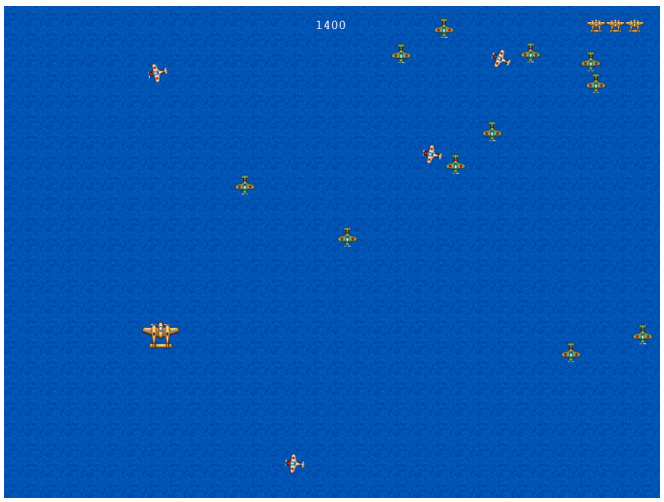
Instead of moving only downwards like the regular enemy, we'll make the shooters move diagonally across the screen. Add the following code into `spawnEnemies()`:

```
spawnEnemies: function () {
  if (this.nextEnemyAt < this.time.now && this.enemyPool.countDead() > 0) {
    this.nextEnemyAt = this.time.now + this.enemyDelay;
    var enemy = this.enemyPool.getFirstExists(false);
    enemy.reset(this.rnd.integerInRange(20, 1004), 0, this.enemyInitialHealth);
    enemy.body.velocity.y = this.rnd.integerInRange(30, 60);
    enemy.play('fly');
  }

  if (this.nextShooterAt < this.time.now && this.shooterPool.countDead() > 0) {
    this.nextShooterAt = this.time.now + this.shooterDelay;
    var shooter = this.shooterPool.getFirstExists(false);

    // spawn at a random location at the top
    shooter.reset(this.rnd.integerInRange(20, 1004), 0,
                  this.shooterInitialHealth);

    // choose a random target location at the bottom
    var target = this.rnd.integerInRange(20, 1004);

    // move to target and rotate the sprite accordingly
    shooter.rotation = this.physics.arcade.moveToXY(
      shooter, target, 768, this.rnd.integerInRange(30, 80)
    ) - Math.PI / 2;

    shooter.play('fly');

    // each shooter has their own shot timer
    shooter.nextShotAt = 0;
  }
},
```



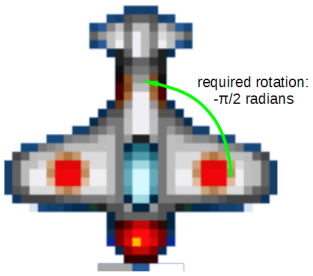
The figure above shows the initial spawn and target areas for the shooters; the arrows show possible flight paths. Here we're using `moveToXY()`, a function similar to `moveToPointer()` which moves the object to a given point in the world.

Both `moveToPointer()` and `moveToXY()` returns the angle towards the target in radians, and we can assign this value to `object.rotation` to rotate our sprite towards the target. But applying the value directly will result in incorrectly oriented shooters:



This is because Phaser assumes that your sprites are oriented to the right. We rotated our sprite counterclockwise `Math.PI / 2` radians (90 degrees) to compensate for the fact that our sprite is oriented downwards.

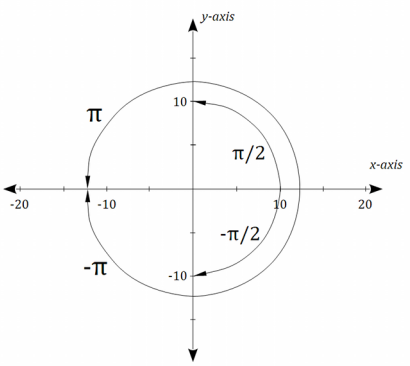
expected sprite



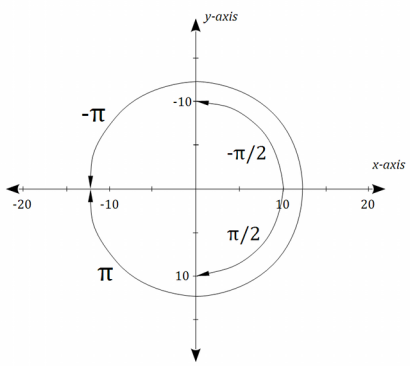
actual sprite

Angles/Rotation in Phaser

Angles in Phaser are same as in Trigonometry, though it might look wrong at first glance for those used to Cartesian coordinates rather than screen coordinates.



angles in Cartesian coordinate system



angles in screen coordinate system

The rotation seems flipped (increasing angles are clockwise rotations rather than counterclockwise) because the `y` values for the two coordinate systems are flipped.

By the way, you can use `object.angle` instead of `object.rotation` if you prefer rotating in degrees rather than radians.

Shooting

Setting up the bullets are pretty much the same as the regular bullets. First the `preload()`:

```
  preload: function () {
    this.load.image('sea', 'assets/sea.png');
    this.load.image('bullet', 'assets/bullet.png');
    this.load.image('enemyBullet', 'assets/enemy-bullet.png');
    this.load.spritesheet('greenEnemy', 'assets/enemy.png', 32, 32);
    this.load.spritesheet('whiteEnemy', 'assets/shooting-enemy.png', 32, 32);
    this.load.spritesheet('explosion', 'assets/explosion.png', 32, 32);
    this.load.spritesheet('player', 'assets/player.png', 64, 64);
  },
```

Then the sprite group at `setupBullets()`:

```
  setupBullets: function () {
    this.enemyBulletPool = this.add.group();
    this.enemyBulletPool.enableBody = true;
    this.enemyBulletPool.physicsBodyType = Phaser.Physics.ARCADE;
    this.enemyBulletPool.createMultiple(100, 'enemyBullet');
    this.enemyBulletPool.setAll('anchor.x', 0.5);
    this.enemyBulletPool.setAll('anchor.y', 0.5);
    this.enemyBulletPool.setAll('outOfBoundsKill', true);
    this.enemyBulletPool.setAll('checkWorldBounds', true);
    this.enemyBulletPool.setAll('reward', 0, false, false, 0, true);

    // Add an empty sprite group into our game
    this.bulletPool = this.add.group();
```

We've already set the shot timer for the individual shooters in the spawning section. All that's left is to create a new function that fires the enemy bullets.

```
  update: function () {
    this.sea.tilePosition.y += 0.2;

    this.checkCollisions();
    this.spawnEnemies();
    this.enemyFire();
    this.processPlayerInput();
    this.processDelayedEffects();
  },
```

And the actual function, iterating over the live shooters in the world:

```
222   enemyFire: function() {
223     this.shooterPool.forEachAlive(function (enemy) {
224       if (this.time.now > enemy.nextShotAt && this.enemyBulletPool.countDead() > 0) {
225         var bullet = this.enemyBulletPool.getFirstExists(false);
226         bullet.reset(enemy.x, enemy.y);
227         this.physics.arcade.moveToObject(bullet, this.player, 150);
228         enemy.nextShotAt = this.time.now + this.shooterShotDelay;
229       }
230     }, this);
231   },
```

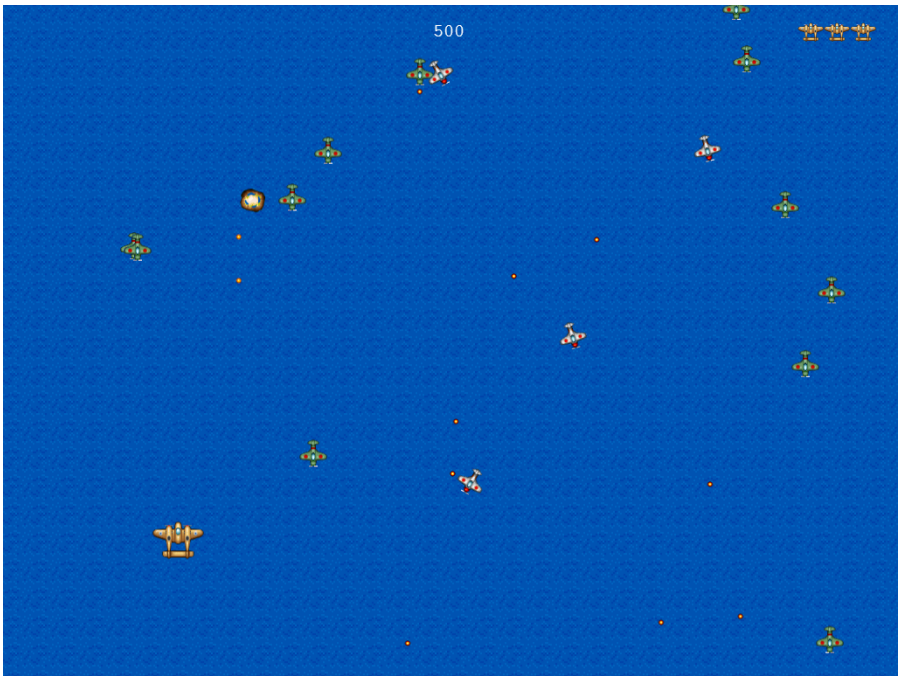
Collision Detection

To wrap things up, let's handle the collisions for the shooters as well as their bullets:

```
  checkCollisions: function () {
    this.physics.arcade.overlap(
      this.bulletPool, this.enemyPool, this.enemyHit, null, this
    );

    this.physics.arcade.overlap(
      this.bulletPool, this.shooterPool, this.enemyHit, null, this
    );

    this.physics.arcade.overlap(
      this.player, this.enemyPool, this.playerHit, null, this
    );

    this.physics.arcade.overlap(
      this.player, this.shooterPool, this.playerHit, null, this
    );

    this.physics.arcade.overlap(
      this.player, this.enemyBulletPool, this.playerHit, null, this
    );
  },
```

We'll also destroy the shooters and bullets in `addToScore()` upon winning:

```
  addToScore: function (score) {
    this.score += score;
    this.scoreText.text = this.score;
    if (this.score >= 2000) {
      this.enemyPool.destroy();
      this.shooterPool.destroy();
      this.enemyBulletPool.destroy();
      this.displayEnd(true);
    }
  },
```



Power-up

Our regular bullet stream is now a lot weaker with the introduction of the shooters. To counter this, let's add a power-up that our players can pickup to get a spread shot.

