An Introduction to HTML5 Game Development

Modern web technologies such as WebGL, canvas and WebAssembly are enabling a new way of building video games – on the web.

Companies like Snapchat, Facebook and WeChat are taking advantage of this trend to host games running on HTML5 in their apps, and it's also possible to ship HTML5 games to the App Stores using tools like Apache Cordova.

In this guide, we'll run through the creation of a really simple Tower Defense game using modern web technologies, and propose some ideas for you to keep working on your own.

There are a host of HTML5 game engines to make your life easier. In this guide, the goal is to learn the basics, so we'll be writing our code from scratch.

The Web

Although we're going to be building our game from scratch, that doesn't mean there aren't some tools that we can use to help us.

One tool, Webpack, will help us keep our code organized by allowing us to `require` the relevant dependencies in our main program. If you haven't used Webpack before, I'd encourage you to read the Getting Started guide here, though this is not required to continue this tutorial.

Webpack runs on Node, which we'll also be using to manage our third party dependencies. This guide assumes you have Node.js installed.

Setting Up a Webpack Development Environment

To get started with Webpack, we need to install it as follows:

```
mkdir mygame && cd mygame
npm init -y
npm install --save-dev webpack
npm install --save-dev webpack-cli
npm install --save-dev webpack-dev-server
```

Now create the following files:

```
/index.html
/src/index.js
```

In `src/index.js` type:

```
console.log("Hello World");
```

and in `index.html` write:

```
<!doctype html>
<html>
  <head>
    <title>My Awesome Game</title>
  </head>
  <body>
    <!-- The main.js file is autogenerated by webpack! -->
    <script src="main.js"></script>
  </body>
</html>
```

Now, start up your game with `node_modules/.bin/webpack-dev-server` and open a browser to `localhost:8080`.

Observe that "Hello World" is written in the console. Now that you are running the `webpack-dev-server`, you can make changes to your JavaScript and note the browser will refresh automatically.

Following Along

If you run into any issues, you can follow along with my commits on Github. This is the commit for the first part of this exercise.

Drawing on the Canvas Element

The Canvas Element

The canvas element is used to draw graphics on a web page and is the central element that enables games on the web. You can draw in 2D or 3D (we'll get to that much later).

Creating a canvas element is simple. Just add the following code between the `body` tags in your `index.html` file.

```
<canvas id="main-canvas" width="600" height="800"></canvas>
```

You can draw on the canvas by adding the following code to `index.js`.

```
const canvas = document.getElementById('main-canvas');
const context = canvas.getContext('2d');
context.fillRect(25, 25, 100, 100);
```

The `context` is how we'll be drawing to our canvas going forward. When we call `fillRect` we tell context we want a rectangle at x = 25, y = 25 with a width of 100 and a height of 100. The units are in pixels based on the top left corner of the canvas element.

Drawing a Character

Games usually have better graphics than simple shapes, so let's add a character.

If you're handy with pencils, you might want to try drawing your own. Otherwise, there's plenty of both free and paid game art available on the web for you to peruse. You can also use my green blob.

To load an image, we are actually going to create a classic `img` element in JavaScript. Because we are going to create a lot of images going forward, let's create a new file for this:

src/assets.js

```
export function loadImage(url) {
    return new Promise((resolve, reject) => {
        const img = new Image();
        img.addEventListener('load', () => {
            resolve(img);
        }, false);
        img.addEventListener('error', e => {
            reject(e);
        });
        img.src = url;
    });
}
```

This helper functions returns a promise that resolves when the image loads. If you haven't used promises in JavaScript before, I highly recommend reading this great primer.

In short: promises allow us to wait for an event (such as image loading) to complete. In this example we need to wait for the image to load before we can draw it:

src/index.js

```
const assets = require('./assets.js');

const canvas = document.getElementById('main-canvas');
const context = canvas.getContext('2d');

assets.loadImage('images/green_blob.png').then(img => {
    context.drawImage(img, 25, 25);
});
```

You should now see your image drawn on the canvas.

Viewport and Scale

So far we're just drawing from the top left of the screen with no regard to size, but this has a few major drawbacks.

We want people to be able to play our game with all kinds of screen sizes and devices. One of the great advantages of HTML5 games is that they are completely cross-platform. They can be played on mobile, on web

Drawing a Background

For practical purposes, it will be useful to know where our canvas is, so let's draw a background. Insert this code just before the `loadImage` call in `index.js` that draws our character:

```
context.fillStyle = '#e2fcbf';
context.fillRect(0, 0, canvas.clientWidth, canvas.clientHeight);
```

You should see a light green box appear behind the character.

Positioning and Resizing the Canvas

Let's say we want this game to be mostly played in portrait mode on mobile devices. It makes sense, then to treat our canvas as if it were sized at around 1080 pixels high and 720 pixels width.

If we actually do this, however, it may be necessary to scroll our browser on desktop, and some phones won't play well either. So let's scale everything accordingly!

