Getting Started with Cocos2d-HTML5

Applies To

- BlackBerry Dev Alpha 10.0.9.1675
- BlackBerry 10 WebWorks SDK 1.0.4.5

Background

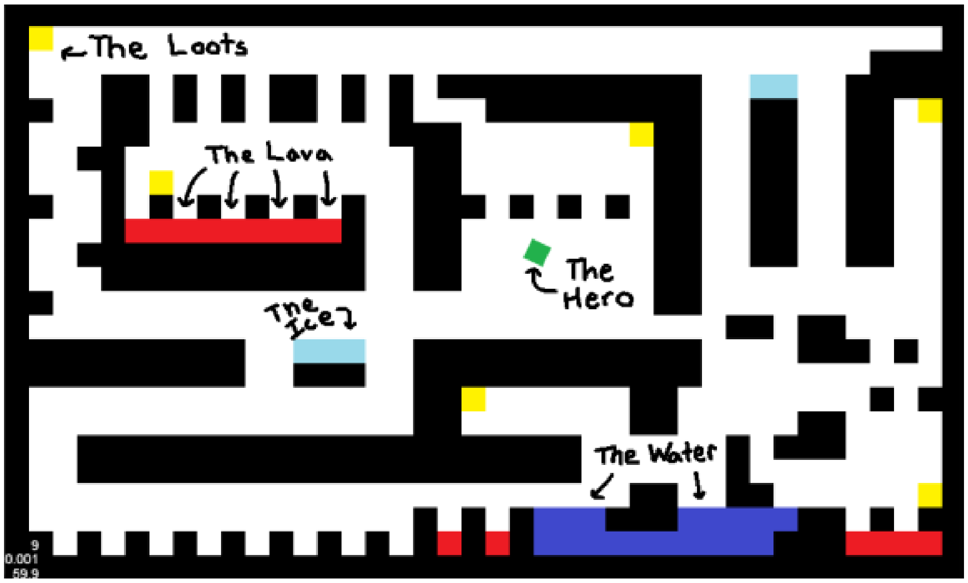
This guide is intended to help developers get started on their Canvas2D game development using the HTML5 port of the Cocos2d-x framework; Cocos2d-HTML5. While the ultimate target is the BlackBerry 10 platform, the concepts discussed in this guide can easily be adapted on a larger scale.

Cocos2d-HTML5 simplifies the game development process for developers by providing a framework for animation, user input,

This guide will use the BoxQuest sample available on Github as a reference, guiding the developer through the various stages of development:

- Environment configuration.
- Creating a Cocos2d-HTML5 skeleton.
- Leveraging sprites and tile maps.
- Physics with Box2DWeb.
- Virtual controls with Freewill.js.

The final result will be the beginnings of a 2D platformer where you control our Hero on a quest to collect the loot.



Environment Configuration

In order to test our application as it is being developed, we will need to ensure we have the appropriate tools installed:

- Google Chrome web browser.
- The Ripple emulator for permitting BlackBerry 10 functionality in the browser.
- The BlackBerry 10 WebWorks SDK for packaging HTML5 applications.
- WampServer, or similar local web server, to access our project through a web browser.

Install Google Chrome

It is important that you are using Google Chrome since we will rely on a Chrome extension (the Ripple emulator.) Google Chrome can be downloaded here.

Install the Ripple Emulator

Though the Ripple emulator is a Chrome extension and is available directly through the Chrome Web Store, please refer to the Installing the Ripple emulator documentation for the most recent, stable version of Ripple.

The Ripple Emulator will enable your Google Chrome to emulate the screen dimensions for a multitude of devices on the BlackBerry, iOS, and Android platforms. In addition, the Ripple Emulator can provide platform specific functionality for the BlackBerry (Smartphone, Tablet, and BlackBerry 10) platforms as well as PhoneGap/Cordova API functionality.

Install the BlackBerry 10 WebWorks SDK

The BlackBerry WebWorks SDK is not a required component; however it will enable developers to package their HTML5 (including Cocos2d-HTML5) projects into a standalone application for the BlackBerry PlayBook and BlackBerry 10 platforms.

Packaged applications run natively on the device and do not inherently require an active internet connection in order to run; in comparison to a web app that a device must actually browse to. The BlackBerry WebWorks SDKs can be downloaded here.

Later in this guide, we will leverage both the Ripple Emulator and BlackBery 10 WebWorks SDK to package and deploy our sample application to a BlackBerry 10 Dev Alpha.

Install WampServer

As mentioned, any web server will do, however for this guide we'll focus on a WampServer installation. Having a web server will allow us to host content locally and access that content through Google Chrome and the Ripple emulator.

WampServer can be downloaded here and should be installed to the default *C:\wamp* installation folder. Once installed, you may need to manually launch WampServer via *Start > WampServer > start WampServer*.

When WampServer is running, you can find the icon in your system tray. At this point, navigating to *localhost* in your web browser should produce the default WampServer welcome screen.

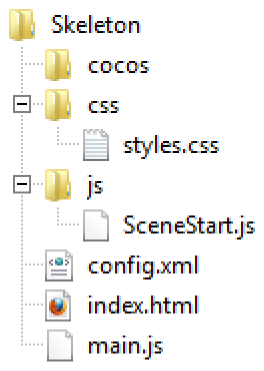
WampServer works right out of the box but is initially configured to only allow access from your local PC. In the event that we want to access the server from a test device, we'll need to modify the configuration slightly.

- Left-click and select *Apache > httpd.conf*.
- Locate the *<Directory "c:/wamp/www/">* element.
- Directly before the closing tag for this element, add *Allow from all*.
- Save and close *httpd.conf*.
- Left-click and select *Restart All Services*.

This will enable access to your root web server folder by all devices that have access to your network. If you do not want this access enabled, the above modification can be skipped.

Creating a Cocos2d-HTML5 Skeleton

The skeleton for any Cocos2d-HTML5 application we create based on this guide will consist of the following file and folder structures. The project name (*Skeleton*) will vary from project to project.



Let's take a look at each of these individually.

config.xml

This is a BlackBerry specific file. When an HTML5 application is packaged into a WebWorks application, a *config.xml* is necessary to provide additional information regarding the name of the application, author, icon, permissions, etc. For our purposes, we will use the following, basic *config.xml*:

```xml
<?xml version="1.0" encoding="utf-8"?>
<widget xmlns="http://www.w3.org/ns/widgets"
        xmlns:rim="http://www.blackberry.com/ns/widgets"
        id="com.oros.boxquest"
        version="1.0.0.0">

    <name>BoxQuest</name>
    <author>Oros</author>
    <content src="index.html" />

    <feature id="blackberry.app.orientation">
        <param name="mode" value="landscape" />
    </feature>
</widget>
```

When creating your own project, there are a few values that need to be taken into consideration.

- `<widget id>` needs to be a unique identifier for your application. The reverse domain format reduces the chance of duplicating an already existing id.