Pre-loading the asset:

```
preload: function () {
  this.load.image('sea', 'assets/sea.png');
  this.load.image('bullet', 'assets/bullet.png');
  this.load.image('enemyBullet', 'assets/enemy-bullet.png');
  this.load.image('powerup1', 'assets/powerup1.png');
  this.load.spritesheet('whiteEnemy', 'assets/shooting-enemy.png', 32, 32);
```

Then creating the sprite group:

```
  setupPlayerIcons: function () {
    this.powerUpPool = this.add.group();
    this.powerUpPool.enableBody = true;
    this.powerUpPool.physicsBodyType = Phaser.Physics.ARCADE;
    this.powerUpPool.createMultiple(5, 'powerup1');
    this.powerUpPool.setAll('anchor.x', 0.5);
    this.powerUpPool.setAll('anchor.y', 0.5);
    this.powerUpPool.setAll('outOfBoundsKill', true);
    this.powerUpPool.setAll('checkWorldBounds', true);
    this.powerUpPool.setAll('reward', 100, false, false, 0, true);

    this.lives = this.add.group();
    for (var i = 0; i < 3; i++) {
```

We also add the possibility of spawning a power-up when an enemy dies, 30% chance for regular enemies and 50% for shooters:

```
  setupEnemies: function () {
...
    this.enemyPool.setAll('checkWorldBounds', true);
    this.enemyPool.setAll('reward', 100, false, false, 0, true);
    this.enemyPool.setAll('dropRate', 0.3, false, false, 0, true);

...
    this.shooterPool.setAll('checkWorldBounds', true);
    this.shooterPool.setAll('reward', 400, false, false, 0, true);
    this.shooterPool.setAll('dropRate', 0.5, false, false, 0, true);
```

Add the call in `damageEnemy()` to a function that spawns power-ups:

```
  damageEnemy: function (enemy, damage) {
    enemy.damage(damage);
    if (enemy.alive) {
      enemy.play('hit');
    } else {
      this.explode(enemy);
      this.spawnPowerUp(enemy);
      this.addToScore(enemy.reward);
    }
  },
```

Here's the new function for spawning power-ups:

```
387   spawnPowerUp: function (enemy) {
388     if (this.powerUpPool.countDead() === 0 || this.weaponLevel === 5) {
389       return;
390     }
391
392     if (this.rnd.frac() < enemy.dropRate) {
393       var powerUp = this.powerUpPool.getFirstExists(false);
394       powerUp.reset(enemy.x, enemy.y);
395       powerUp.body.velocity.y = 100;
396     }
397   },
```

Weapon levels

You might have noticed the `this.weaponLevel == 5` in the last code snippet. Our weapon strength will have up to 5 levels, each incremented by picking up a power-up.

Setting the initial value to zero:

```
  setupPlayer: function () {
...
    this.player.body.setSize(20, 20, 0, -5);
    this.weaponLevel = 0;
  },
```

Adding a collision handler:

```
  checkCollisions: function () {
    this.physics.arcade.overlap(
      this.bulletPool, this.enemyPool, this.enemyHit, null, this
    );
...

    this.physics.arcade.overlap(
      this.player, this.powerUpPool, this.playerPowerUp, null, this
    );
  },
```

And a new function for incrementing the weapon level:

```
364   playerPowerUp: function (player, powerUp) {
365     this.addToScore(powerUp.reward);
366     powerUp.kill();
367     if (this.weaponLevel < 5) {
368       this.weaponLevel++;
369     }
370   },
```

A common theme in shoot 'em ups is that your weapon power resets when you die. Let's add that into our code:

```
  playerHit: function (player, enemy) {
...
    if (life) {
      life.kill();
      this.weaponLevel = 0;
      this.ghostUntil = this.time.now + 3000;
```

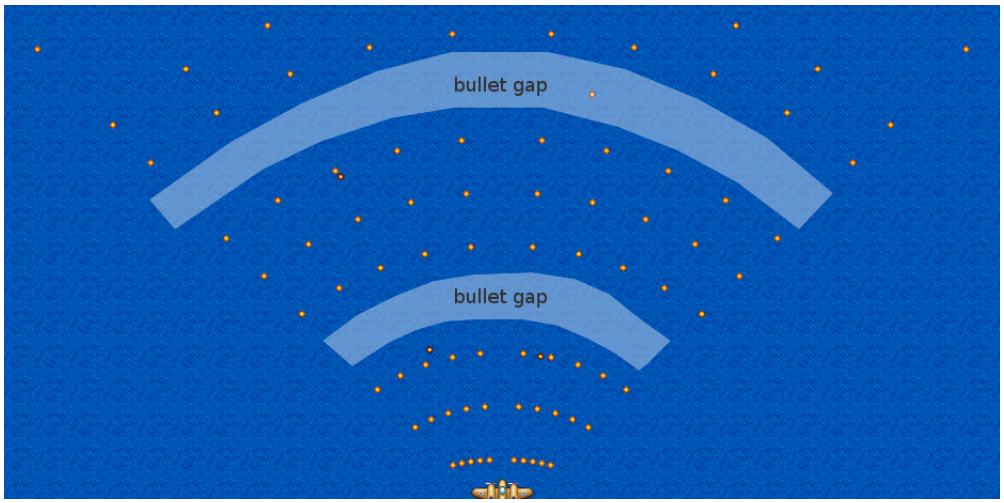
And finally, the code for implementing the spread shot:

```
  fire: function() {
    if (!this.player.alive || this.nextShotAt > this.time.now) {
      return;
    }

    if (this.bulletPool.countDead() === 0) {
      return;
    }

    this.nextShotAt = this.time.now + this.shotDelay;

    // Find the first dead bullet in the pool
    var bullet = this.bulletPool.getFirstExists(false);

    // Reset (revive) the sprite and place it in a new location
    bullet.reset(this.player.x, this.player.y - 20);

    bullet.body.velocity.y = -500;

    var bullet;
    if (this.weaponLevel === 0) {
      if (this.bulletPool.countDead() === 0) {
        return;
      }
      bullet = this.bulletPool.getFirstExists(false);
      bullet.reset(this.player.x, this.player.y - 20);
      bullet.body.velocity.y = -500;
    } else {
      if (this.bulletPool.countDead() < this.weaponLevel * 2) {
        return;
      }
      for (var i = 0; i < this.weaponLevel; i++) {
        bullet = this.bulletPool.getFirstExists(false);
        // spawn left bullet slightly left off center
        bullet.reset(this.player.x - (10 + i * 6), this.player.y - 20);
        // the left bullets spread from -95 degrees to -135 degrees
        this.physics.arcade.velocityFromAngle(
          -95 - i * 10, 500, bullet.body.velocity
        );
```

```
        bullet = this.bulletPool.getFirstExists(false);
        // spawn right bullet slightly right off center
        bullet.reset(this.player.x + (10 + i * 6), this.player.y - 20);
        // the right bullets spread from -85 degrees to -45
        this.physics.arcade.velocityFromAngle(
          -85 + i * 10, 500, bullet.body.velocity
        );
      }
    }
  },
```

One last thing before you test your new spread shot: let's increase the win condition to 20,000 points so that the game will not end before you can see your new weapon in all its greatness:

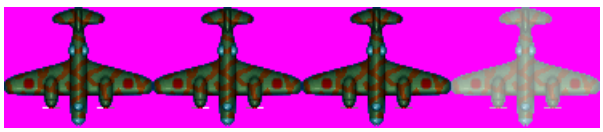
```
    if (this.score >= 2000) {
    if (this.score >= 20000) {
```



Note that it's you can run out of available bullet sprites as shown with the bullet gaps above. You can avoid this by increasing the amount of bullet sprites created in the `setupBullets()` function, but it's not really that necessary gameplay-wise.

Boss Battle

Shooters are nice, but our game wouldn't be a proper shoot 'em up if it didn't have a boss battle.