First, let's remove any margin on our webpage and add a background with some classic CSS. We'll also center our canvas using `margin: auto`:

```
<style>

  body {
    margin: 0;
    background-color: #000000;
  }

  canvas {
    display: block;
    margin: 0 auto;
  }
```

```
</style>
```

Now let's add some JavaScript so that our canvas is resized to the size of the screen:

```
const VIEWPORT_HEIGHT = 1080;
const VIEWPORT_WIDTH = 720;

const SCREEN_VIEWPORT_RATIO = window.innerHeight / VIEWPORT_HEIGHT;

canvas.setAttribute('height', window.innerHeight);
canvas.setAttribute('width', VIEWPORT_WIDTH *
SCREEN_VIEWPORT_RATIO);
```

This will ensure the canvas is set to the same height as the screen, and (importantly) that the width is scaled propotionally.

Resize the browser and refresh the page – you'll notice that the canvas resizes proportionally. The character, however, is always the same size.

Scaling Content on the Canvas

In the last section we introduced a new variable, `SCREEN_VIEWPORT_RATIO`. This tells us how much we should scale the elements we draw.

Replace your `drawImage` call with the following:

```
context.drawImage(
    img,
    25 * SCREEN_VIEWPORT_RATIO,
    25 * SCREEN_VIEWPORT_RATIO,
    128 * SCREEN_VIEWPORT_RATIO,
    128 * SCREEN_VIEWPORT_RATIO);
```

Now when the browser is resized and refreshed, the character is resized with it. Notice that we added two new parameters to the `drawImage` call — one for width and one for height.

Refactoring

It's going to be pretty annoying to have to write `SCREEN_VIEWPORT_RATIO` all the time, so let's build a new component and wrap our canvas element.

src/canvas.js

```
const VIEWPORT_HEIGHT = 1080;
const VIEWPORT_WIDTH = 720;
const SCREEN_VIEWPORT_RATIO = window.innerHeight / VIEWPORT_HEIGHT;

export class GameCanvas {
  constructor(canvasElement) {
    this.canvasElement = canvasElement;
    this.context = canvasElement.getContext('2d');

    canvasElement.setAttribute('height', window.innerHeight);
    canvasElement.setAttribute('width', VIEWPORT_WIDTH *
SCREEN_VIEWPORT_RATIO);

    this.context.fillStyle = '#e2fcbf';
    this.context.fillRect(0, 0, canvasElement.clientWidth,
canvasElement.clientHeight);
  }

  drawImage(image, x, y, width, height) {
    this.context.drawImage(
      image,
      x * SCREEN_VIEWPORT_RATIO,
      y * SCREEN_VIEWPORT_RATIO,
      width * SCREEN_VIEWPORT_RATIO,
      height * SCREEN_VIEWPORT_RATIO);
  }
}
```

As you can see, most of the code form `index.js` has now been refactored into `canvas.js`.

src/index.js

```
const assets = require('./assets.js');
import { GameCanvas } from './canvas.js';

const gameCanvas = new GameCanvas(document.getElementById('main-
canvas'));

assets.loadImage('images/green_blob.png').then(img => {
  gameCanvas.drawImage(img, 25, 25, 128, 128);
});
```

You can check that everything is working against the Github repository here.

Game Entities

Currently our game code loads an image and then draws it to the screen, but our character probably has more to him than that.

Let's sketch out a basic class structure for an entity in our game:

Entity

- load : Promise – this will be called to load any sounds or graphics associated with our entity
- draw : void – this will be called on each frame to draw the entity

- update : void – this will do any other per frame updates for the entity

In JavaScript that looks something like this:

src/entity.js

```
export class Entity {
  load() {
    throw new TypeError('Abstract class "Entity" cannot be
instantiated directly.');
  }

  draw(_canvas) {
    throw new TypeError('Abstract class "Entity" cannot be
instantiated directly.');
  }

  update() {
    throw new TypeError('Abstract class "Entity" cannot be
instantiated directly.');
  }
}
```

We can then create a monster class that implements these methods:

src/monster.js

```
const assets = require('./assets.js');
import { Entity } from './entity.js';

export class Monster extends Entity {
  load() {
    return assets.loadImage('images/green_blob.png').then(img => {
      this.image = img;
    });
  }

  draw(canvas) {
    canvas.drawImage(this.image, 25, 25, 128, 128);
  }

  update() {
    // do nothing yet
  }
}
```

Our main game code now looks like this:

src/index.js

```
const assets = require('./assets.js');
import { GameCanvas } from './canvas.js';
import { Monster } from './monster.js';

const gameCanvas = new GameCanvas(document.getElementById('main-
canvas'));

const monster = new Monster();
monster.load().then(() => {
  monster.draw(gameCanvas);
});
```

The Render Loop

Introducing the Game Loop

Once upon a time, games basically looked like a simple while loop.

Don't actually write this anywhere

```
while (!exit) {
    update();
    draw();
}
```

Games have evolved a lot since those times, and now the game loop can actually get very complicated. In JavaScript, we have `requestAnimationFrame` which allows the game to request to repaint the screen.