- `<widget version>` needs to be incremented with each release of your application. The term *build id* refers to the last digit of the version. If you are packaging and signing an application for deployment / sale, you will need to increment the build id. The Ripple emulator automatically does this for you.
- `<content>` refers to the main entry point into our application; this will generally be *index.html*.
- `<feature id="blackberry.app.orientation">` specifies that we want our application to be locked to landscape orientation; this may vary based on the application.

For full details regarding the *config.xml* file, please refer to Creating a configuration document.

styles.css

This file is more a matter of personal preference, however in this sample application we will be filling the entire screen with our canvas. As such, we want to ensure that there is no unnecessary white-space surrounding our canvas.

```
body {
    background-color: black;
    cursor: default;
    font-family: sans-serif;
    height: 100%;
    margin: 0;
    padding: 0;
    width: 100%;
}
```

A few optional styles are also applied to the background, cursor, and font for our application. Finally, we indicate that we want the `body` to inhabit the full dimensions of the available screen.

index.html

Now we're getting into the true heart of the application. As indicated in the *config.xml* file, our *index.html* will serve as the main entry point into our application. We'll analyze this file in sections, but it can also be viewed in its entirety on Github.

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript">
            (function () {
                var meta = document.createElement('meta');
                meta.setAttribute('name', 'viewport');
                meta.setAttribute('content', 'initial-scale=' + (1.0 / window.devicePixelRatio) + ',user-scalable=no');
                document.getElementsByTagName('head')[0].appendChild(meta);
            }());
        </script>
        <link type="text/css" rel="stylesheet" href="css/styles.css" />
    </head>
    <body>
        <canvas id="ccCanvas"></canvas>
        <script type="text/javascript" src="local:///chrome/webworks.js"></script>
        <script type="text/javascript">
            function onwebworksready() {
                document.removeEventListener('webworksready', onwebworksready, false);

                var script = document.createElement('script');
                script.id = 'cocos2d-html5';
                script.c = {
                    box2d: false,
                    engineDir: './cocos/cocos2d/',
                    appFiles: [
                        './js/SceneStart.js'
                    ]
                };
                script.src = script.c.engineDir + 'platform/jsloader.js';
                document.ccConfig = script.c;
                document.body.appendChild(script);
            }

            function ondomcontentloaded() {
                window.removeEventListener('DOMContentLoaded', ondomcontentloaded, false);
                document.addEventListener('webworksready', onwebworksready, false);
            }

            window.addEventListener('DOMContentLoaded', ondomcontentloaded, false);
        </script>
    </body>
</html>
```

We start off by defining the standard HTML5 DOCTYPE and an opening our `<html>` element.

The next step is to define our head element, in which we will set our viewport and import our custom style sheet.

Due to the behavior of the `<meta>` viewport tag on various devices, we need to leverage a `<script>` to properly calculate our `initial-scale`. Full documentation on this approach can be found here.

Next, we'll actually start loading the content of our page.

All we're doing here is adding a unique `<canvas>` to our `<body>` and importing our **`webworks.js`** `<script>`. This `<script>` is specific to BlackBerry development and provides the JavaScript-to-native functionality of WebWorks. It is automatically packaged into our application at the noted location when we build our project, so this file does not actually exist as a part of our skeleton.

Following this we create our own custom `<script>` where we define our `onwebworksready` function; this function gets triggered only after WebWorks has completed its own initialization and after the `DOMContentLoaded` event. Essentially, at this point all of our initialization is complete, and we're ready to start loading the Cocos2d-HTML5 framework.

To load Cocos2d-HTML5, we create a `script` object and give it the following, required `id`: `cocos2d-html5`. We add a configuration (`c`) to the object which holds a few key values:

- `box2d` is a boolean variable that indicates whether we want to load Box2DWeb physics into our application. Even though we will use Box2DWeb eventually, we still keep this ***`false`*** as we will be importing our source file directly into our Web Worker separately.

- engineDir represents the path to where our Cocos2d-HTML5 framework is (or will be) located.
- appFiles represents an array [] of additional JavaScript files that we want to import into our application.

Finally we indicate the path to jsloader.js, a component of the Cocos2d-HTML5 framework and we add the new script object to our document. Doing so will automatically initiate the next phase of Cocos2d-HTML5 initialization; specifically the loading of *main.js*.

We noted though that onwebworksready only occurs *after* the DOMContentLoaded event. As such, we must define the following function to be triggered by DOMContentLoaded.

And finally, when our browser actually loads this <script>, it will also register our DOMContentLoaded event listener that, in turn, initiates our onwebworksready function and kicks off Cocos2d-HTML5 initialization.

To finish up, we close off our <body> and <html> elements. And with that, we now have a complete *index.html* file that is loaded as the main entry point into our application.

main.js

Within *index.html*, we defined a custom script object that injected jsloader.js into our document. The default behavior upon completion is to also load *main.js* into the document, which we must define ourselves.

```
/*global cc, SceneStart */

var ccApplication = cc.Application.extend({
    ctor: function () {
        this._super();
        this.startScene = SceneStart;

        cc.COCOS2D_DEBUG = 0;
        cc.initDebugSetting();

        cc.setup('ccCanvas', window.innerWidth, window.innerHeight);
        document.querySelector('#Cocos2dGameContainer').style.overflow = 'hidden';
        document.querySelector('#Cocos2dGameContainer').style.position = 'fixed';
        document.querySelector('#ccCanvas').style.position = 'fixed';

        cc.Loader.getInstance().onloading = function () {
            cc.LoaderScene.getInstance().draw();
        };

        cc.Loader.getInstance().onload = function () {
            cc.AppController.shareAppController().didFinishLaunchingWithOptions();
        };

        cc.Loader.getInstance().preload([
        ]);
    },

    applicationDidFinishLaunching: function () {
        var director = cc.Director.getInstance();
        director.setDisplayStats(true);
        director.runWithScene(new this.startScene());
        return true;
    }
});

var ccMain = new ccApplication();
```

For starters, we'll be extending the *cc.Application* class to define the actions of our Cocos2d-HTML5 application as it loads; this is all done within the ctor function.

The first thing you should always do when extending a Cocos2d-HTML5 class is to call that object's `super` function. We also set `this.startScene` equal to `SceneStart` however note that `SceneStart` hasn't been defined yet and will actually be defined in *SceneStart.js*.

Next, we'll set the debug level for our application, where `0` indicates no logging, `1` indicates minimal logging, and `2` indicates full logging.

Following this, we call the Cocos2d-HTML5 `setup` function on our `<canvas>`.

We've also set three additional CSS styles *after* calling the `setup` function to ensure that our `<canvas>` and containing `<div>` remain properly styled.

Next on the list, we define our `onloading` and `onload` functions, these are essentially the same across all Cocos2d-HTML5 applications. We then call the `preload` function with an empty `[]` since we don't currently have any assets to preload, and close off `ctor` function.

Once the `preload` function completes, it will automatically initiate the next phase of initialization: creating and presenting our scene.

To accomplish this, we need to define the `applicationDidFinishLaunching` function.

Everything up until the last line of code was merely the *definition* of a class, but we need a way to actually initiate the `ctor` function when this script loads. That happens on the very last line where we set our `ccMain` variable to a new `ccApplication` instantiation.

SceneStart.js

In *main.js*, we defined `this.startScene` to be equal to `SceneStart` which is being defined in *SceneStart.js*. Once *main.js* completes all of its preloading, it will then invoke `SceneStart.onEnter` and create a new instance of `LayerStart` to be added to our scene.

