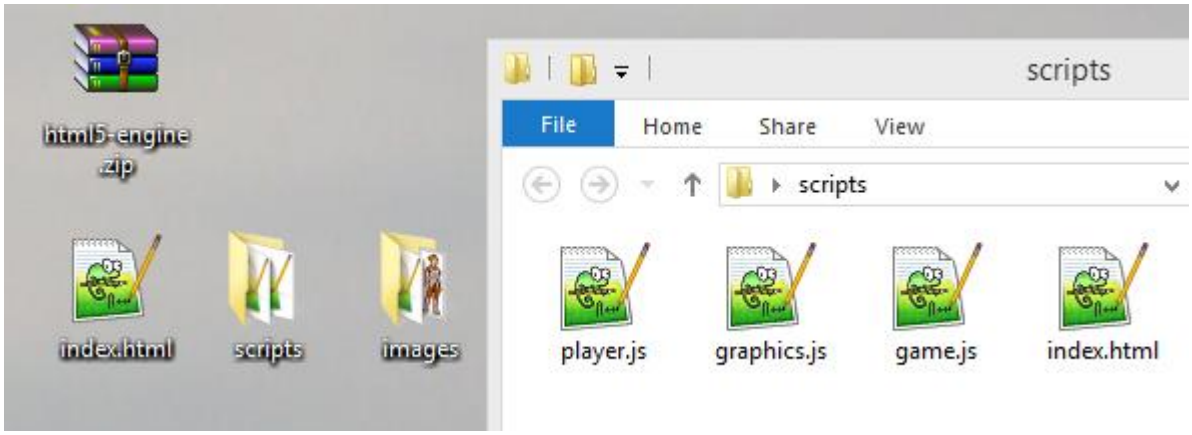
How to Make an HTML5 Game, Part 1: The Basic Engine

Did you play *Super Markup Man* yet? It's the HTML5 game I made that also teaches HTML. In an earlier post, I said I wanted to help you learn how to make your own game. Today is the day! But we're not going to build something as complex as *Super Markup Man*. First things first: understanding what makes up a basic game engine. Download this zip file, unzip the files to your desktop, and open the *index.html* file in a browser. That's *Super Markup Man* stripped down to the very basics. And it takes the work of all of these files to pull it off!



If you open a file in a text editor (on Windows, I prefer Notepad++), you can see what's going on behind the scenes. Since *index.html* is the file that runs the game, let's look at that one first. Open it up and check out what's on line 25:

<canvas id="game"></canvas>

That's it… that's the game! It looks simple, because we don't actually do any game programming in HTML. That's all done with Javascript. Look at the top of the file, on line 18. Here, we are linking to three external Javascript files:

<script src="scripts/graphics.js"></script>
<script src="scripts/player.js"></script>
<script src="scripts/game.js"></script>

You can find these files in the *scripts* folder. When you open them, you'll see that I've left plenty of comments for you, explaining the code. I don't want to just repeat what's already in my comments, so I'll leave it up to you to read over them. What I want to do for this blog post is provide a general overview of what each of the files is doing so you're not completely lost when you tackle it by yourself.

Open the *graphics.js* file. This file handles all of the visual stuff for the game. There are three functions in here. The first one, on line 10, is part of a **window.onload** event. That means we

don't want that code to run until after the browser window has loaded. Once we know the window's loaded, we can create a link between a Javascript variable and the canvas element we had set up in our HTML page. This is accomplished on line 12:

canvas = document.getElementById('game');

The second function, **loadImg(x)** on line 29, is my little way of making sure all of the graphics have loaded before the game starts. If you try to play a game with a lot of graphics before all of the graphics are done loading, the game will crash. So it's important for your game engine to wait. Because we're just working with a simple engine, we only have one image to load, which we do on line 52:

var spriteSheet = loadImg('images/markup-man.png');

The third and last function in *graphics.js* is the **render()** function on line 55. This is what draws everything onto the canvas element. Nothing in *graphics.js* calls that function, though. This is being called in another file, *game.js*. It's okay for different Javascript files to talk to each other. In fact, *graphics.js* calls a function, **init()**, that's located in *game.j*s. On line 24 (or line 43, whichever happens first), **init()** is called once all of the images have loaded. Let's go take a look at it now.

Open *game.js*. On line 28 is the **init()** function again. This is the function that basically "turns on" the game engine. It creates the player and starts a loop that calls the **main()** function every 17 milliseconds. That's super fast, right? So if **main()** is getting called every 17 milliseconds, it must be pretty important. It's located on line 37. **main()** does two things:

update();
render();

Remember, the **render()** function is back in the *graphics.js* file. That's why this loop is being called every 17 milliseconds; we need it to re-draw the game as quickly as possible in order for it to look good. We also need to update the game logic at the same time. That's what the **update()** function is for. To recap, **init()** starts the game, but **main()** *is* the game that does two important tasks: **update()** and **render()**.