But first: why do we need to draw every frame? Simply speaking: we probably want our game to move. Rather than animating objects using HTML attributes you might be familiar with, it is more efficient to redraw our canvas each frame. The reason for this is that normal HTML elements in documents contain complex layout behavior that causes the browser to have to do a lot of recalculation whenever something moves. In games, we are handling our own movement and layout, so we don't need all that extra computation slowing us down.

The Game Lifecycle

Our game is going to have three distinct stages:

- loading
- playing
- exiting (in reality we don't need to do much here)

Let's think about loading:

We have the load method on our entity already. We probably want to load all our entities at the start of the game, before anything happens.

First let's create a new array representing all the entities in our game.

```
const monster = new Monster();
const entities = [ monster ];
```

Now, let's use the `Promise.all` method to make wait for all our entities load:

```
Promise.all(
  entities.map(entity => entity.load())
).then(() => {
  // ...
});
```

Finally we can draw them to the screen:

```
Promise.all(
  entities.map(entity => entity.load())
).then(() => {
  entities.forEach(entity => {
    entity.draw(gameCanvas);
  });
});
```

Looping

This is functionally the same as what we had before, but we want to do this every frame. So instead of just drawing everything straight away let's do this:

```
Promise.all(
  entities.map(entity => entity.load())
).then(() => {
  requestAnimationFrame(drawGame);
});

function drawGame() {
  entities.forEach(entity => {
    entity.draw(gameCanvas);
  });
  requestAnimationFrame(drawGame);
}
```

You can't see the difference yet, because nothing is moving. So how do we do that?

Movement and Animation

Moving

I'm envisaging some sort of tower defense game here, so let's make our monster move toward the bottom of the screen.

On each frame, we want to move the character. Fortunately, we already created the update method for this. Update the update method in the `monster.js` file to look like the following:

```
  update() {
    this.yPosition = this.yPosition + 1;
  }
```

Let's also add a constructor so that the `yPosition` has a default value:

```
  constructor() {
    super();
    this.yPosition = 25;
  }
```

... and let's make sure we're drawing the character at the right place ...

```
  draw(canvas) {
    canvas.drawImage(this.image, 25, this.yPosition, 128, 128);
  }
```

We need to make sure we're calling `update`, so in `index.js` do this:

```
Promise.all(
  entities.map(entity => entity.load())
).then(() => {
  requestAnimationFrame(gameLoop);
});

function gameLoop() {
  entities.forEach(entity => {
    entity.update();
  });
  entities.forEach(entity => {
    entity.draw(gameCanvas);
  });
  requestAnimationFrame(gameLoop);
}
```

When you refresh now, you should see the monster move slowly down the screen. But not everything is right here.

Cleaning Up

As the monster moves, you'll likely see a trail behind it. This is because we're not actually clearing the canvas on each frame, we're just drawing on top of whatever was there before.

To address this, let's add a `clear` method to the canvas:

```
  clear() {
    this.context.fillStyle = '#e2fcbf';
    this.context.fillRect(
      0,
      0,
      this.canvasElement.clientWidth,
      this.canvasElement.clientHeight);
  }
```

And make sure we call it in `index.js`:

```
function gameLoop() {
  entities.forEach(entity => {
    entity.update();
  });
  gameCanvas.clear();
  entities.forEach(entity => {
    entity.draw(gameCanvas);
  });
  requestAnimationFrame(gameLoop);
}
```

If you're following along, here's the Github commit for our current state.

Timing

Currently we add one to the `y` position of the monster on each frame, but different browsers on different devices may well run at different framerates.

To get around that, we can measure how much time passes between each frame. Each frame, we will check the time and see how many seconds have passed since the previous frame:

```
let time = 0;

Promise.all(
  entities.map(entity => entity.load())
).then(() => {
  time = Date.now();
  requestAnimationFrame(gameLoop);
});
```

```
function gameLoop() {
  const newTime = Date.now();
  const elapsedTimeInSec = (newTime - time) / 1000;
  time = newTime;
  entities.forEach(entity => {
    entity.update(elapsedTimeInSec);
  });
  gameCanvas.clear();
  entities.forEach(entity => {
    entity.draw(gameCanvas);
  });
  requestAnimationFrame(gameLoop);
}
```

Of course our monster and entity base class both need to be updated for this change:

In src/entity.js

```
  update(_elapsedSec) {
    throw new TypeError('Abstract class "Entity" cannot be
instantiated directly.');
  }
```

In src/monster.js

```
  update(elapsedSec) {
    this.yPosition = this.yPosition + 10 * elapsedSec;
  }
```

Adding Some Life

The monster currently proceeds down the screen at a pretty consistent rate, but it isn't very lifelike. Since the sprite isn't animated in any way, we can add some life by messing with basic transforms like scale.

In order to this, we can add an additional `scale` parameter to the `drawImage` function in `canvas.js`. A naive implementation might look like this:

```
drawImage(image, x, y, width, height, scale = 1.0) {
  this.context.drawImage(
    image,
    x * SCREEN_VIEWPORT_RATIO,
    y * SCREEN_VIEWPORT_RATIO,
    width * SCREEN_VIEWPORT_RATIO * scale,
    height * SCREEN_VIEWPORT_RATIO * scale);
}
```

The problem here is that the scaling operation is based on the same coordinate system that we're drawing from, so the image will only scale to the right and bottom, instead of uniformly from the center.