```
/*global cc */

var LayerStart = cc.Layer.extend({
    ctor: function () {
        this._super();
        return true;
    }
});

var SceneStart = cc.Scene.extend({
    onEnter: function () {
        this._super();
        this.addChild(new LayerStart());
    }
});
```

For now, *SceneStart.js* is pretty basic but will be growing as we add more content to our application. We define `LayerStart` as an extension of `Layer` and within its `ctor` function we do what we always do first when extending: call `this._super()`. For now, we're not adding content so all we do is `return true` to indicate success.

Similarly for our `SceneStart` variable, all we're doing is initializing through `this._super()` and then adding an instance of `LayerStart` to the scene.

cocos

You will notice that in our skeleton file structure, we have a folder called *cocos* that does not actually have any contents. This is where our Cocos2d-HTML5 framework will be stored for access by our WebWorks application; if you recall, we defined this location in *index.html*.

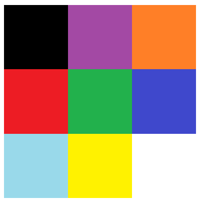
The actual framework can be downloaded from the Cocos2d-HTML5 Github repository.

In this Github repository are two folders that we will want to download and place inside the **cocos** folder, these are: **cocos2d** and **CocosDenshion**. With that, we've completed the final step in creating a skeleton Cocos2d-HTML5 application. If you navigate to your *Skeleton* project now in Ripple, you should see the following blank screen (don't worry, we'll fill it soon.)

Leveraging Sprites and Tile Maps

In the previous section, we focused on setting up a skeleton for Cocos2d-HTML5 that can serve as a starting point for our applications. The next step will be to populate our application with a background, a hero, and collectible loot.

Before we get started adding these resources, we'll need to download tiles.png from Github as that will be source for most of our graphics.

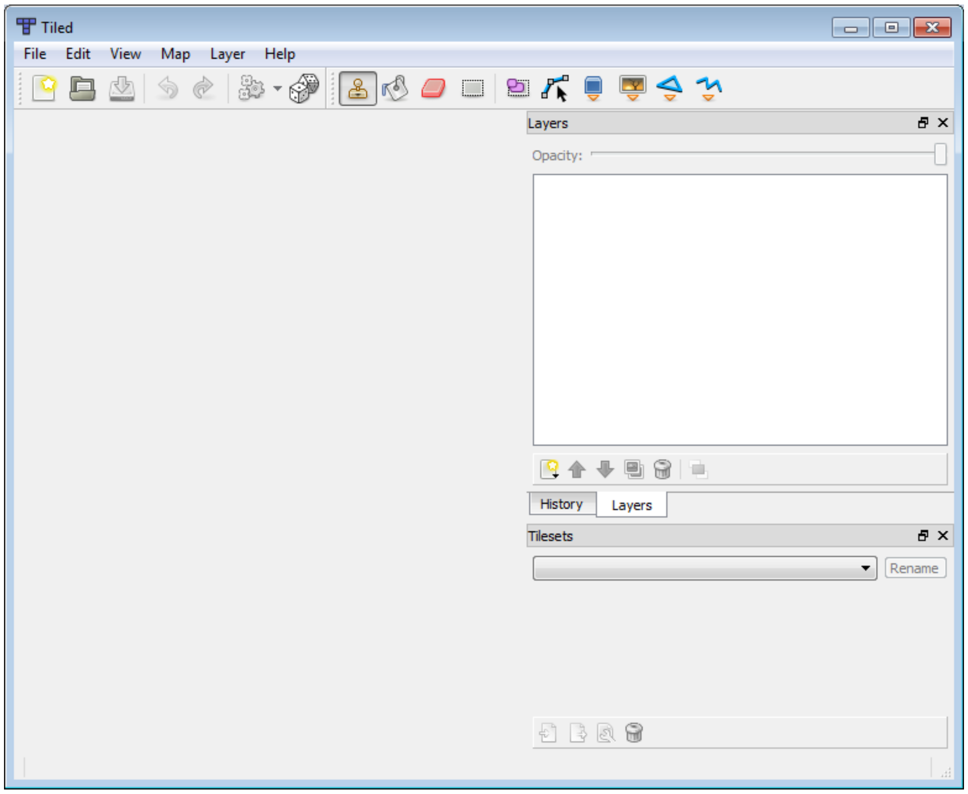


We will need to create the following folder and save *tiles.png* there: ...*Skeleton**images*

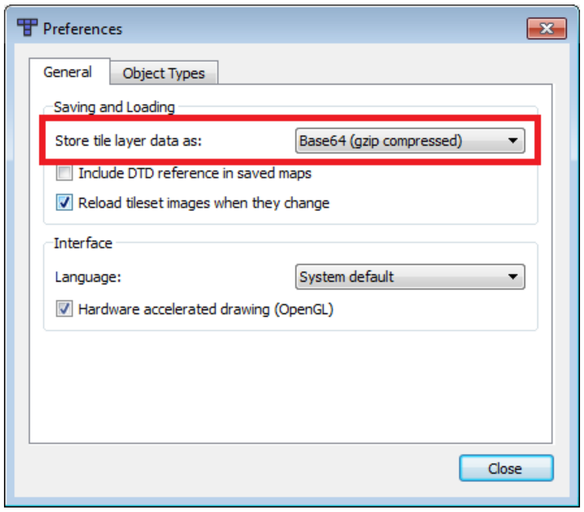
Creating a Tile Map with Tiled

Cocos2d-HTML5 supports the TMX tile map file type so we will use the software called Tiled to create our TMX resources. Tiled can be [downloaded here](#).

Once installed, launch Tiled and you should see the following screen.