First let's setup the sprite sheet pre-loading:

```
  preload: function () {
...
    this.load.spritesheet('whiteEnemy', 'assets/shooting-enemy.png', 32, 32);
    this.load.spritesheet('boss', 'assets/boss.png', 93, 75);
    this.load.spritesheet('explosion', 'assets/explosion.png', 32, 32);
    this.load.spritesheet('player', 'assets/player.png', 64, 64);
  },
```

Then the `setupEnemies()` code:

```
  setupEnemies: function () {
...
    this.shooterInitialHealth = 5;

    this.bossPool = this.add.group();
    this.bossPool.enableBody = true;
    this.bossPool.physicsBodyType = Phaser.Physics.ARCADE;
    this.bossPool.createMultiple(1, 'boss');
    this.bossPool.setAll('anchor.x', 0.5);
    this.bossPool.setAll('anchor.y', 0.5);
    this.bossPool.setAll('outOfBoundsKill', true);
    this.bossPool.setAll('checkWorldBounds', true);
    this.bossPool.setAll('reward', 10000, false, false, 0, true);
    this.bossPool.setAll('dropRate', 0, false, false, 0, true);

    // Set the animation for each sprite
    this.bossPool.forEach(function (enemy) {
      enemy.animations.add('fly', [ 0, 1, 2 ], 20, true);
      enemy.animations.add('hit', [ 3, 1, 3, 2 ], 20, false);
      enemy.events.onAnimationComplete.add( function (e) {
        e.play('fly');
      }, this);
    });

    this.boss = this.bossPool.getTop();
    this.bossApproaching = false;
    this.bossInitialHealth = 500;
  },
```

We made a group containing our single boss. This is for two reasons: to put the boss in the proper sprite order - above the enemies, but below the bullets and text; and to deal with the sprite vs sprite collision bug we mentioned way back. We also stored the actual boss in a property for convenience.

We then replace what happens when we reach 20,000 points from ending the game to spawning the boss:

```
  addToScore: function (score) {
    this.score += score;
    this.scoreText.text = this.score;
    if (this.score >= 20000) {
      this.enemyPool.destroy();
      this.shooterPool.destroy();
      this.enemyBulletPool.destroy();
      this.displayEnd(true);
    }
    // this approach prevents the boss from spawning again upon winning
    if (this.score >= 20000 && this.bossPool.countDead() == 1) {
      this.spawnBoss();
    }
  },
```

Then the new `spawnBoss()` function:

```
436   spawnBoss: function () {
437     this.bossApproaching = true;
438     this.boss.reset(512, 0, this.bossInitialHealth);
439     this.game.physics.enable(this.boss, Phaser.Physics.ARCADE);
440     this.boss.body.velocity.y = 15;
441     this.boss.play('fly');
442   },
```

The `bossApproaching` flag is there to make the boss invulnerable until it reaches its target position. Let's add the code to `processDelayedEffects()` to check this:

```
  processDelayedEffects: function () {
...
      this.showReturn = false;
    }

    if (this.bossApproaching && this.boss.y > 80) {
      this.bossApproaching = false;
      this.boss.health = 500;
      this.boss.nextShotAt = 0;

      this.boss.body.velocity.y = 0;
      this.boss.body.velocity.x = 200;
      // allow bouncing off world bounds
      this.boss.body.bounce.x = 1;
      this.boss.body.collideWorldBounds = true;
    }
  },
```

Once it reaches the target height, it becomes a 500 health enemy and starts bouncing from right to left using the built-in physics engine.

Next is to setup the collision detection for the boss, taking into account the invulnerable phase:

```
  checkCollisions: function () {
...
      this.player, this.powerUpPool, this.playerPowerUp, null, this
    );

    if (this.bossApproaching === false) {
      this.physics.arcade.overlap(
        this.bulletPool, this.bossPool, this.enemyHit, null, this
      );

      this.physics.arcade.overlap(
        this.player, this.bossPool, this.playerHit, null, this
      );
    }
```

And modify the `damageEnemy()` to get our game winning condition back:

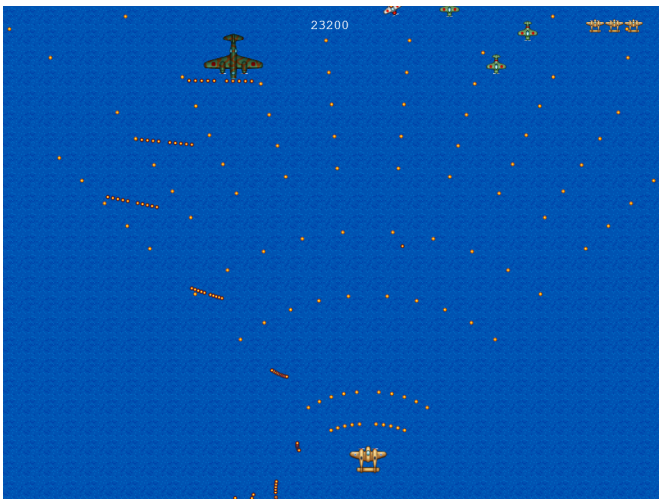
```
  damageEnemy: function (enemy, damage) {
    enemy.damage(damage);
    if (enemy.alive) {
      enemy.play('hit');
    } else {
      this.explode(enemy);
      this.spawnPowerUp(enemy);
      this.addToScore(enemy.reward);
      if (enemy.key === 'boss') {
        this.enemyPool.destroy();
        this.shooterPool.destroy();
        this.bossPool.destroy();
        this.enemyBulletPool.destroy();
        this.displayEnd(true);
      }
    }
  },
```

We've saved the boss shooting code for last:

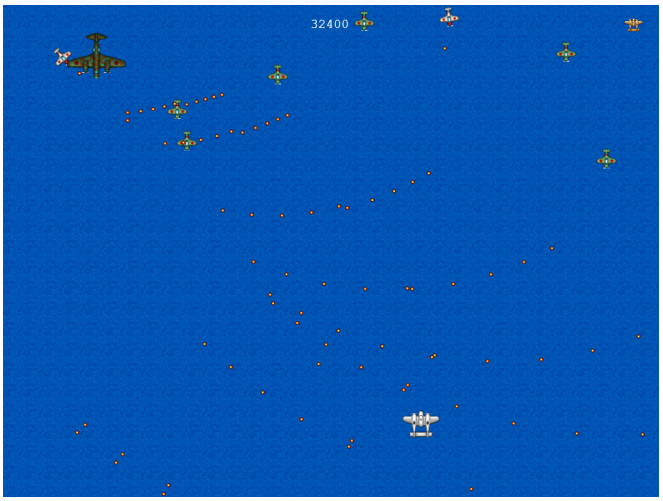
```
  enemyFire: function() {
...
    }, this);

    if (this.bossApproaching === false && this.boss.alive &&
        this.boss.nextShotAt < this.time.now &&
        this.enemyBulletPool.countDead() > 9) {

      this.boss.nextShotAt = this.time.now + 1000;

      for (var i = 0; i < 5; i++) {
```

```
        // process 2 bullets at a time
        var leftBullet = this.enemyBulletPool.getFirstExists(false);
        leftBullet.reset(this.boss.x - 10 - i * 10, this.boss.y + 20);
        var rightBullet = this.enemyBulletPool.getFirstExists(false);
        rightBullet.reset(this.boss.x + 10 + i * 10, this.boss.y + 20);

        if (this.boss.health > 250) {
          // aim directly at the player
          this.physics.arcade.moveToObject(leftBullet, this.player, 150);
          this.physics.arcade.moveToObject(rightBullet, this.player, 150);
        } else {
          // aim slightly off center of the player
          this.physics.arcade.moveToXY(
            leftBullet, this.player.x - i * 100, this.player.y, 150
          );
          this.physics.arcade.moveToXY(
            rightBullet, this.player.x + i * 100, this.player.y, 150
          );
        }
      }
    }
  },
```

There are two additional phases to this boss fight after the “approaching” phase. First is where the boss just fires 10 bullets concentrated to the player.



Then once the boss’s health goes down to 250, the boss now fires 10 bullets at the area around the player. While this is the same amount of bullets as the previous phase, the spread makes it much harder to dodge.



Sound Effects

We've saved the sound effects for the end of the workshop because integrating it with the main tutorial may make it more complicated that it should be.