In order to make this happen, we need to offset the image by the difference between the scaled size and the original size.

```
drawImage(image, x, y, width, height, scale = 1.0) {
  const scaledWidth = width * scale;
  const scaledHeight = height * scale;
  this.context.drawImage(
    image,
    (x - (scaledWidth - width) / 2) * SCREEN_VIEWPORT_RATIO,
    (y - (scaledHeight - height) / 2) * SCREEN_VIEWPORT_RATIO,
    scaledWidth * SCREEN_VIEWPORT_RATIO,
    scaledHeight * SCREEN_VIEWPORT_RATIO);
}
```

We can animate scale a similar way to how we animated the `yPosition`:

```
update(elapsedSec) {
  this.yPosition = this.yPosition + 10 * elapsedSec;
  this.scale = this.scale + 0.1 * elapsedSec;
}
```

Implementing this, we see the monster gets larger and larger, eternally, but what would be better here is if the monster pingponged between a small and a large size.

If you remember your trigonometry, you might remember the shape of a sine wave. Sine waves are great for use in animation, because they have inbuilt easing, which avoids sharp transitions and looks natural to the human eye.

The general form of a sine function is:

$y = a \sin(b(x-h)) + k$

In this case, *y* will be our scale. *k* will be our base scale, which is 1 (a regular size). *x* is going to represent *time*, the total time the monster has been alive.

Now we can get experimental:

a controls the amplitude of the sine function, which defines how strong the scaling effect should be. After some experimentation, I settled on a very modest 0.05.

b controls the speed of the scaling. I used 2 for this. You should feel free to play around and see what feels right.

To implement animation with a sine wave, we can use the default functions already present in JavaScript:

```
update(elapsedSec) {
  this.yPosition = this.yPosition + 10 * elapsedSec;

  this.totalTime += elapsedSec;
  this.scale = 0.05 * Math.sin(2 * this.totalTime) + 1.0;
}
```

The code for this is here.

Responding to Input

Building a Wall

So far we have a monster that moves inoxerably toward the bottom of the screen (and far beyond). It would be nice if we had a way to stop him.

One way of doing this might be to add a wall entity that can be placed by the player.

Here is my wall class:

src/wall.js

```
const assets = require('./assets.js');
import { Entity } from './entity.js';

export class Wall extends Entity {
  constructor(x, y) {
    super();
    this.xPosition = x;
    this.yPosition = y;
  }

  load() {
    return assets.loadImage('images/wall.png').then(img => {
      this.image = img;
    });
  }

  draw(canvas) {
    canvas.drawImage(this.image, this.xPosition, this.yPosition,
128, 108);
  }

  update(elapsedSec) {
  }
}
```

To confirm that things are working, we can add a test wall into our `index.js` file:

```
const wall = new Wall(25, 150);

const entities = [ monster, wall ];
```

Of course, the monster doesn't stop when it hits the wall. We'll get to that later!

User Interface

To enable the user to place our wall, we might need a bit of UI. To this end, let's create another entity that exists for this purpose:

src/toolPalette.js

```
const assets = require('./assets.js');
import { Entity } from './entity.js';

export class ToolPalette extends Entity {
  constructor() {
    super();
  }

  load() {
    return assets.loadImage('images/wall.png').then(img => {
      this.wallToolImage = img;
    });
  }

  draw(canvas) {
    canvas.fillRect('#ffffff', 0, canvas.height - 130, canvas.width,
130);
    canvas.fillRect('#333333', 0, canvas.height - 133, canvas.width,
3);
```

```
    canvas.drawImage(
      this.wallToolImage,
      canvas.width / 2 - 128 - 8,
      canvas.height - 120,
      128,
      108);
  }

  update(elapsedSec) {
  }
}
```

A few modifications to `canvas.js` make this possible:

```
  fillRect(style, x, y, width, height) {
    this.context.fillStyle = style;
    this.context.fillRect(
      x * SCREEN_VIEWPORT_RATIO,
      y * SCREEN_VIEWPORT_RATIO,
      width * SCREEN_VIEWPORT_RATIO,
      height * SCREEN_VIEWPORT_RATIO);
  }

  get height() {
    return VIEWPORT_HEIGHT;
  }

  get width() {
    return VIEWPORT_WIDTH;
  }
```

You can also replace the test wall with the ToolPalette in `index.js`:

```
const entities = [ monster, new ToolPalette() ];
```

Mouse Events

We want to enable the user to drag the wall out onto the battlefield. For that we need to be able to tell when the mouse is down, or if the user is touching the screen.