The first thing we will want to do is ensure that our TMX files are being created in a format that Cocos2d-HTML5 will understand. From the top toolbar, navigate *Edit > Preferences* and ensure that *Base64 (gzip compressed)* is selected for *Store tile layer data as* under the *General* tab.



Store tile layer data as: Base64 (gzip compressed)

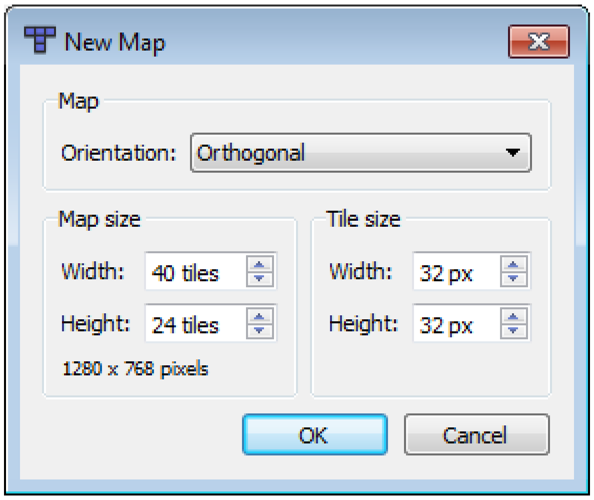
☐ Include DTD reference in saved maps

☑ Reload tileset images when they change

Language: System default

☑ Hardware accelerated drawing (OpenGL)

Next, we'll create a new a new project by navigating *File > New…* from the main toolbar. We'll create a resource to match the screen dimensions of a BlackBerry Dev Alpha:


New Map

Map

Orientation: Orthogonal

Map size

Width: 40 tiles

Height: 24 tiles

1280 x 768 pixels

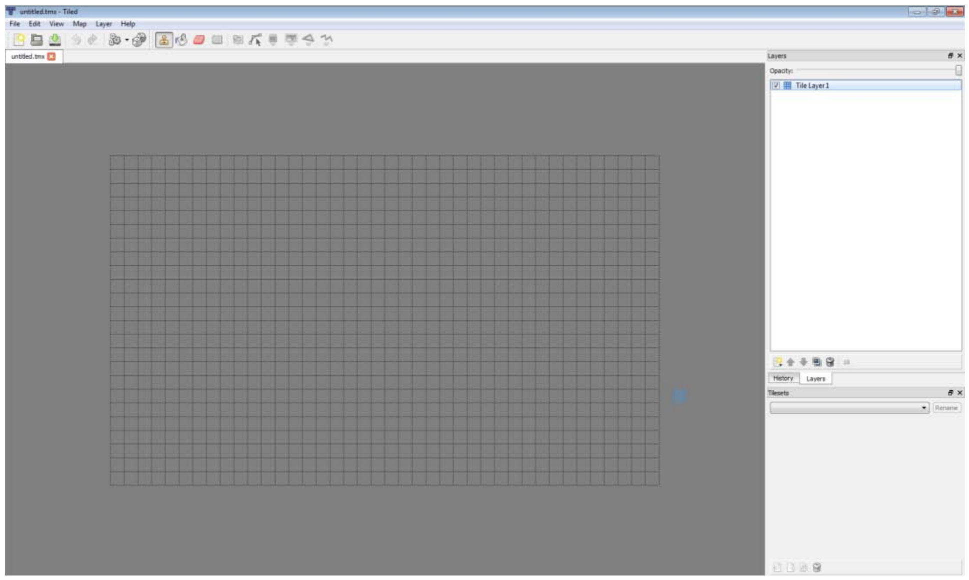
Tile size

Width: 32 px

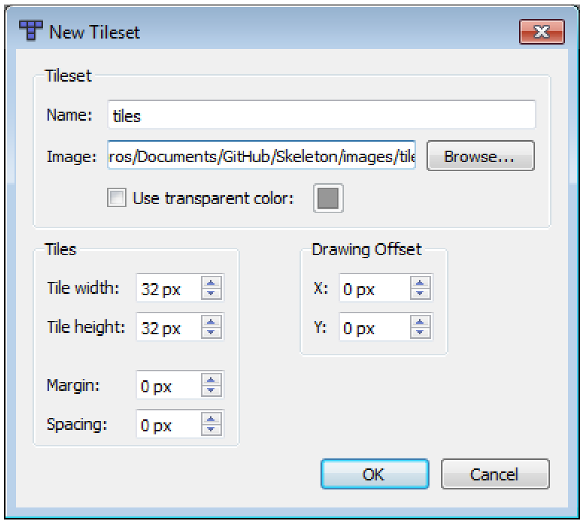
Height: 32 px

- *Orthogonal* indicates we will have a square-grid system in place, as opposed to hexagonal.
- *Map size* indicates the number of tile rows and columns in our map.
- *Tile size* indicates the dimensions of each tile; this is set to match up to our *tileset.png*.

Based on the *Map size* and *Tile size*, the overall dimensions of our tile map will be *1280 x 768 pixels*; the dimensions of our BlackBerry Dev Alpha. Click *OK* and we should see our tile map ready for editing.



The next step is to import *tiles.png* so that we can draw to our tile map. From the top toolbar, navigate *Map > New Tileset…* and import the *tiles.png* that is saved under the *images* sub-folder as follows:



New Tileset

Tileset

Name: tiles

Image: ros/Documents/GitHub/Skeleton/images/tile

Browse...

Use transparent color:

Tiles

Tile width: 32 px

Tile height: 32 px

Margin: 0 px

Spacing: 0 px

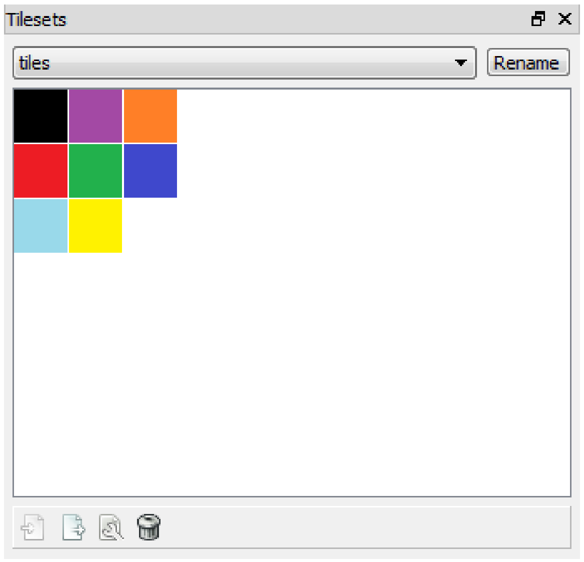
Drawing Offset

X: 0 px

Y: 0 px

OK Cancel

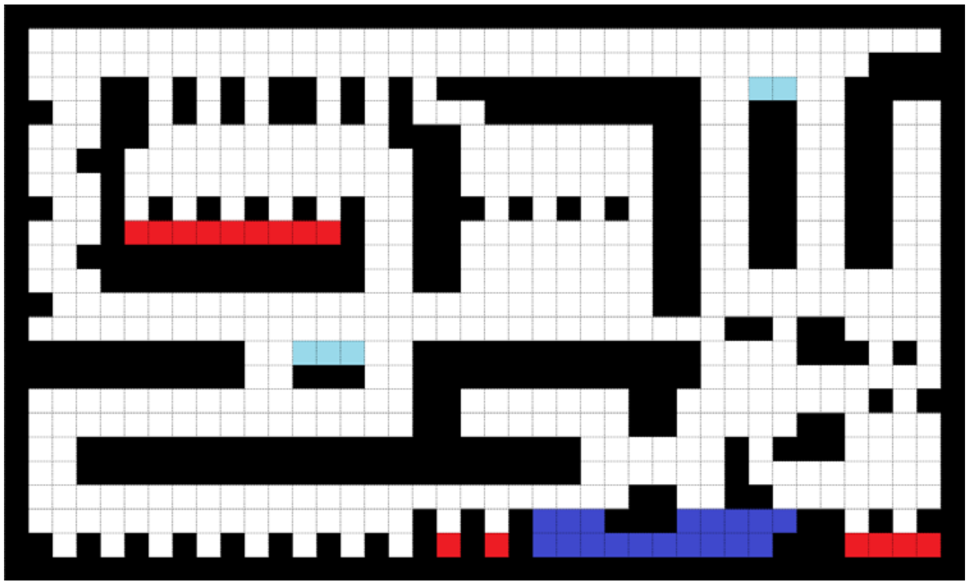
Once complete, click the *OK* button and you should find your new tile set added to the *Tilesets* panel in the bottom-right corner of the window.



You can now select various tiles from the *Tilesets* panel and then paint those to your tile map. In our case, we will use the following colours:

- *Black* to represent walls / solid objects.
- *Red* to represent lava.
- *Blue* to represent water.
- *Light blue* to represent ice.
- *White* to represent empty space.

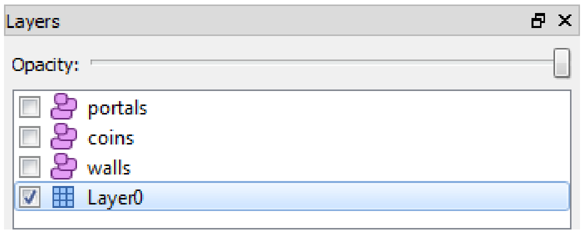
Use these tiles to generate a custom level. You can be as creative as you like at this stage. The file we will be using is *0-0.tmx* and available on Github; it should be saved to the *tmx* sub-folder of the skeleton project.



