Making a HTML 5 Game Using Phaser

Workshop Made by Michael Rechenberg

So you want to make a HTML 5 game? Learn more about Javascript and the awesome Phaser library? Just bored and went to a random workshop? No matter the reason, by the end of this workshop you’ll understand the tools necessary for if you want to go on afterwards and make a more advanced game.

**A Bit of Background on Web Development**

Since Phaser creates games designed for use in web browsers for mobile and desktop, you’ll need to be familiar with a few of the core languages used in the web. [Talk about using console/inspect element to view a web page]

1. HTML

* HTML stands for Hypertext Markup Language and it designates the structure of webpages. For this workshop we won’t use a lot of HTML, but we need a little of it to display our game. If you would like to learn more about HTML, W3Schools.com has a good tutorial on basic HTML.

1. CSS

* CSS stands for Cascading Style Sheets and it designates the aesthetics/style of webpages. We won’t use CSS in this workshop, but if are designing a game to be hosted on your website and you want your website to look nice, CSS will do that.

1. Javascript

* Javascript is the language that we will spend the majority of the workshop using. It is actually a programming language whereas HTML and CSS are not. Javascript is not the only programming language for the web (PHP is popular as well), but Javascript is extremely popular with great libraries and frameworks (jQuery, Node.js, and Phaser!)

**Setup**

Phaser needs a web server to work with, so we will be working with Cloud9, a nice cloud-based development environment that requires little work to setup. If you would like to work on a game after this workshop, you can use your own machine as a webserver (‘localhost’)…the specifics of setting up a localhost depends on your operating system so if you’d like to set one up Google how to set up a localhost for your specific operating system.

To get set up with Cloud9:

1. Go to <https://c9.io/>
2. Make an account (or sign in with GitHub)
3. Create a new workspace by clicking the big plus symbol in the middle of the page. Name the workspace whatever you’d like and give it a brief description. There will be an option called “Clone from Git or Mercurial URL”…copy-paste this link into that text field

<https://github.com/MichaelRechenberg/PhaserWorkshop.git>

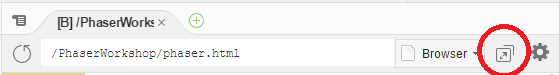
1. If everything goes well, you’ll see a file called README.md that is already open. If it says “This repo contains all of the example code for the CS Sail 2016 Workshop on Phaser” and you see a bunch of files ending in .js then everything worked. Raise your hand if you have any questions.

This is the structure of the repository:

* phaser.min.js: Contains the Phaser library in minimized form (uses the least amount of space possible i.e. all unnecessary whitespace is removed). Do not modify this file
* phaser.html: This is the HTML that the webserver will serve to the client. It contains a div element that the Phaser game we make will be inserted into.
* Various Javascript files: These files contain snippets of code that we will go over in the workshop. We will make our own file in which we will type in the code.
* Assets folder: this will contain any assets we need (images, sound, etc.)

Viewing Your Work: Viewing the Game in a Web Page

Throughout the workshop we will work on a Javascript file and then view the result of the code in a web page. In order to do this we’ll do a “live-preview” of the HTML page that links to our Javascript code.

* Open the phaser.html page (or whatever HTML page you create if you wish to work on games after this workshop)
  + Look at the 2nd <script> tag in the HTML page. In its “src” attribute it should equal the name of a file we are working on (main.js, main01.js, main02.js, etc.). Make sure that whatever Javascript file you want to view/test is included in this src attribute…otherwise you’ll be viewing the result of different code.
* While the HTML source code is open in a tab, click the “Preview” button on the top-center of the page
  + Click the “Live Preview” Option
  + To open the preview in a new tab within your actual browser click this button:

Debugging (Optional)

Being able to debug your game is essential: lucky for you Chrome comes with its own console for print statements and debugger that you can use to step through your code.