Let's introduce a helper class:

src/inputHelper.js

```
let __Instance = null;

export class InputHelper {
  constructor(canvas) {
    this._isMouseDown = false;
    this._mouseX = 0;
    this._mouseY = 0;

    canvas.htmlElement.addEventListener('mousedown', () => {
      this._isMouseDown = true;
    });

    canvas.htmlElement.addEventListener('mouseup', () => {
      this._isMouseDown = false;
    });

    canvas.htmlElement.addEventListener('mousemove', e => {
      this._mouseX = (e.pageX - canvas.htmlElement.offsetLeft) /
canvas.viewportRatio;
      this._mouseY = (e.pageY - canvas.htmlElement.offsetTop) /
canvas.viewportRatio;
    });

    __Instance = this;
  }

  static get instance() {
    return __Instance;
  }
  get mouseX() {
    return this._mouseX;
  }

  get mouseY() {
    return this._mouseY;
  }

  get isMouseDown() {
    return this._isMouseDown;
  }
}
```

`InputHelper` is a singleton class, which is rarely best practice but in the scenario where we're managing input, makes sense.

We can initialize the `InputHelper` in `index.js`:

```
new InputHelper(gameCanvas);
```

and use it anywhere by including it:

```
import { InputHelper } from './inputHelper.js';
console.log('x =', InputHelper.instance.mouseX, 'y =',
InputHelper.instance.mouseY);
```

Specifically we want to use in the update event of the tool palette as follows:

```
  update(_elapsedSec) {
    if (_mouseInBounds(this.wallToolBounds)) {
      if (!this.wasMouseDown && InputHelper.instance.isMouseDown) {
        this.isPlacingWall = true;
        this.wallGrabPos = {
          x: InputHelper.instance.mouseX - this.wallToolBounds.x,
          y: InputHelper.instance.mouseY - this.wallToolBounds.y
        };
      }
    }
    if (this.isPlacingWall && this.wasMouseDown && !
InputHelper.instance.isMouseDown) {
      this.isPlacingWall = false;
    }
    this.wasMouseDown = InputHelper.instance.isMouseDown;
  }
```

I've made a couple of other changes here, to make our lives easier. Specifically, the constructor of the tool palette now defines the bounds of the wall tool and defaults some of our other new variables:

```
  constructor(canvas) {
    super();

    this.isPlacingWall = false;
    this.wasMouseDown = false;

    this.wallToolBounds = {
      x: canvas.width / 2 - 128 - 8,
      y: canvas.height - 120,
      width: 128,
      height: 108
    };
  }
```

I'm also using a new helper function called `_mouseInBounds`:

```
function _mouseInBounds(bounds) {
  return InputHelper.instance.mouseX > bounds.x &&
    InputHelper.instance.mouseX < bounds.x + bounds.width &&
    InputHelper.instance.mouseY > bounds.y &&
    InputHelper.instance.mouseY < bounds.y + bounds.height;
}
```

And finally the `draw` function of the tool palette is updated to use the new bounds, and to draw the wall as it is dragged:

```
  draw(canvas) {
    canvas.fillRect('#ffffff', 0, canvas.height - 130, canvas.width,
130);
    canvas.fillRect('#333333', 0, canvas.height - 133, canvas.width,
3);
    canvas.drawImage(
      this.wallToolImage,
      this.wallToolBounds.x,
      this.wallToolBounds.y,
      this.wallToolBounds.width,
      this.wallToolBounds.height);

    if (this.isPlacingWall) {
      canvas.drawImage(
        this.wallToolImage,
        InputHelper.instance.mouseX - this.wallGrabPos.x,
        InputHelper.instance.mouseY - this.wallGrabPos.y,
        this.wallToolBounds.width,
        this.wallToolBounds.height);
    }
  }
```

You can follow along where we are now on Github.

Placement

We can drag walls now, but when dropped they just vanish, which isn't ideal. We want to make sure they actually place correctly, ideally on a grid.

To achieve this, let's introduce a new class:

src/gameplayGrid.js

```
const assets = require('./assets.js');
import { Entity } from './entity.js';

const COLUMN_SIZE = 120;
const ROW_SIZE = 120;

export class GameplayGrid extends Entity {
  constructor() {
    super();
    this.wallLocations = [];
  }

  load() {
    return assets.loadImage('images/wall.png').then(img => {
      this.wallImage = img;
    });
  }

  addWallPiece(x, y) {
    this.wallLocations.push({
      x: Math.floor(x / COLUMN_SIZE),
      y: Math.floor(y / ROW_SIZE)
    });
  }

  draw(canvas) {
    this.wallLocations.forEach(wallLocation => {
      canvas.drawImage(
        this.wallImage,
        wallLocation.x * COLUMN_SIZE - 4,
        wallLocation.y * ROW_SIZE + 6,
        128,
        108
      )
    });
  }

  update(_elapsedTime) {
    // Do nothing
  }
}
```

This gameplay grid is responsible for deciding where walls should be positioned (and later, things other than walls).

To use it, we'll also want to make some updates to the tool palette's `update` method:

```
    if (this.isPlacingWall && this.wasMouseDown && !
InputHelper.instance.isMouseDown) {
      this.grid.addWallPiece(
        InputHelper.instance.mouseX,
        InputHelper.instance.mouseY
      );
      this.isPlacingWall = false;
    }
```

We also need to ensure that we're passing the grid into the tool palette's contructor in `index.js`:

```
const entities = [ monster, grid, new ToolPalette(gameCanvas,
grid) ];
```

You can follow along on Github here.