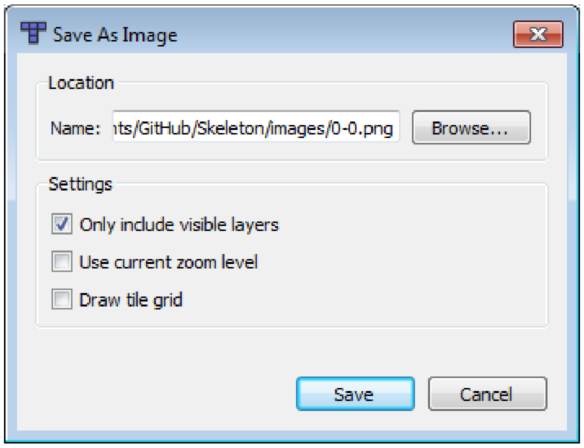
If you download *0-0.tmx* from Github, you will notice that we've:

- Renamed *Tile Layer 1* to *Layer0*; and
- Added a few additional *Object* layers (portals, coins, and walls.)

For the time being, we will only focus on the *portals* and *coins* layers as these will dictate where those objects appear in our actual world. Before we add these layers manually, we want to export our tile map as an image, so that we can use it as a background for our application. Ensure that only *Layer0* is selected in the *Layers* panel.



Following this, from the top toolbar navigate *File > Save As Image…* and save *0-0.png* to the *images* sub-folder of your project.


Save As Image

Location

Name: ıts/GitHub/Skeleton/images/0-0.png

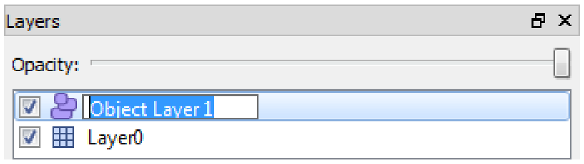
Settings

[✓] Only include visible layers

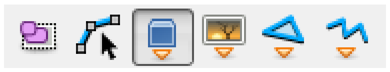
[ ] Use current zoom level

[ ] Draw tile grid

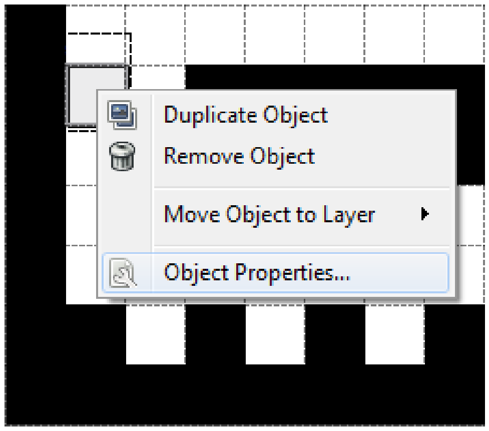
Once saved, we can focus on adding the *portals* and *coins* layers. From the top toolbar, navigate *Layer > Add Object Layer*. This will create a new layer in your *Layers* panel.



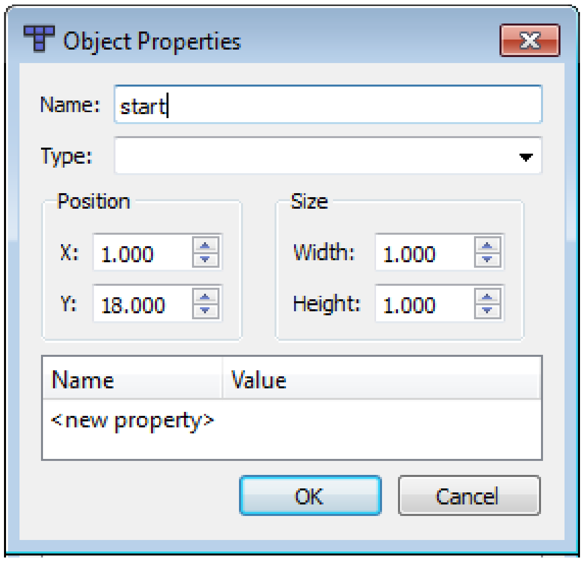
Rename *Object Layer 1* to *portals*; we will use this layer to define the start and end points for our hero. For this layer, we will need to use the *Insert Object* tool while the *portals* layer is specifically selected in the *Layers* panel.



Insert an object at coordinates (1, 18) – near the bottom-left of the map – and right-click on the newly added object and select *Object Properties…* from the pop-up menu.



Set the properties for this object as follows.

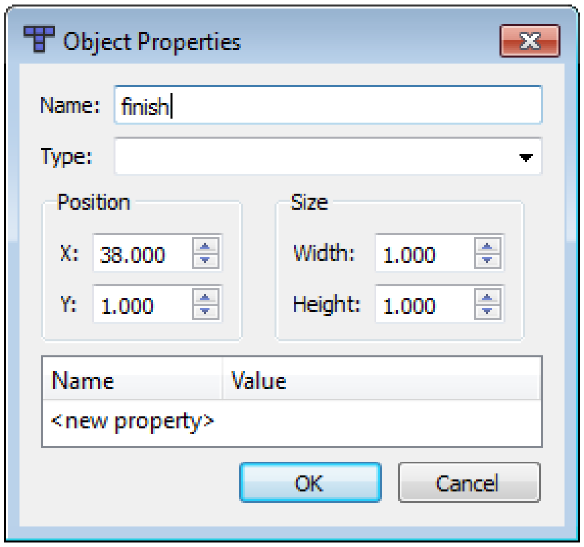


Name: start
Type:
Position
X: 1.000
Y: 18.000
Size
Width: 1.000
Height: 1.000

Name	Value
<new property>	

This object that we just placed will represent the starting point for our hero, it is important that the starting point is placed *first*.

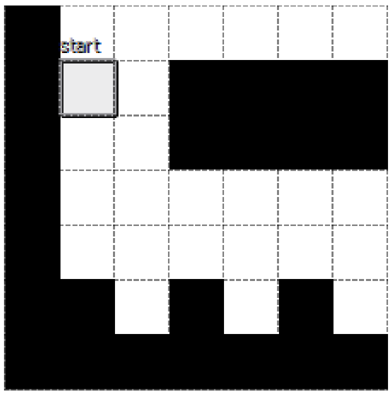
Next, we will place the *finish*. Very similar to the *start*, the finish will be placed at coordinates (38, 1) – near the top-right of the map. Place the new object and set its properties as follows.

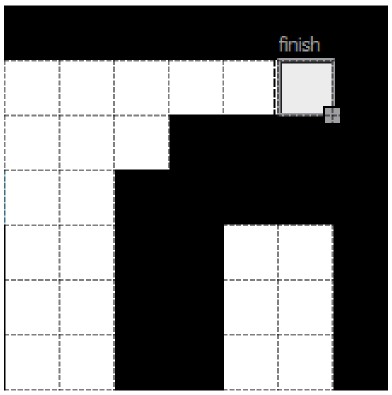


Name: finish
Type:
Position
X: 38.000
Y: 1.000
Size
Width: 1.000
Height: 1.000

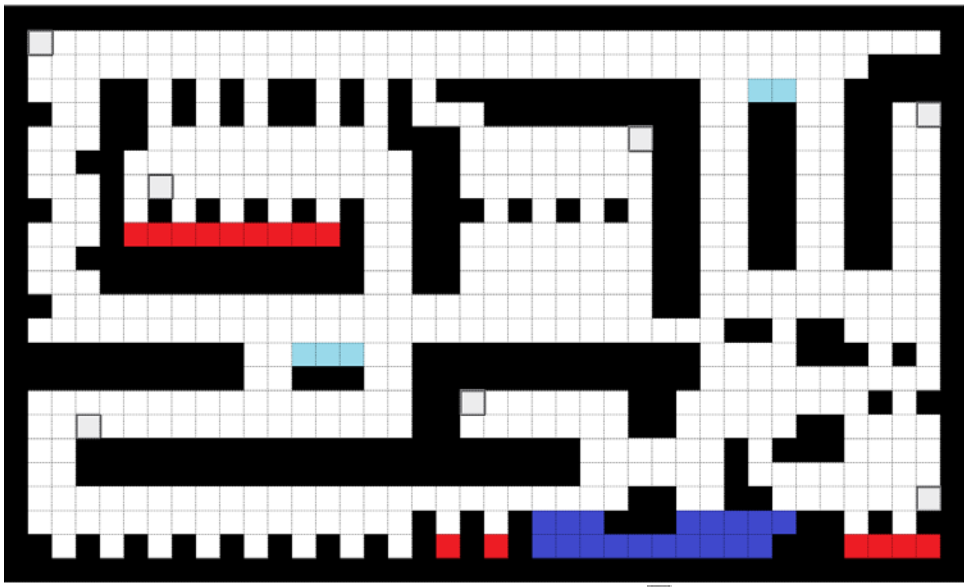
Name	Value
<new property>	

We should now have two portals, a *start* and a *finish*, place in opposite corners of the map.