Anyway, adding sound effects in Phaser is as easy as adding sprites. First, pre-load the sounds:

```
  preload: function () {
...
    this.load.spritesheet('player', 'assets/player.png', 64, 64);
    this.load.audio('explosion', ['assets/explosion.wav']);
    this.load.audio('playerExplosion', ['assets/player-explosion.wav']);
    this.load.audio('enemyFire', ['assets/enemy-fire.wav']);
    this.load.audio('playerFire', ['assets/player-fire.wav']);
    this.load.audio('powerUp', ['assets/powerup.wav']);
  },
```

Then initialize the audio, adding a new function `setupAudio()`:

```
  create: function () {
...
    this.setupAudio();

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

...

  setupAudio: function () {
    this.explosionSFX = this.add.audio('explosion');
    this.playerExplosionSFX = this.add.audio('playerExplosion');
    this.enemyFireSFX = this.add.audio('enemyFire');
    this.playerFireSFX = this.add.audio('playerFire');
```

```
    this.powerUpSFX = this.add.audio('powerUp');
  },
```

Then play the audio when they are needed. Enemy explosion:

```
  damageEnemy: function (enemy, damage) {
    enemy.damage(damage);
    if (enemy.alive) {
      enemy.play('hit');
    } else {
      this.explode(enemy);
      this.explosionSFX.play();
      this.spawnPowerUp(enemy);
```

Player explosion:

```
  playerHit: function (player, enemy) {
    // check first if this.ghostUntil is not not undefined or null
    if (this.ghostUntil && this.ghostUntil > this.time.now) {
      return;
    }

    this.playerExplosionSFX.play();

    // crashing into an enemy only deals 5 damage
```

Enemy firing:

```
  enemyFire: function() {
    this.shooterPool.forEachAlive(function (enemy) {
      if (this.time.now > enemy.nextShotAt && this.enemyBulletPool.countDead() > 0) {
        var bullet = this.enemyBulletPool.getFirstExists(false)
        bullet.reset(enemy.x, enemy.y);
        this.physics.arcade.moveToObject(bullet, this.player, 150);
        enemy.nextShotAt = this.time.now + this.shooterShotDelay;
        this.enemyFireSFX.play();
      }
    }, this);

    if (this.bossApproaching === false && this.boss.alive &&
        this.boss.nextShotAt < this.time.now &&
        this.enemyBulletPool.countDead() > 9) {

      this.boss.nextShotAt = this.time.now + 1000;
      this.enemyFireSFX.play();

      for (var i = 0; i < 5; i++) {
```

Player firing:

```
  fire: function() {
    if (!this.player.alive || this.nextShotAt > this.time.now) {
      return;
    }

    this.nextShotAt = this.time.now + this.shotDelay;
    this.playerFireSFX.play();

    if (this.weaponLevel == 0) {
      if (this.bulletPool.countDead() == 0) {
```

Power-up pickup:

```
  playerPowerUp: function (player, powerUp) {
    this.addToScore(powerUp.reward);
    powerUp.kill();
    this.powerUpSFX.play();
    if (this.weaponLevel < 5) {
      this.weaponLevel++;
    }
  },
```

And now we're done with the full game. We wrap up the tutorial in the next chapter.

Afternoon 6: Wrapping Up

We need to do one last thing before we unleash our game to the public.

Restore original game flow

At the start of the tutorial, we modified our game to skip directly to the `Game` state. Now that the game's done, we'll need restore it to its original flow that we discussed in Afternoon 0.

Let's start by deleting the `preload()` function in `game.js`:

```
BasicGame.Game.prototype = {

  preload: function () {
    this.load.image('sea', 'assets/sea.png');
    this.load.image('bullet', 'assets/bullet.png');
    this.load.image('enemyBullet', 'assets/enemy-bullet.png');
    this.load.image('powerup1', 'assets/powerup1.png');
    this.load.spritesheet('greenEnemy', 'assets/enemy.png', 32, 32);
    this.load.spritesheet('whiteEnemy', 'assets/shooting-enemy.png', 32, 32);
    this.load.spritesheet('boss', 'assets/boss.png', 93, 75);
    this.load.spritesheet('explosion', 'assets/explosion.png', 32, 32);
    this.load.spritesheet('player', 'assets/player.png', 64, 64);
    this.load.audio('explosion', ['assets/explosion.wav']);
    this.load.audio('playerExplosion', ['assets/player-explosion.wav']);
    this.load.audio('enemyFire', ['assets/enemy-fire.wav']);
    this.load.audio('playerFire', ['assets/player-fire.wav']);
    this.load.audio('powerUp', ['assets/powerup.wav']);
  },

  create: function () {
```

Do the same for `mainMenu.js`:

```
BasicGame.MainMenu.prototype = {

  preload: function () {
    this.load.image('titlepage', 'assets/titlepage.png');
  },

  create: function () {
```

Revert the starting state in `app.js` to `Boot`:

```
  //  Now start the Boot state.
  game.state.start('Game');
  game.state.start('Boot');
```

And before we forget, let's destroy the sprites that we added in the previous chapter when we quit the game:

```
quitGame: function (pointer) {

    //  Here you should destroy anything you no longer need.
    //  Stop music, delete sprites, purge caches, free resources, all that good stuff.
    this.sea.destroy();
    this.player.destroy();
    this.enemyPool.destroy();
    this.bulletPool.destroy();
    this.explosionPool.destroy();
    this.shooterPool.destroy();
    this.enemyBulletPool.destroy();
    this.powerUpPool.destroy();
    this.bossPool.destroy();
    this.instructions.destroy();
    this.scoreText.destroy();
    this.endText.destroy();
    this.returnText.destroy();
    //  Then let's go back to the main menu.
    this.state.start('MainMenu');

}
```

Sharing your game

The good thing about HTML5 games is that it's no different from a typical static HTML web site: if you want to share your game to the world, all you need to do is find a web server, upload your files there, and access the server through your browser.

If you used a cloud IDE like Codio or Nitrous.IO for this tutorial, you don't need to do anything – you can just share the preview URL you used when you developed your game. If you developed locally, however, you'll need to decide from the thousands of web hosting solutions out there to host your game.

The only free hosting solution I can recommend right now is Github Pages. Most of the alternatives are either seedy ad-infested sites or free-tier cloud solutions (e.g. AWS, Azure) that require a bit of tinkering just to serve our game.

Unfortunately, Github Pages is not as easy as "drag-and-drop"; you still need to know Git before you can use. So for the sake of those who aren't experienced web developers, we'll be using the simplest free static web hosting out there that doesn't bombard you with ads: Neocities.

Steps for deploying to Neocities

Neocities has a straightforward sign-up page and a simple drag-and-drop interface making it easy even for beginners. There are some caveats, though:

- Neocities doesn't support folders
- Neocities doesn't allow you to upload `.wav` files

Taking these into account, here are the steps to using Neocities to host your game:

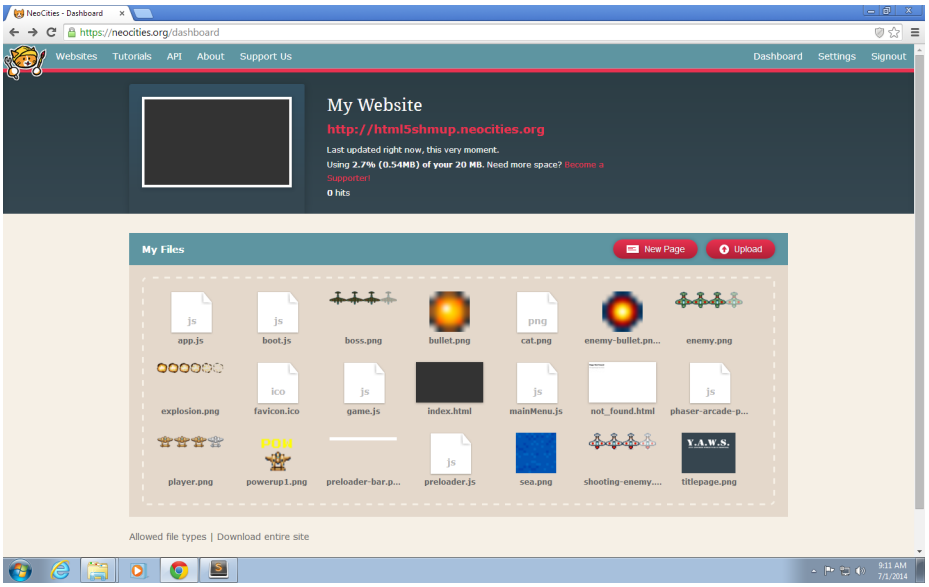
1. **Remove all audio** - Remove all of the `load.audio()` calls in `preloader.js` and the `add.audio()` and `audio.play()` calls in `game.js`. Refer to the previous chapter to find their locations.
2. **Update asset locations** - Move all images from the assets folder to the root then update all of the references in `boot.js` and `preloader.js` to point to the correct location i.e.

```
preload: function () {

  //  Here we load the assets required for our preloader (in this case a loading bar)
  this.load.image('preloaderBar', 'preloader-bar.png');

},
```

```
preload: function () {
  this.load.image('sea', 'sea.png');
  this.load.image('bullet', 'bullet.png');
  this.load.image('enemyBullet', 'enemy-bullet.png');
  this.load.image('powerup1', 'powerup1.png');
  this.load.spritesheet('greenEnemy', 'enemy.png', 32, 32);
  this.load.spritesheet('whiteEnemy', 'shooting-enemy.png', 32, 32);
  this.load.spritesheet('boss', 'boss.png', 93, 75);
  this.load.spritesheet('explosion', 'explosion.png', 32, 32);
  this.load.spritesheet('player', 'player.png', 64, 64);
}
```

3. **Sign-up for Neocities** - Fill up the form at https://neocities.org/new.
4. **Overwrite index.html** - Replace its contents with your `index.html`.

5. **Upload the game files** - Drag and drop all of the `.js` and `.png` files as well `favicon.ico` to the files box. If dragging multiple files doesn't work, upload them one by one.



6. **Verify the game works by opening the site** - If all goes well, you should now see your game (sans sound).

Evening: What Next?

Congratulations! You've just created and deployed your first HTML5 game!