* While your HTML page is open right-click anywhere on the page that is not within the 800x600 rectangle that the game resides. Click “Inspect”. Then on the top toolbar click “Console.” This will display anything printed out by console.log() statements.
* To debug your code line by line, click “Sources” on the top toolbar. Click your .js file and you can insert breakpoints, step into, and step over functions from here.
  + You should never have to “step into” phaser.min.js. It’s minified so as much white space is removed as possible to make the file size as small as possible. This results in it being impossible for a ‘sane’ human to read, so don’t even bother.

**Creating the Phaser.Game Object (main01.js)**

Now with all that boring setup out of the way let’s get down to coding!

First, create a new file called “workshop.js”. You do this by first clicking on the PhaserWorkshop directory and clicking File > New File. Then to save the file click File > Save as and name the file “workshop.js”. You will type your code here during the workshop (and if we are running short on time and you want to follow along but you have a weird error, you can copy the corresponding main#.js file to workshop.js or refer to the main#.js later).

Open up phaser.html and where it says src= “main.js” in the second <script> tag, replace “main.js” with “workshop.js”

Copy and paste the contents of main01.js into workshop.js

View the game in a webpage like how I described in the previous page. You should see a black box 800x600 pixels wide. Let me know if you’re having errors.

**Displaying Images (main02.js)**

Now we’re going to start displaying sprites to the screen.

First we have to load our assets (which for now is just an image). We do this on line 41 with a call to game.load.image(). The first argument is the key (‘sail’ for this example) by which we will refer to the image in later function calls and the second parameter is the name of the file (if the file is not in the same directory as the HTML, you must provide the relative path to the file…assets/images/CS\_Sail.png).

But just loading the asset does not display the image in the game. To actually display the image we will create it in the create() function with a call to game.add.sprite(). This function takes 3 arguments: the first argument is the x position of where you want to place the sprite, the second argument is the y position, and the third argument is the name of the key of the image we want (which is ‘sail’ for this example). [EXPLAIN HOW X AND Y WORKS] This function also returns a reference by which you can manipulate the image later…we will store this reference in a variable named ‘sail’. To modify the image, simply invoke a Phaser.Sprite method on a reference to a sprite object, as we do on line 52 by calling the .anchor.setTo() function [explain what anchor does] on the variable sail.

If you preview the HTML page (did you remember to change the src attribute in the second <script> tag to “workshop.js”?) you’ll see a picture of CS Sail in the middle of the screen. Let me know if you have issues.

**Physics (main03.js)**

[load background and dude]

For line 44 we use game.load.spritesheet() instead of game.load.image()…I will go over why we do that later on in the workshop.

[declare player variable, add background]

To use physics we’ll first have to start the system. We do so by calling game.physics.startSystem(), passing in which physics system (Ninja, Arcade) we want to use.

[add player]

Now we want physics to work on the spritesheet that we’ve called the player, so we call game.physics.arcade.enable(), passing in the reference to the player spritesheet (variable player)

[bounce, gravity, collidWorldBounds]

These lines give the player some physics characteristics.

If you preview the HTML you’ll see the player fall downwards and bounce when he hits the bottom of the game. Let me know if you have any issues.

**Movement and Animation/SpriteSheet (main04.js)**

[Add movementControls and ground variables]

[Add ground, enable physics on ground, scale, enableBody, body.immovable]

Now we are going to go more into the concept of SpriteSheets. Go to the assets folder and open up dude.png. You’ll see that it is one image that contains different frames of the dude. When we loaded in the spritesheet earlier, the parameters we sent in split up the image into each of its frames. The spritesheet contains 9 frames, where 0-3 contain animations for running to the left, 5-8 contain images for running to the right, while frame 4 is for when the player is standing still.

So on lines 82 and 83 we create two animations, one for walking to the left and one for walking to the right using animations.add(). The first argument is key that we use to refer to a specific animation, the second parameter is an array containing the frames we want to use in our animation, the third argument is the frame rate (a larger number means that the animation will play faster), and the last argument is a Boolean value indicating if we want the animation to loop.