Next, we will repeat the process that we performed for the *portals*, but now we will create another object layer called *coins*. In our case, we've added seven *coin* objects around the tile map as follows.



Coin objects are represented by the ▢ *icons.*

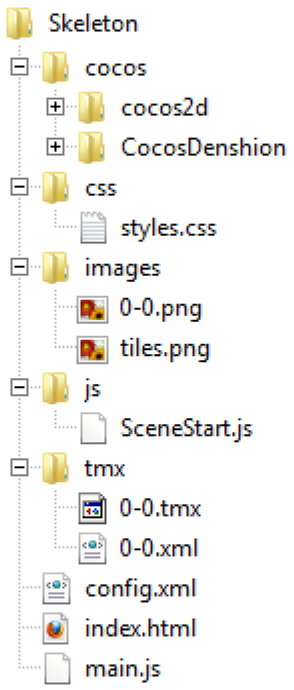
When adding the *coins*, it is important that our *coins* layer is selected in the *Layers* panel, otherwise our objects will be associated with a different layer. With the coins, we do not need to assign any names; we will only be referencing the position of each object.

There is one important layer that we haven't addressed yet, and that is the *walls* layer. We will take a closer look at this when we're actually adding physics to our application, however the same approach will apply. Basically, we will be creating objects in a third layer that overlap with all of the black tiles in our tile map, then using that data to create fixed objects in our physics world; more on that later.

Now that we have a tile map with *portals* and *coins*, as well as an image representation of the background, we can go ahead and save our TMX file by selecting *File > Save As…* from the main toolbar. We will want to save this file as *0-0.tmx* in the *tmx* sub-folder of our project.

We will also save an *XML* version of the very same file; all that entails is changing the file extension when we actually save the file. The purpose for this is that the Ripple emulator has issues with the MIME type of TMX file formats. Since a TMX file *is* really just an XML file, by modifying the file extension we allow Ripple to successfully process the file when testing our application.

Our file and folder structures will consequently now look as follows.



Now that we have all our assets (*tiles.png*, *0-0.png*, and *0-0.xml*) in place, we can go ahead and leverage them within our application. This will involve modifying:

- main.js
- SceneStart.js

Let's take a look at each of these individually.

main.js

We don't require a great deal of changes in *main.js*, however this is where we preload all or our assets. Previously, we were not preloading anything, but now we need to make sure that the Cocos2d-HTML5 framework is aware of the assets we want to use.

Instead of re-posting the entire source code of this file, we will only highlight new additions. ***Code in blue*** *refers to code that previously existed and should be used as a reference for where the new code goes.*