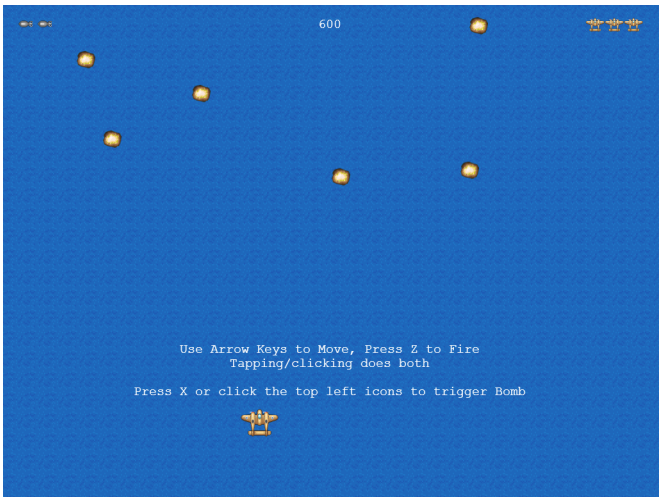
Your journey is far from over, though, and in this chapter we'll go through your next steps.

Challenges

A common problem with coding workshops is that some participants *think* they have already grasped the concepts well when in reality they just knew how to correctly copy-paste the code examples. Prove that you're not one of those people by taking on the following challenges:

- **Add bombs to the game**

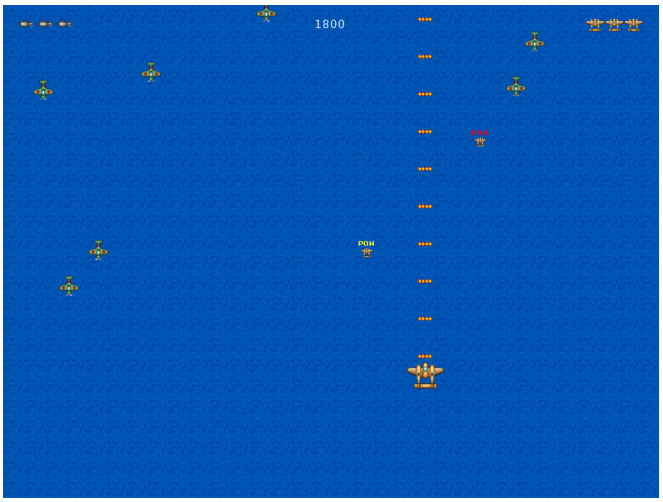
  Players start with 3 bombs, the current count represented by icons on the top left corner of the scene. Pressing X or tapping one of these icons will trigger the bomb, continuously destroying all enemy bullets and dealing a small amount of damage for a few seconds. This is usually done the moment before an enemy bullet collides with the player.



Hint: Use the `bomb.png` as the icon and `bomb-blast.png` as the effect that will cover the whole screen colliding with all enemies.

- **Use the second power-up**

  There's an additional power-up image in the `assets` folder. Use it to give the player a speed boost or a different weapon. For example, here we made the red power-up give a concentrated shot which can be more effective in the boss battle:



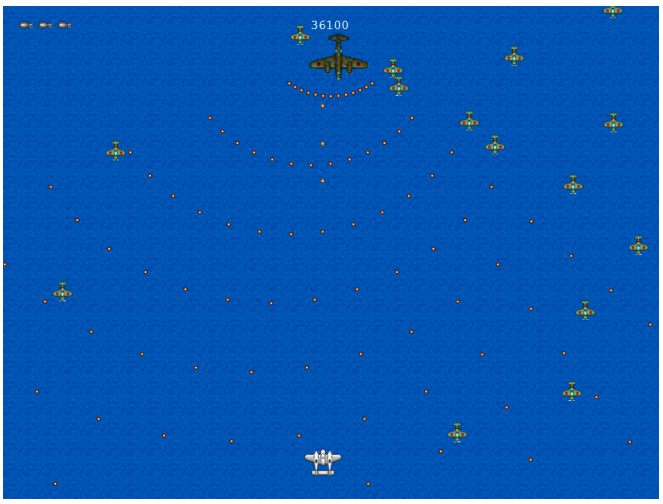
- **Create a difficulty progression**

  Apart from the boss fight at 20000 points, the difficulty stays the same for most of the game session. Add some flags and additional checking to make the game slightly more difficult as the game progresses. For example:

Score	Enemy Spawn Rate	Shooter Spawn Rate	Boss
0 - 2000	1.0s	n/a	n/a
2000 - 5000	0.8s	3.0s	n/a
5000 - 10000	0.6s	2.5s	n/a
10000 - 17500	0.5s	2.0s	n/a
17500 - 25000	0.3s	1.5s	n/a
> 25000	0.6s	stop spawning	spawn the boss

- **Add new patterns and phases to the boss fight**

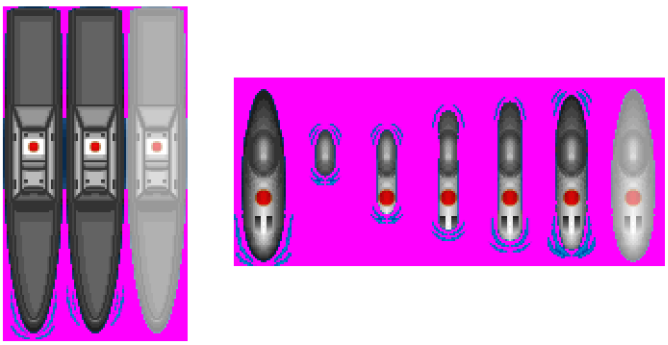
  The patterns can be movement patterns (not just bouncing left and right) and shooting patterns.



A classic shoot 'em up pattern

- **Display the breakdown of kills at the end of the game**

- **Add new enemies: the destroyer and the sub**



There are two unused enemy sprite sheets: one for a destroyer and one for a submarine. Being sea units, they will have to behave a bit differently from their flying counterparts, namely, they are

below all other flying sprites, and they can't overlap with each other.

- **Refactor parts of the code**.

  There are still places where the code is duplicated 3 or more times. Turn them into functions to reduce the code size.

  You can also try converting some of the game objects into JS objects. The *Tank* example in *Phaser Examples* is way to implement this.

- **Convert time-related events to use Phaser's time classes**

  Many of the time-related code in our game only uses the current time as reference. This results in incorrect behavior in certain situations (e.g. pausing the game).

  Replace those code with the appropriate Time and Timer functions. See the *Time* section of *Phaser Examples* for ideas on how to do this.

What we didn't cover

We've skipped a lot of Phaser topics. Here are some topics you might want to look into after this workshop:

- Other *Phaser* settings (e.g. auto-scale, pause on lose focus)
- Background music
- Mobile support (e.g. additional features, packaging to app stores)
- Other physics systems
- Persisting data
- Interacting with libraries and APIs

Many of these are covered by the official documentation. For the rest, feel free to ask about them at the official Phaser forum.

We also did not cover how to prepare assets for your game. There are lists of free resources out there like this wiki page (which also lists where we got our sounds, OpenGameArt.org). You can also Google for assets, but you have to check their licenses and see if you can use them in your games.

Processing assets is also something that is out of the scope of this tutorial. For example, our art assets came from SpriteLib but they had to be converted into sprite sheets that are compatible with *Phaser* (e.g. convert blue to transparent, add damage effect to enemy, etc.), and the volume of our sound assets had to be tweaked a bit.

For image editing, you can look for Paint.NET and Gimp tutorials. For sound editing, you check out Audacity tutorials.

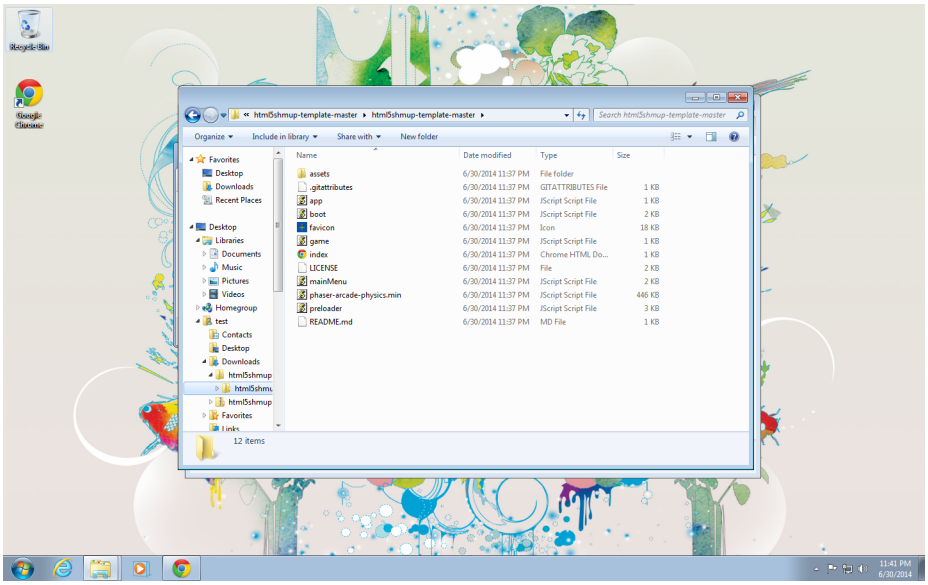
Appendix A: Environment Setup Tutorials

This section is divided into 3 sections. The **Basic** section which provides the most basic ways of setting up your development environment for *Phaser*, the **Advanced** section which are for experienced developers who want a more comfortable environment at the price of complexity, and the **Cloud** section where we have tutorials on how to develop without requiring anything other than a browser and a stable internet connection.