We want to have the animations play when the player moves left or right, but before we can do that we have to create our input. To do this we call game.input.keyboard.createCursorKeys(), which returns a reference to which we can access the up, down, left, and right arrows later.

Now we start to do work in the update() function, which is called every frame. First, we want to calculate if the player has collided with the ground, so line 92 does that by calling game.physics.arcade.collide(). The arguments given are the two sprites to test collisions against. This function is extremely powerful and we use it in an even better manner later.

[Just type out the giant if/else statement and jumping if statement]

These statements basically say if the right or left arrow is pressed, move the character the corresponding direction by adding velocity in that direction and playing the corresponding animation. If neither left or right is pressed, the dude will stop moving in the x-direction, all animations will stop, and the dude will face the screen.

The last if statement says if the up button is pressed AND the player is touching down, then jump in the air (without the second condition, the dude could ‘fly’ indefinitely like he had a jetpack)

**Groups and Shooting (main05.js)**

[add spaceBar and playerProjectiles variables]

[add projectiles image in preload]

Now we are going to work with Groups, another Phaser goodie. Groups allow you to refer to a bunch of objects with only one variable. It’s like if you are running a store and you sell apples but you want to increase the price of apples. You could have variable apple1, apple2, …apple1337 and so on and update their price, OR you could use a group and make the change in one line.

The group we are going to make contains the projectiles that the dude will shoot. First, we tell the Game that we want to make a Group by calling game.add.group() and storing its return value in playerProjectiles. Next, we modify some of the attributes of the playerProjectiles group. enableBody = true means that physics will apply to any sprite made from this group. The 2 calls to setAll sets 2 properties for all sprites made from the playerProjectiles group: checkWorldBounds means that the sprite will update its internal Boolean value indicating if it within the “World” i.e. what the player can see (this is not the full definition of a “World”, but it is sufficient for this workshop); and outOfBoundsKill, which means that if the sprite is out of bounds of the world the kill() method, which makes it so the sprite is not redrawn on when update() executes (saving time because drawing is a relatively expensive operation.

Next, we work on adding the space bar functionality. Using game.input.keyboard.addKey() and game.input.keyboard.addKeyCapture() we map a keycode (Phaser.KeyCode.SPACEBAR) to the variable spaceBar. Then we write the line spaceBar.onDown.add(shoot,this); which adds an event listener to Phaser’s Input Manager so that when the user presses the spacebar down, Phaser’s Input Manager will call the function shoot. “this” refers to the context in which the first argument will be called with.

[add timeElapsed and TIME\_TO\_SHOOT before update, if timeElapsed statement]

Finally, we write the shoot() function that will be invoked when the user presses down on the space bar. We will create sprites that belong to the playerProjectiles class and send them off with an initial –y velocity have them fly upwards. To do that we declare the function like any other Javascript function and have an if statement to check to see if we can shoot (so the user cannot spam the space bar and shoot a bajillion shots in a row). Then we store the reference to the sprite returned by playerProjectiles.create(player.x-5, player.y, ‘projectile’); into the variable bullet. The first to arguments are the x and y coordinates of the sprite and the third argument is the key string used to identify our projectile sprite. Because when we created playerProjectiles we said that all children of this group will have their physics enabled (enableBody = true), we can access the bullet’s physics properties immediately, namely setting it’s y-component of its to velocity to -200 (sending it upwards). And we reset timeElapsed to 0.

**Enemies, Looping, Overlapping Sprites (main06.js)**

[preload enemy ship, make enemies variable, spawnTime, spawnSpeed, lives]

Now we are going to make our enemy ships. As with the projectiles we will make a Group in create() that has physics enabled for all children and add children to it in a later function. We want to spawn ships continuously, so we will use a Loop event in Phaser. Phaser has an internal clock that determines how long something has elapsed (and can differ from on-the-wall clock time) and you can set it so certain functions are called after X amount of milliseconds have elapsed. To do this we call game.time.events.loop(spawnTime, spawnEnemy, this). The first argument is how many milliseconds should pass before the function is called, spawnEnemy is the function we want to call, and ‘this’ is the context.