```
        cc.Loader.getInstance().preload([
            {type: 'tmx', src: './tmx/0-0.xml'},
            {type: 'image', src: './images/0-0.png'},
            {type: 'image', src: './images/tiles.png'}
        ]);
```

You can see that we are now actually providing assets in our preload array. The one special case is that we are providing the XML version of our tile map, instead of the TMX version, for Ripple compatibility.

SceneStart.js

With the preloading configured, we can now actually use those resources to start populating our scene's layer with content. The first thing we want to do is create placeholder variables for our `background`, `hero`, and `coins` sprites.

```
var LayerStart = cc.Layer.extend({
    background: null,
    hero:       null,
    coins:      null,

    ctor: function () {
        this._super();
        return true;
    }
});
```

Next, we'll focus on adding the `background` sprite. To do this, all we really need to do is create a new instance of the sprite class with a supplied image, set its properties, and then add it to our layer.

```
    ctor: function () {
        this._super();

        this.background = cc.Sprite.create('./images/0-0.png');
        this.background.setAnchorPoint(new cc.Point(0.0, 0.0));
        this.background.setPosition(new cc.Point(0.0, 0.0));
        this.addChild(this.background, 0);

        return true;
    }
```

It is imperative that the path used in the call to `create` is the same path that we preloaded in *main.js*. We set the anchor point for this sprite to be (0, 0) – the bottom-left corner of the sprite. And we then set the position of the anchor to be (0, 0) – the bottom-left corner of our window. Finally, we add the sprite to our layer.

Similar to the background sprite, we'll use the same approach for the hero sprite.

```
        this.addChild(this.background, 0);

        this.hero = cc.Sprite.create('./images/tiles.png', new cc.Rect(32.0, 32.0, 28.0, 28.0));
        this.hero.setAnchorPoint(new cc.Point(0.5, 0.5));
        this.hero.setPosition(new cc.Point(0.0, 0.0));
        this.hero.j = []; /* Will hold the impulse force acting on the hero. */
        this.addChild(this.hero, 2);

        return true;
    }
```

We've added the hero sprite after the code that adds our background sprite. The key differences are:

- We're referencing `tiles.png` instead of `0-0.png`;
- Since `tiles.png` is a sprite sheet, we' supply an x-coordinate, y-coordinate, and dimensions for the specific image we want to use from the sprite sheet; and
- Our anchor point is set to the center of the sprite, as opposed to the bottom left corner;

We also create an impulse variable (`this.hero.j`) to keep track of the force acting on our hero.

Next, we want to add the portals and coins, however their positions will be drawn from the *tmx* resource we created in the previous steps. So, first we will need create a *tmx* variable and a counter variable.

```
    ctor: function () {
        var tmx, n;

        this._super();
```

We then populate `tmx` by referencing the *XML* version of our exported resource.

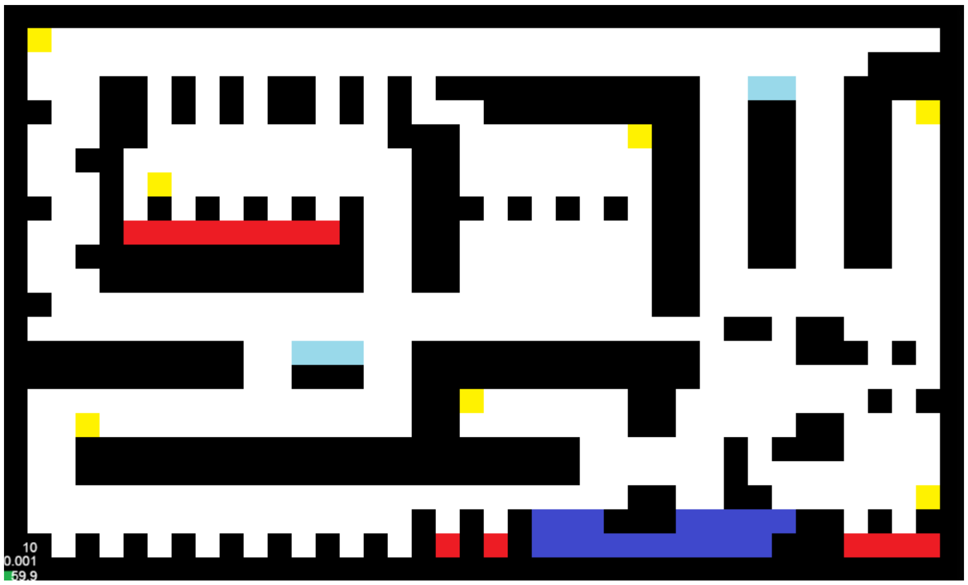
```
        this._super();

        tmx = cc.TMXTiledMap.create('./tmx/0-0.xml');

        this.background = cc.Sprite.create('./images/0-0.png');
```

Finally, we create sprites for the coins and the *finish* portal in the same way as before, though now we're cycling through any number of objects based on the *XML* resource when adding our coins.

```
        this.addChild(this.hero, 2);

        /* Load the coins. */
        this.coins = tmx.getObjectGroup('coins').getObjects();
        this.coins.sprites = [];
        for (n = 0; n < this.coins.length; n = n + 1) {
            this.coins.sprites.push(cc.Sprite.create('./images/tiles.png', new cc.Rect(32.0, 64.0, 32.0, 32.0)));
            this.coins.sprites[n].setPosition(
                new cc.Point(
                    this.coins[n].x + this.coins[n].width / 2.0,
                    this.coins[n].y + this.coins[n].height / 2.0
                )
            );
            this.addChild(this.coins.sprites[n], 3);
        }
        this.coins.sprites.count = this.coins.sprites.length;

        /* Load the finish portal. */
        this.finish = cc.Sprite.create('./images/tiles.png', new cc.Rect(32.0, 0.0, 32.0, 32.0));
        this.finish.setPosition(new cc.Point(
            tmx.getObjectGroup('portals').getObjects()[1].x + tmx.getObjectGroup('portals').getObjects()[1].width / 2.0,
            tmx.getObjectGroup('portals').getObjects()[1].y + tmx.getObjectGroup('portals').getObjects()[1].height / 2.0
        ));
        this.finish.setOpacity(0.0);
        this.addChild(this.finish, 1);