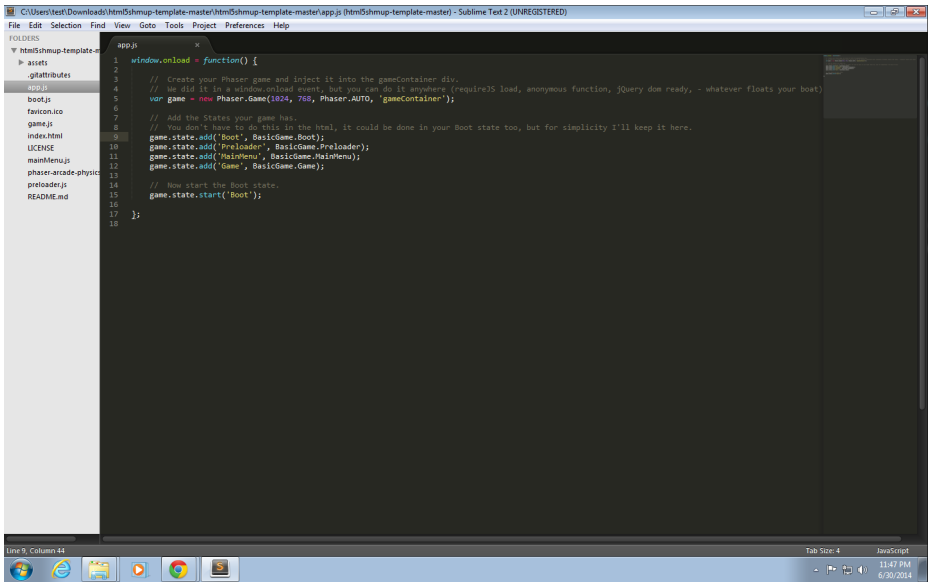
Basic Setup

Here's a basic step-by-step tutorial on preparing your system for the workshop:

1. Download the basic game template from Github and extract it into a folder.



2. Download either version 2 or beta version 3 of Sublime Text and install it in your computer.
3. In Sublime Text, add the folder you extracted to the current project by using `Project -> Add Folder to Project...`



4. This last step, setting up a web server, will depend on your operating system:

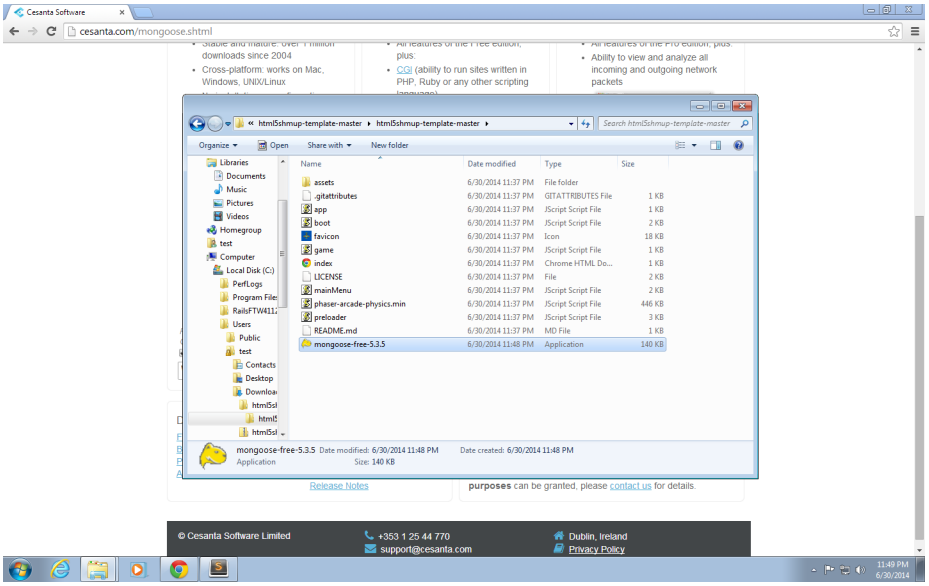
   If you're using **Windows**, the smallest and easiest web server to setup is Mongoose.

   If you're using a **Mac** or a **Linux** / **Unix** machine, the easiest is Python's `SimpleHTTPServer` since pretty much all of these OSs have Python pre-installed.

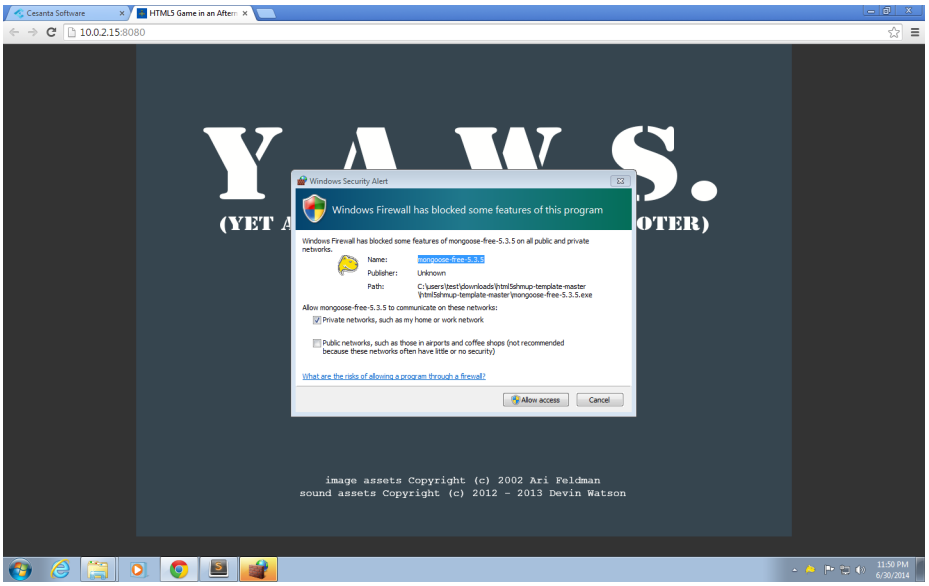
Mongoose Setup

Repeating Mongoose's tutorial:

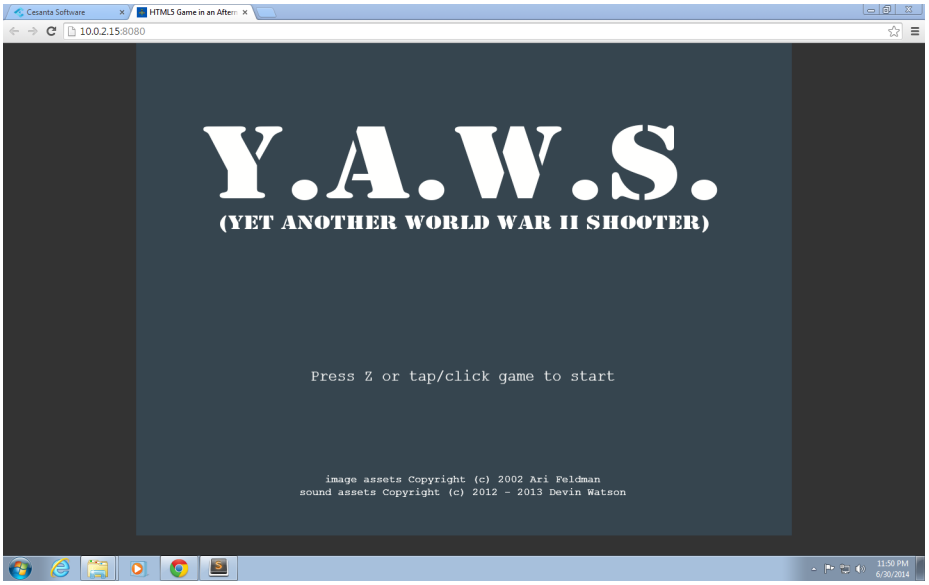
1. Download Mongoose Free Edition and copy it into the working folder.



2. Run Mongoose. Unblock the firewall for Mongoose by clicking `Allow access`.

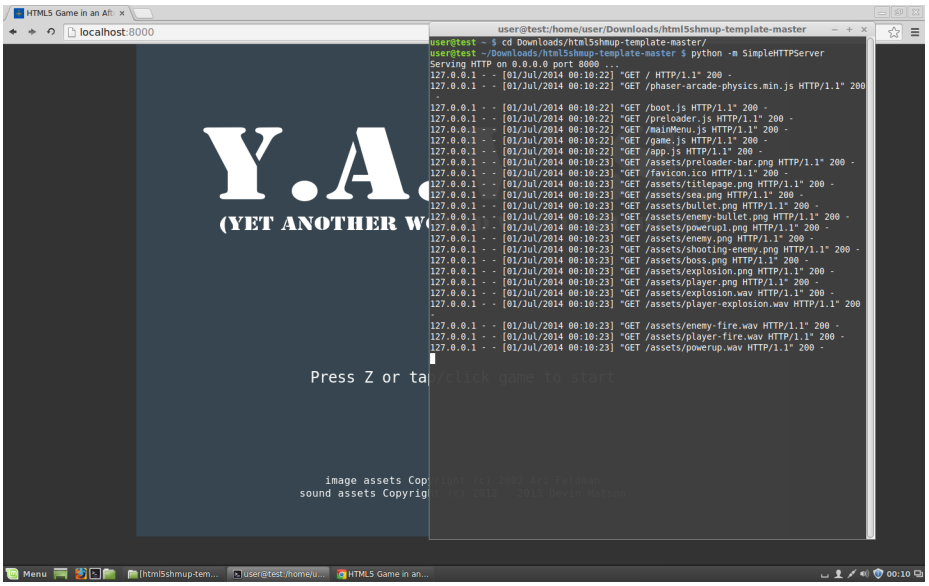


3. Your browser should now be open at the game template.



Starting a Simple Python HTTP Server

1. Open your terminal and go to your working folder.
2. Run `python -m SimpleHTTPServer`.
3. Open your browser to http://localhost:8000/ to access your game.