Strict Types and Flow

At this point we are starting to have quite a complex ecosystem of classes. Flow, a tool developed at Facebook, enables us to specify types in JavaScript. This can help find errors instead of us catching them at runtime, especially as our game becomes more complicated.

To install Flow, execute the following commands:

```
npm install --save-dev babel-cli babel-preset-flow flow-bin
```

```
.\node_modules\.bin\flow init
```

We also need to create a `.babelrc` file for stripping out the Flow type information from the final JavaScript (since browsers won't recognize it):

.babelrc

```
{
  "presets": ["flow"]
}
```

Flow will only check files with `// @flow` at the top. I've gone ahead and done this to the entire project so that we get the benefit of Flow typechecking going forward.

To see any current Flow errors in the project run the `flow` command:

```
.\node_modules\.bin\flow
```

Flow is up and working correctly here.

Simple Collisions

Spawning Monsters

Ideally we want our monster to stop when he reaches a wall – maybe chew on it for a time. In order to do that, the monster needs to have awareness of the grid.

We can do that by making the grid responsible for the management of monsters.

First, let's give the Monster class an `xPosition` attribute:

```
  xPosition: number;
...
```

```
  constructor(xPos: number) {
    super();
...
  draw(canvas: GameCanvas) {
    canvas.drawImage(
      this.image,
      this.xPosition,
      this.yPosition,
      128,
      128,
      this.scale);
  }
```

Now let's add some new monster management methods to the gameplay grid:

```
  spawnMonster() {
    const column = Math.floor(Math.random() * NUM_COLUMNS);
    const monster = new Monster(column * COLUMN_SIZE);
    monster.load().then(() => this.monsters.push(monster));
  }
```

We also need to make sure that all the monsters that the gameplay grid has in it are updated and drawn at every frame:

```
  draw(canvas: GameCanvas) {
    this.monsters.forEach(monster => monster.draw(canvas));
...
  update(elapsedTime: number) {
    this.monsters.forEach(monster => monster.update(elapsedTime));
  }
```

We don't want our monster to be in the game from the start, anymore. Instead, we can spawn it using the new `spawnMonster` method in `index.js`:

```
const grid = new GameplayGrid();
const entities = [ grid, new ToolPalette(gameCanvas, grid) ];
```

```
grid.spawnMonster();
```

Note that each time the page is refreshed, the monster spawns in a random column.

Hitting the Wall

Now that the monsters are managed by the grid, which also knows where the walls are, we can introduce some simple collision logic:

First, let's give our monster a state, so we know if it's attacking or descending:

```
type State = 'DESCENDING' | 'ATTACKING';

export default class Monster extends Entity {
  xPosition: number;
  yPosition: number;
  scale: number;
  totalTime: number;
  image: Image;
  state: State;
```

Note that `type State =` syntax – this is our way of recreating in Flow what in other languages we would call an enum. Flow will ensure that `state` is always one of those string values.

We can also write the logic to see if the monster is at the wall:

```
  testWallCollision(bounds: Bounds) {
    const monsterBounds = {
      x: this.xPosition,
      y: this.yPosition,
      width: 128,
      height: 128
    };
    if (doBoundsIntersect(bounds, monsterBounds)) {
      this.state = 'ATTACKING';
    }
  }
```

This uses a new utility method `doBoundsIntersect`:

```
export function doBoundsIntersect(a: Bounds, b: Bounds) {
    return !(
      b.x > a.x + a.width ||
      b.x + b.width < a.x ||
      b.y > a.y + a.height ||
      b.y + b.height < a.y);
}
```

Once we update the `update` method of the monster:

```
  update(elapsedSec: number) {
    if (this.state === 'DESCENDING') {
      this.yPosition = this.yPosition + 10 * elapsedSec;
    }

    this.totalTime += elapsedSec;
    this.scale = 0.05 * Math.sin(2 * this.totalTime) + 1.0;
  }
```

... and ensure that we are checking for collisions in the gameplay grid:

```
  update(elapsedTime: number) {
    this.monsters.forEach(monster => {
      monster.update(elapsedTime);

      this.wallLocations.forEach(wallLocation => {

monster.testWallCollision(this._getBoundingBoxForWallLocation(wallLo
cation));
      });
    });
  }
```

…monsters will stop when they reach walls. They aren't actually attacking them, for now. That's going to require us to get a bit smarter about how our walls work.

The current state of the project can be perused here.