We’ve set it up so spawnEnemy() will be called after spawnTime milliseconds have elapsed in Phaser’s internal clock, but now we have to actually write the spawnEnemy function. The code is similar to when we worked with playerProjectiles: make an enemy by calling the create function on the Group, giving it an x and y value, then updating its velocity.

Now that we have enemies, we need to add code that emulates their expected behavior: if an enemy touches the ground we want the player to lose a life and if a player’s projectile hits an enemy we want to remove it from the screen. In both cases we will deal with Sprite overlapping by adding rules via game.physics.arcade.overlap() in Phaser’s update() function. More specifically, we will specify a function to be called if Phaser detects that ANY enemy sprite hits the ground or is hit by ANY player projectiles…to which end Groups make our lives a lot easier. In update(), the line: game.physics.arcade.overlap(playerProjectiles, enemies, projectileEnemyCollision, null, null, this); means that if ANY sprite of the playerProjectiles Group overlaps with ANY sprite of the enemies Group, Phaser will call projectileEnemyCollision (the other arguments are extra options which are not needed for this game). We write a similar rule for the enemies hitting the ground, except for the second argument we pass in the variable referring to the Sprite associated with the ground of our game. Overlap() can accept multiple different types in its parameters and you can look in Phaser’s documentation for more information.

removeLive() is a simple function that invokes the kill() function on the enemy sprite that made contact in the ground and decreases the lives variable by one until it reaches zero, then it prints out to the console “GG”. The kill() function does not remove the sprite from memory, but simply makes it so it is not drawn when Phaser redraws the next frame (which is the more expensive part of the rendering process). projectileEnemyCollision is similar in that it invokes the kill() function on both the projectile and the enemy. The ordering of the parameters of projectileEnemyCollision and removeLife depends on the ordering of parameters when we created the rule with overlap(): since the 1st argument to overlap for projectileEnemyCollision was the playerProjectiles Group, the projectile parameter will contain the specific Sprite that is a child of the playerProjecitles Group and was overlapped; a similar reasoning follows for enemies. If you mix Groups and Sprites things get a little dicey, and I’d recommend you look at the documentation for more detail.

**Text Objects(main.js)**

[add score, scoreText, livesText variables before create()

To finish our game we just need to add a score and a mechanism to let the user see how many lives they have and what score they have. To do this we will use Text Objects. At the end of the create function we will add 2 text objects using game.add.text(), passing in an initial x and y value as well as the text we want to display. Since the default text color is black and the livesText will be over a dark green ground, we pass in an optional style argument, which is a JSON object that sets the “fill” property of the text to #FFFFFF (white in hexadecimal).

Now that we have our Text Objects created we just have to update their text when the player increases their score (hits an enemy) or loses a life (enemy hits the ground). To do this we invoke setText() on the livesText variable in removeLife() and invoking setText on scoresText in projectileEnemyCollision().

Lastly, we create a gameOverScreen() function that will create some more text objects telling the user that they lost and what their final score was. We make these Text Objects in a similar fashion to livesText and scoresText, using anchor.setTo() to make positioning easier.

**Conclusion**

We’re done! The game is finished and feel free to play it our tweak some values to make the game harder (maybe have enemies spawn more rapidly or descend at a faster rate). Now this version of the game makes the user refresh the game in order to play again. We can prevent this dilemma by using States in Phaser, but given time constraints we don’t have enough time to go over it in detail. However, if you would like to see States in action, I have written the code for you that is the same exact game but with States that you can see work if you “Live Preview” the file phaser-states.html. Converting our game to use States is not terribly difficult—but if you would like to learn it yourself the best I could do is point you to the Phaser website for tutorials.

**Happy Coding!**