The **update()** function is on line 46. It does a bit more than the others do. First, it updates the player via **player.update()**. We'll look at that in more detail in a minute. The other thing we're doing in **update()** is some actual game stuff. Now this is a pretty simple game. All that happens is that the border of the game window changes to red when you touch it. We're able to do this by updating the canvas's style:

canvas.style.borderTop = '2px solid red';

Each **if/else** statement is checking to see where the player's position is. **If** the player is at the top of the screen, it will change the top border to red, or **else** it will change it back to black. The player's position is calculated with **player.x** and **player.y**. The **x** and **y** values are built into the

player object. It'll make sense once we look at the *player.js* file, but don't go there yet. There's one last thing to explain in *game.js*.

At the top of the file is a bunch of crazy-looking code. In order for the player to be able to control his/her character, we need to set up "event listeners." This is a way of telling the browser to "listen" for certain events to happen. In this case, we want it to "listen" for keyboard presses. We have two listeners for this on lines 11 and 22:

addEventListener("keydown", function (e) { });
addEventListener("keyup", function (e) { });

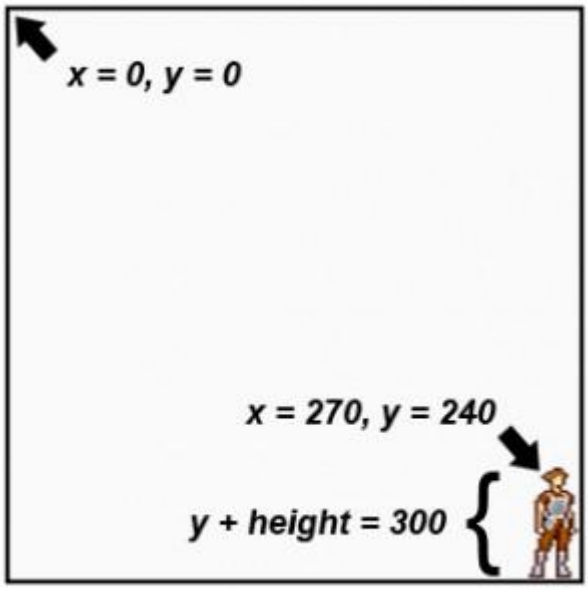
These let us capture any time the player pushes a key down and any time he/she lets go of a key. It stores all pressed keys in a special variable called **keysDown**. The other variable on line 8, **keys**, is like a dictionary I made of only the keyboard keys we care about. Javascript doesn't know what the word "left" means; it only knows what "37" means. The dictionary ties the key code to the actual key name to make it easier for us to use later. So instead of checking for **37 in keysDown**, we can check for **keys.left in keysDown**. Again, you'll see what I mean in the next file.

Finally, it's time to open *player.js!* In here, we have a function called **createPlayer(s)** that will… you guessed it… create the player for us. When the player is created, a few important values are given to him/her, including the player's sprite sheet (or graphic), a starting position, an **update()** function, and a **draw()** function. Those two functions are called from the other files (every 17 milliseconds), but all of the logic takes place here.

For now, the only logic I have programmed in is the player's movement. There are more **if/else** statements here, only this time, they look at the keyboard variables set up in *game.js* to determine where the player should move. If the player is currently pressing the left arrow key, it'll update his/her **x** value. **x** is the left-to-right position. **y** is top-to-bottom. And we change it (and anything else that belongs to the player) by using the word **this**:

this.x -= this.speed;

That will move the player to the left. The smaller the **x** value, the farther left the player is. If we want to move to the right, we can add to make **x** larger. Likewise, the greater the **y** value, the farther down on the screen the player is. The **x** and **y** values start at zero in the top-left corner. All movement is based off of that point. Don't forget!



The player's own **x** and **y** values are also based on the top-left position of the character graphic. So when the player is standing at the far right of the screen, touching the border, the **x** value is only 270, even though the width of the game window is 300. We need to take into account that the player is 30 pixels wide. If we ever let the player's **x** value equal 300, the player would actually fall off the screen. That's why the **if** statements that deal with movement have a second true/false question:

if (this.x + this.width <= canvas.width && keys.right in keysDown) { }

Whew. I'm tired now. No more programming talk! I'm sure this was confusing, but if you read through the comments in the code, it should start making a little sense. Don't be afraid to try editing some of it, either! You can always download the zip file again if it gets messed up. In my next post, I'll go into even more detail on how to make your character jump on top of platforms. Yeah, just like in *Super Markup Man!* Stay tuned… it's gonna get awesome real soon.