Down with the walls

Currently `GameplayGrid` is entirely responsible for drawing walls, but they should really be their own entity type, similar to how monster is nested under the grid.

src/wall.js

```
// @flow

import assets from './assets.js';
import Entity from './entity.js';
import GameCanvas from './canvas.js';
import { type Bounds } from './mathTypes.js';

export default class Monster extends Entity {
  xPosition: number;
  yPosition: number;
  underAttack: boolean;
  destroyed: boolean;
  attackTime: number;
  hits: number;
  image: Image;

  constructor(xPosition: number, yPosition: number) {
    super();
    this.xPosition = xPosition;
    this.yPosition = yPosition;
    this.underAttack = false;
    this.attackTime = 0;
    this.hits = 3;
  }
```

```
load() {
  return assets.loadImage('images/wall.png').then(img => {
    this.image = img;
  });
}

draw(canvas: GameCanvas) {
  if (this.destroyed) {
    return;
  }
  canvas.drawImage(
    this.image,
    this.xPosition,
    this.yPosition,
    128,
    108);
}

update(_elapsedSec: number) {
}

attack(hits: number) {
  this.hits -= hits;
  if (this.hits <= 0) {
    this.destroyed = true;
  }
}

get boundingBox() {
  // Return a bounding box that is slightly smaller than the wall
  // to make sure that passing monsters don't get distracted!
  return {
    x: this.xPosition + 20,
    y: this.yPosition + 24,
    width: 80,
    height: 80
  };
}
```

```
  get isDestroyed() {
    return this.destroyed;
  }
}
```

This requires us to make a few corresponding changes to `GameplayGrid`:

src/gameplayGrid.js

```
// @flow

import assets from './assets.js';
import Entity from './entity.js';
import GameCanvas from './canvas.js';
import Monster from './monster.js';
import Wall from './wall.js';
import { type Bounds } from './mathTypes.js';

const COLUMN_SIZE = 120;
const ROW_SIZE = 120;
const NUM_COLUMNS = 6;

export default class GameplayGrid extends Entity {
  monsters: Array<Monster>;
  walls: Array<Wall>;

  constructor() {
    super();
    this.walls = [];
    this.monsters = [];
  }

  load() {
  }

  addWallPiece(x: number, y: number) {
    const column = Math.floor(x / COLUMN_SIZE);
    const row = Math.floor(y / ROW_SIZE);
    const wall = new Wall(
      column * COLUMN_SIZE - 4,
      row * ROW_SIZE + 6);
    wall.load().then(() => this.walls.push(wall));
  }

  spawnMonster() {
    const column = Math.floor(Math.random() * NUM_COLUMNS);
    const monster = new Monster(column * COLUMN_SIZE);
    monster.load().then(() => this.monsters.push(monster));
  }

  draw(canvas: GameCanvas) {
    this.monsters.forEach(monster => monster.draw(canvas));
    this.walls.forEach(wall => wall.draw(canvas));
  }

  update(elapsedTime: number) {
    this.monsters.forEach(monster => {
      monster.update(elapsedTime);

      this.walls.forEach(wall => {
        if (!wall.isDestroyed) {
          monster.testWallCollision(wall);
        }
      });
    });
  }
}
```

We also need to tell our monster which wall it is attacking:

```
  testWallCollision(wall: Wall) {
    if (this.state === 'ATTACKING') {
      return;
    }
    const monsterBounds = {
      x: this.xPosition,
      y: this.yPosition,
      width: 128,
      height: 128
    };
    if (doBoundsIntersect(wall.boundingBox, monsterBounds)) {
      this.state = 'ATTACKING';
      this.target = wall;
      this.lastAttackTime = this.totalTime;
    }
  }
```

Finally, when the monster is attacking, we need to actually do damage to the wall, and once it's destroyed it's time to keep going:

```
  update(elapsedSec: number) {
    const target = this.target;

    if (this.state === 'DESCENDING') {
      this.yPosition = this.yPosition + 10 * elapsedSec;
    } else if (this.state === 'ATTACKING' && target) {
      if (target.isDestroyed) {
        this.state = 'DESCENDING';
      } else if (this.totalTime - this.lastAttackTime > 1) {
        target.attack(1);
        this.lastAttackTime = this.totalTime;
      }
    }

    this.totalTime += elapsedSec;
    this.scale = 0.05 * Math.sin(2 * this.totalTime) + 1.0;
  }
```

Now we should have a system where a monster reaches a wall, attacks it for about three seconds, then continues.

It doesn't *look* very exciting, [but here it is].

Special Effects

Ghostly Walls

When we are dragging the walls into place, they look like any other wall. It could be confusing, since these walls haven't been placed yet, so they cannot be attacked.

To make it clearer to the player what's going on, let's apply some special effects to the walls.

The 2D context provides the `globalCompositeOperation` attribute, which enables us to use compositioning operations like lighten. To see what this looks like, let's add a new option to our draw image call in `canvas.js`:

```
drawImage(
  image: Image,
  x: number,
  y: number,
  width: number,
  height: number,
  scale: number = 1.0,
  compositeOperation: string = 'source-over'
) {
  const scaledWidth = width * scale;
  const scaledHeight = height * scale;
  this.context.save();
  this.context.globalCompositeOperation = compositeOperation;
  this.context.drawImage(
    image,
    (x - (scaledWidth - width) / 2) * SCREEN_VIEWPORT_RATIO,
    (y - (scaledHeight - height) / 2) * SCREEN_VIEWPORT_RATIO,
    scaledWidth * SCREEN_VIEWPORT_RATIO,
    scaledHeight * SCREEN_VIEWPORT_RATIO);
  this.context.restore();
  }
```

The addition of **context.save** and **context.restore** help ensure that we aren't messing up any other images when we change the **globalCompositionOperation**.