Advanced Setup

Some experienced web developers might open the basic template and be disappointed at how plain it looks compared to the code they use in their day to day work. To answer this problem, I've made a couple of alternative templates that have the 2 features that I can't live without when developing front-ends: LiveReload and a means for concatenating/minifying/preprocessing JS and CSS.

JavaScript / NodeJS Template

You can find a starting template for NodeJS at the `javascript` branch of the base template.

This template is a slightly modified version of Luke Wilde's phaser-js-boilerplate which uses Browserify, Jade, Stylus, Lodash, JsHint, Uglify.js, Google Analytics, Image optimisation tools, LiveReload, Zip compression, and partial cache busting for assets.

To setup:

```
$ git clone https://github.com/bryanbibat/html5shmup-template.git
$ cd html5shmup-template
$ git checkout javascript
$ npm install
```

Run `grunt` (which you might have to install via `npm install -g grunt-cli`) to start server and open the default browser to http://localhost:3017. You can change the port settings in `src/js/game/properties.js`.

Run `grunt build` to compile everything (pre-process, concatenate, minify, etc.) to the `build` folder for production release.

Refer to the original boilerplate's Github Read Me for other details.

Ruby Template

You can find a starting template for Ruby at the `ruby` branch of the base template.

This template uses Middleman for features like LiveReload and Asset Pipeline. Compared to the NodeJS template, this template's set of libraries are more oriented towards the Ruby ecosystem: ERb and Haml instead of Jade, Sass instead of Stylus, and so on.

To setup:

```
$ git clone https://github.com/bryanbibat/html5shmup-template.git
$ cd html5shmup-template
$ git checkout ruby
$ bundle install
```

To start server:

```
$ bundle exec middleman server
```

Your game will be available at http://localhost:4567. Note that LiveReload is set up to work only for localhost, if you want to make it work on a different machine in the network, you must specify the host in `config.rb` e.g.

```
activate :livereload, host: 192.168.1.111
```

To compile everything to the `build` folder:

```
$ bundle exec middleman build
```

Refer to the Middleman docs for other details.

Note that Middleman's Sprockets interface doesn't support audio so `audio_path` won't work. Check out `_preloader.js.erb` for my workaround.

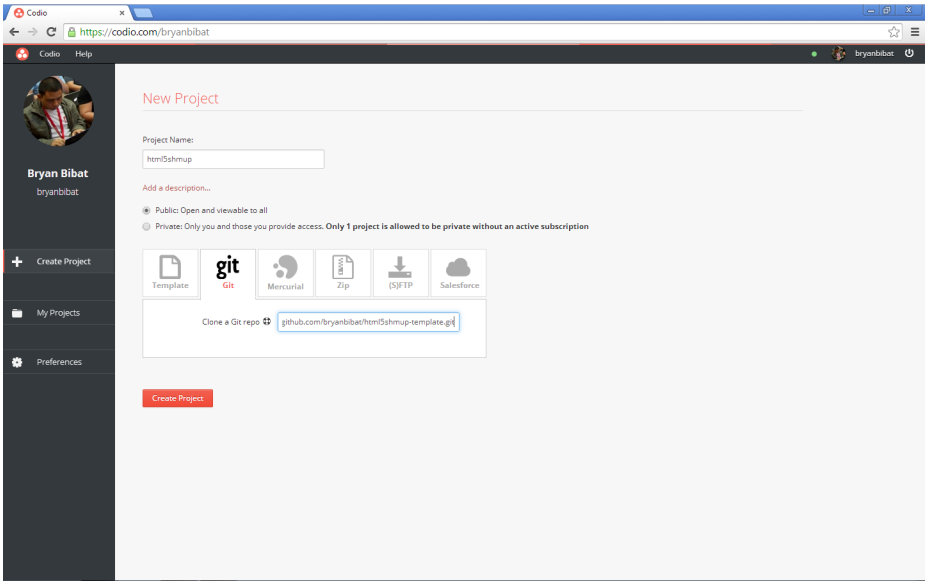
Cloud IDE Setup

Online IDEs like Codio and Nitrous.IO serve as alternative to desktop/laptop-based development. They take away the hassle of having to install additional software on your computer and replace it with the hassle of finding a venue that has reliable internet - this can be a big problem for workshops.

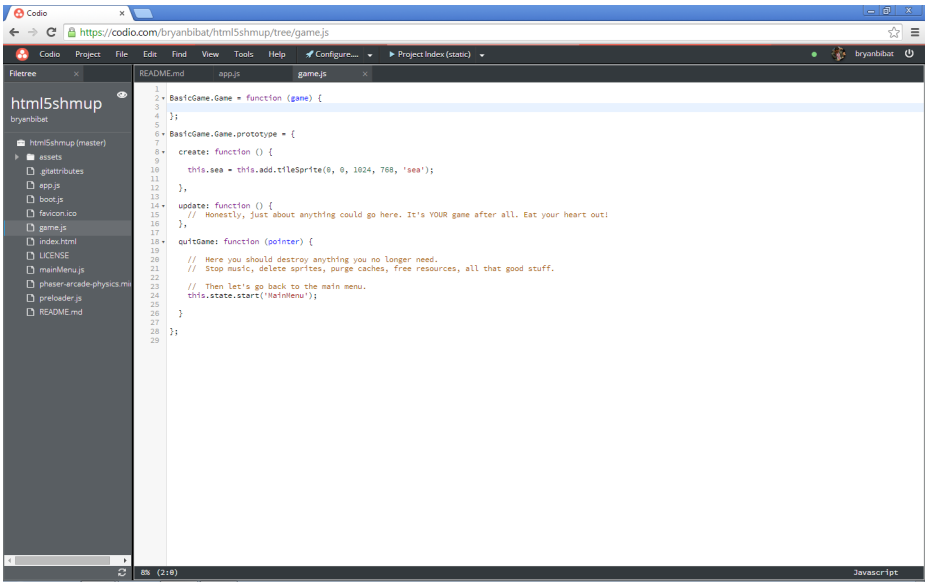
We'll run through the steps of setting up two types development environment: one using Codio on the basic template, and another using Nitrous.IO on the advanced Ruby template.

Codio + Basic Template

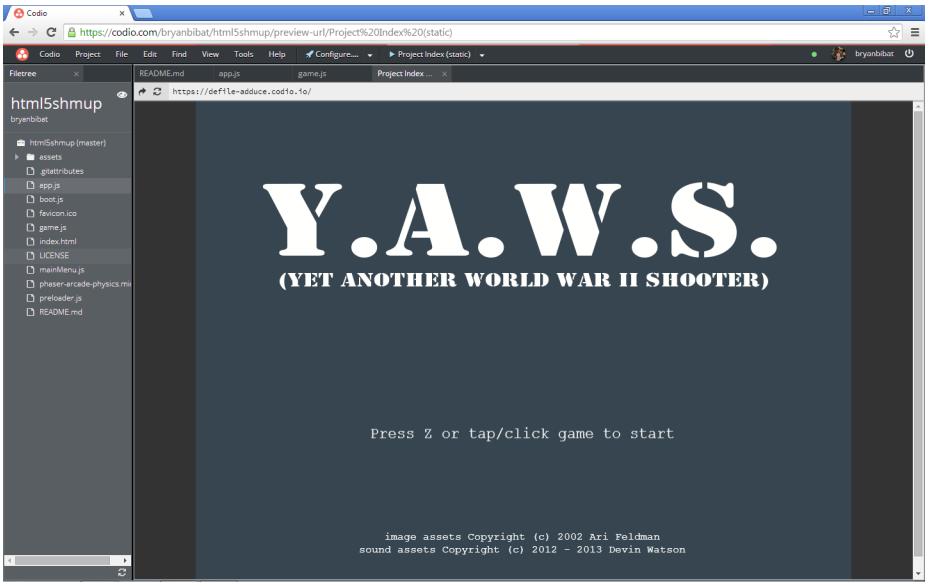
1. Sign-up for Codio by filling out the form at https://codio.com/p/signup.
2. At the dashboard, click "Create Project". Fill out the Project Name and choose Git with the Git repo being the template, https://github.com/bryanbibat/html5shmup-template.git, then click the "Create Project" button.