Let's test this out, by changing the **draw** function of in **toolPalette.js**:

```
  draw(canvas: GameCanvas) {
    canvas.fillRect('#ffffff', 0, canvas.height - 130, canvas.width,
130);
    canvas.fillRect('#333333', 0, canvas.height - 133, canvas.width,
3);
    canvas.drawImage(
      this.wallToolImage,
      this.wallToolBounds.x,
      this.wallToolBounds.y,
      this.wallToolBounds.width,
      this.wallToolBounds.height);

    if (this.isPlacingWall) {
      canvas.drawImage(
        this.wallToolImage,
        InputHelper.instance.mouseX - this.wallGrabPos.x,
        InputHelper.instance.mouseY - this.wallGrabPos.y,
        this.wallToolBounds.width,
        this.wallToolBounds.height,
        1.0,
        'hard-light');
    }
  }
```

I think **hard-light** looks great here, but you should feel free to mess around. Mozilla's documentation covers the different options available.

Opacity

It's also not very obvious when a wall is being attacked. One way we could communicate the strength of the wall is by lowering it's opacity as it loses health.

We can add this line to **`canvas.js`** to enable drawing images with opacity:

```
this.context.globalAlpha = opacity;
```

... and this to the wall code ...

```
  update(elapsedSec: number) {
    const targetOpacity = this.hits * 0.33;

    if (this.opacity > targetOpacity) {
      this.opacity -= 0.5 * elapsedSec;
    }
  }
```

I also tightened up the attack sequence to correspond with the monster's scale transformation:

```
      } else if (this.totalTime - this.lastAttackTime > Math.PI) {
        target.attack(1);
        this.lastAttackTime = this.totalTime;
      }
```

You can see all the changes required to make this happen here.

Winning and Losing

This is all well and good, but there's not a lot of game here right now. There is one key thing our game needs: a losing condition.

Since this is a tower defense, there's no real winning per se (though you should feel free to add a winning condition as well – perhaps a timer or a number of enemies defeated), but it is important than when a monster reaches the bottom of the screen, you lose.

Losing

We want the player to lose when the monster reaches the bottom of the screen. To do this, let's add another method to our monster:

```
  get yPos() {
    return this.yPosition;
  }
```

This will return a value with the y position of our monster. We can then check it in `gameplayGrid.js`:

```
const NUM_ROWS = 8;
...
  gameState: 'PLAYING' | 'LOST' = 'PLAYING';
...
    if (this.gameState === 'LOST') {
      return;
    }

    this.monsters.forEach(monster => {

      monster.update(elapsedTime);

      if (monster.yPos > NUM_ROWS * ROW_SIZE) {
        this.gameState = 'LOST';
      }

      this.walls.forEach(wall => {
        if (!wall.isDestroyed) {
          monster.testWallCollision(wall);
        }
      });
    });
    this.walls.forEach(wall => wall.update(elapsedTime));
  }
```

Drawing Text

WHen the game is lost, we should probably let the player know, so they can stop playing. It would likely look better if we designed a graphic for this, but since we're art constrained, let's look at the canvas APIs for drawing text.

Based on the capabilities here, we can add the following method to our `canvas.js` file:

```
  writeText(
    text: string,
    font: string,
    color: string,
    alignment: string,
    x: number,
    y: number,
    width: number
  ) {
    this.context.fillStyle = color;
    this.context.font = font;
    this.context.textAlign = alignment;
    this.context.fillText(
      text,
      x * SCREEN_VIEWPORT_RATIO,
      y * SCREEN_VIEWPORT_RATIO,
      width * SCREEN_VIEWPORT_RATIO);
  }
```

We can then call it in the `draw` method of our `gameplayGrid.js` file:

```
    if (this.gameState === 'LOST') {
      canvas.writeText('You lose!', '72px sans-serif', '#111111',
'center', canvas.width / 2, canvas.height / 2);
    }
```

Subtle? Not really, but it works!

Here's the latest code, if you're following along.

Extending what we've built

I really hope this is a useful starting point for you to be able to create game of your own, but it's really pretty bare bones for the moment. Here are some ideas for extensions you could make to what we have here:

- **Add health bars**: you know how to add text – add a health bar to the bottom of the screen. Make some changes to `gameGrid.js` so you can survive three (or more) monster collisions.
- **Create a weapon**: use some of the code we've written in `toolPalette.js` to create some kind of weapon. You know how to make things move, and how to do collisions! Add a projectile that can defeat the monsters.
- **More enemy types**: currently we have this one green blob that moves at a constant speed. Experiment with new enemy types. Review the Special Ef-

fects chapter to see if it's possible to use the same asset with a different look, or create your own original art.

Conclusion

Throughout this guide, hopefully you've learned some of the basics of developing HTML5 games.