3. Wait until Codio finishes creating your project. You should be able to start editing your files once that is done.

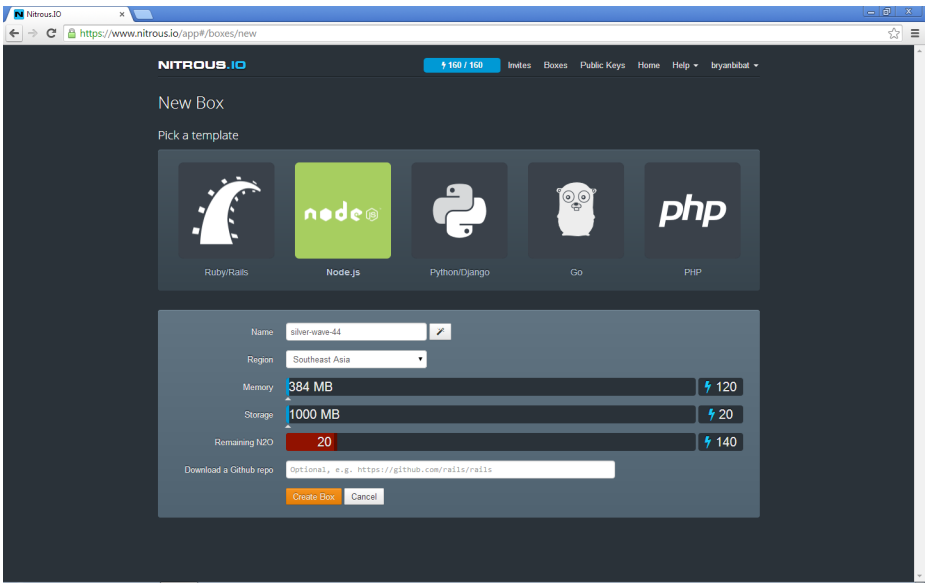


4. Click the "> Project Index (static)" button/link at the header to open your game in a new tab. You can also access your game by opening the URL shown in another browser tab or window.



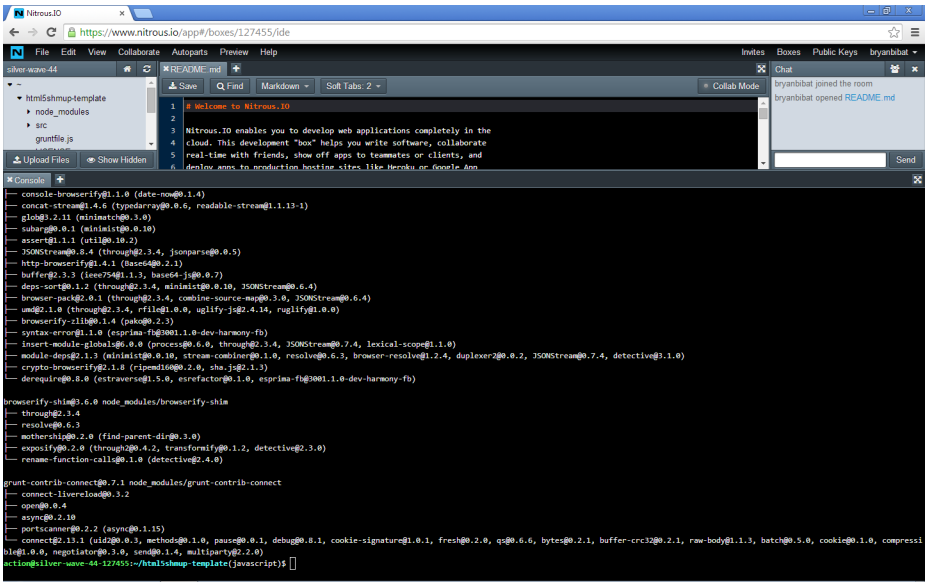
Nitrous.IO + NodeJS Template

1. Sign-up for Nitrous.IO by filling out the form at https://nitrous.io/users.
2. At the dashboard, click "New Box". Choose the "Node.js" template, a Region close to you, and click the "Create Box" button. Leave the Github repository blank.

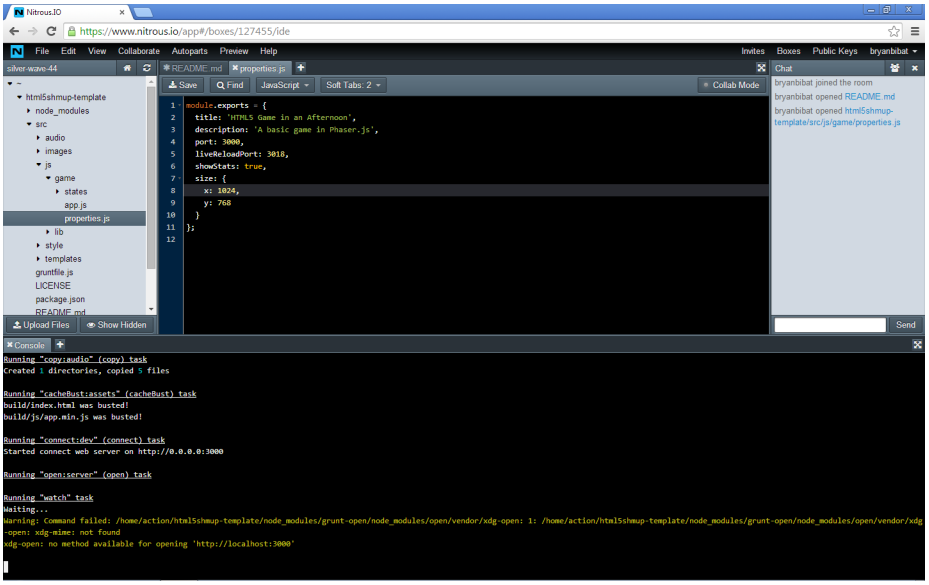


3. Wait until Nitrous.IO finishes creating your project. Once that is done, checkout and setup the Ruby template with the same commands listed in the previous section:

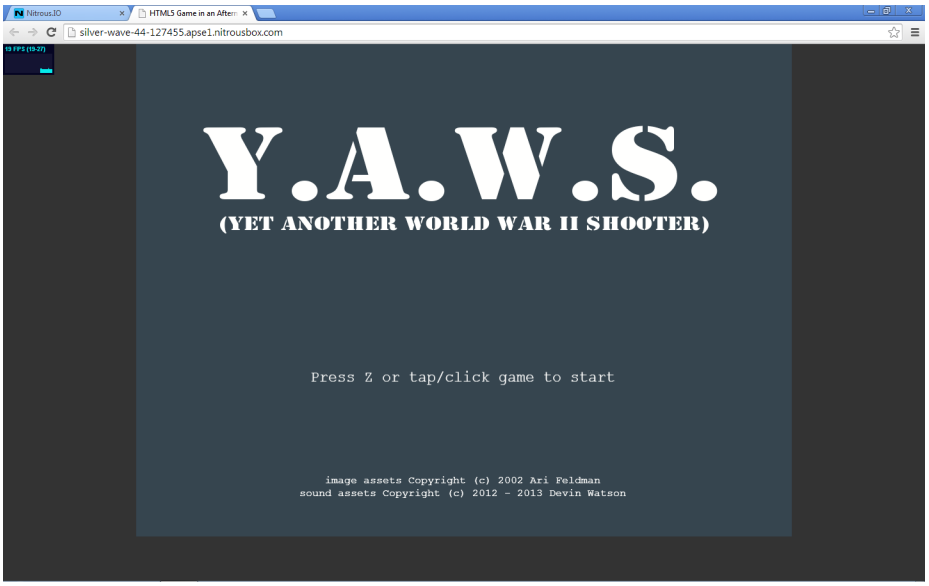
```
$ git clone https://github.com/bryanbibat/html5shmup-template.git
$ cd html5shmup-template
$ git checkout javascript
$ npm install -g grunt-cli
$ npm install
```



4. Open `src/js/game/properties.js`, modify the port to 3000, and run `grunt`:



5. Click the “Preview” -> “Port 3000” button/link at the header to open your game in a new browser tab.



You can now edit your files and make your game. After saving, LiveReload will refresh the game automatically.

Appendix B: Expected Code Per Chapter

We understand that there are cases you need to "cheat" and need to look for the "correct" code after each chapter.

Maybe you've been spending too much time trying to find where you mistyped the code. Maybe a participant had to leave the workshop for 1 hour to deal with an emergency.

Regardless of the reason, here are the working code for each chapter of the workshop.

- **Overview of the Starting Code** - Browse in Github, Download Zip
- **Sprites, the Game Loop, and Basic Physics** - Browse in Github, Download Zip
- **Player Actions** - Browse in Github, Download Zip
- **Object Groups** - Browse in Github, Download Zip
- **Health, Score, and Win/Lose Conditions** - Browse in Github, Download Zip
- **Expanding the Game**
  - **Harder Enemy** - Browse in Github, Download Zip
  - **Power-up** - Browse in Github, Download Zip
  - **Boss Battle** - Browse in Github, Download Zip
  - **Sound Effects** - Browse in Github, Download Zip
- **Wrapping Up**
  - **Restore original game flow** - Browse in Github, Download Zip
  - **Sharing your game** - Browse in Github, Download Zip

To make sure the lazy people don't cheat all the way, we won't provide links to solutions for the *Challenges*.