        return true;
    }
```

We haven't actually used the *start* portal, meaning our hero sprite will default to position (0, 0). This is fine for now, as we will use the *start* portal when we're actually adding physics to our world.

If we run the application at this point, we should be rendering the following.



Note the background, yellow coins, and in the bottom-left corner the hero is slightly visible. Our *finish* portal isn't visible since we set its opacity to *0* however it is hidden away at the end of the top-right corridor.

At this point, you should have an idea of how we can leverage *TMX* resources within our Cocos2d-HTML5 applications, as well as create basic sprites from standalone files or sprite sheets and add them to our scene. The next step will be to add physics to our world.

Physics with Box2DWeb and Web Workers

This is going to be a fairly large section as we are combining two important (and not overtly simple) ideas: Box2DWeb and Web Workers. Box2DWeb is a physics engine that handles a lot of complicated behavior on our behalf. Concepts like gravity, collisions, and rotation to name a few are implemented within the framework so that the developer doesn't need to figure this out themselves.

For a very good overview and complementary set of tutorials on Box2DWeb, this CreativeJS compilation of Seth Ladd's tutorials provides a wealth of knowledge:

In conjunction with Box2DWeb, we'll use Web Workers for the primary reason of separating the main application thread (UI / interaction) from the physics calculations. JavaScript, inherently, is not king of mathematical computation so we want to separate any intense work being done to avoid impacting our frame rates and application responsiveness as much as possible. This way, instead of performing calculations every frame, our main application thread simply grabs the most up-to-date calculations available and leverages those.

Overall, this combination yielded much smoother results within this sample Cocos2d-HTML5 application as opposed to relying on the standard *update* method that you can schedule every animation frame with the Cocos2d-HTML5 framework. The difference again being that we are now operating on two separate threads, as opposed to lining up rendering and physics on the same thread.

First, we'll focus on setting up a skeleton Web Worker and then we'll populate it with our physics engine. Finally, we'll leverage the physics engine data to update our scene.

Setting up the Skeleton Web Worker

To begin, we will need to download the Box2DWeb framework. This can be obtained at the following Google Code repository.

In our case, we will rename the JavaScript file to *Box2D.js* and we will save it to the *js* sub-folder of our project.

In the same *js* sub-folder, we will also create a new file called *Box2dWebWorker.js*. This is where our main implementation will go, however we'll discuss its actual contents in a moment.

SceneStart.js

We'll need to make some modifications to the existing *SceneStart.js* file. As before, ***code in blue*** is previously existing code and is included for reference. The first thing we'll need to do is create a global namespace.

```
var _g = {
    LayerStart: null
};

var LayerStart = cc.Layer.extend({
```

In this case, _g simply stands for *global*, and the naming of *LayerStart* is intentional for consistency. We're creating this variable because we will need a reference to our root layer that can be leverage inside the Web Worker's *message received* listener.

We'll also create a placeholder variable for our physics Web Worker.

```
var LayerStart = cc.Layer.extend({
    physics:    null,
    background: null,
```

Having created our global namespace, we'll want to initialize it as soon as possible which, in our case, is right after we call the layer's super function.

```
        this._super();
        _g.LayerStart = this;

        tmx = cc.TMXTiledMap.create('./tmx/0-0.xml');
```

Now that we have the preliminary initialization out of the way, we can actually create a Web Worker. This step is fairly straightforward.

```
        tmx = cc.TMXTiledMap.create('./tmx/0-0.xml');

        this.physics = new Worker('./js/Box2dWebWorker.js');

        this.background = cc.Sprite.create('./images/0-0.png');
```

Here we're assigning our placeholder variable to be a new Web Worker; the Web Worker itself will be initialized with *Box2dWebWorker.js*; this file is currently empty, but we'll rectify that soon. Once implemented we'll be able to *send* messages from ***SceneStart.js* to *Box2dWebWorker.js***.

Next, we'll initialize the Web Worker to be able to *receive* messages.

```
        this.physics = new Worker('./js/Box2dWebWorker.js');
        this.physics.addEventListener('message', function (e) {
        });

        this.background = cc.Sprite.create('./images/0-0.png');
```

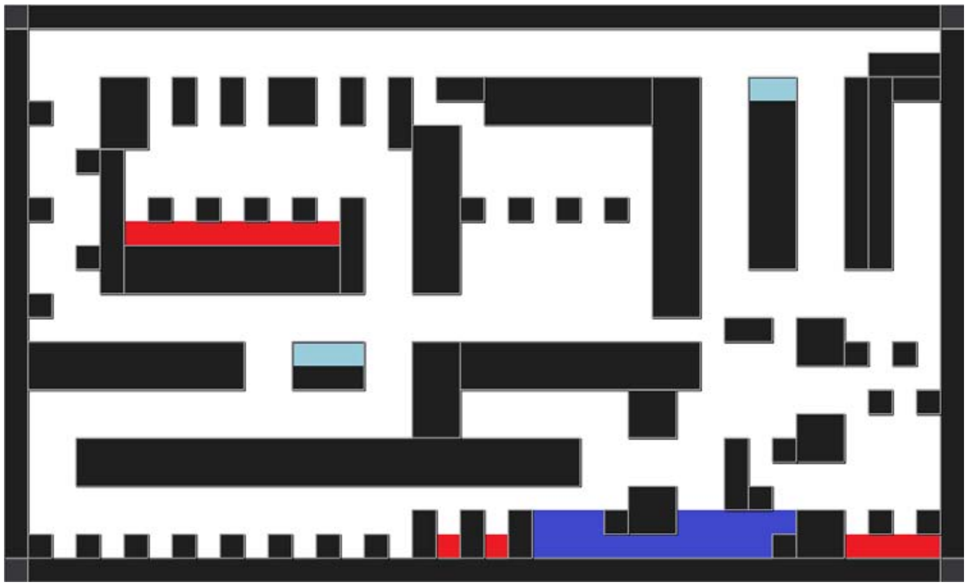
This will allow us to send messages from ***Box2dWebWorker.js* to *SceneStart.js*** and permits two-way communication between the main application thread and the Web Worker. It is important to note that we can transmit variables (strings, numbers, complex objects, etc.) but we *cannot* transmit functions.

Now that our Web Worker is ready, we'll send a message to *Box2dWebWorker.js* to indicate that we want it to initialize itself and we'll provide some values.

```
        this.physics.addEventListener('message', function (e) {
        });

        this.physics.postMessage({
            msg: 'init',
            walls: tmx.getObjectGroup('walls').getObjects(),
            coins: tmx.getObjectGroup('coins').getObjects(),
            portals: tmx.getObjectGroup('portals').getObjects()
        });

        this.background = cc.Sprite.create('./images/0-0.png');
```

All four variables (*msg*, *walls*, *coins*, and *portals*) are arbitrarily named beyond the fact that these names make sense for readability's sake. When sending and receiving messages, we'll use the *msg* variable as an indication of the action we want to take. The remaining variables are object sets from our *TMX* resource that we will use to construct our physics world.

Previously, we did not actually create the *walls* layer with *Tiled*. However, following the instructions in the previous section should give you a good idea of how to create these walls and in the simplest sense all we've done is create additional objects overlapping with the black wall colours.


*A screenshot of the TMX resource's **walls** object layer.*

Alternatively, you can download the *TMX* (and *XML* counterpart) of the complete tile map from the BoxQuest Github repository.

The final change we will perform in *SceneStart.js* for the time being is setting up a scheduled update function. We will use this function to send information about user-input to the Web Worker to introduce forces on our hero.

```
        this.addChild(this.finish, 1);

        this.schedule(this.update);
        return true;
    },

    update: function () {
    }
});

var SceneStart = cc.Scene.extend({
```
Pay particular attention to the **,** *that has been introduced to separate the ctor and update functions.*

With these changes, we've:

- Configured a skeleton Web Worker that will later be used to send and receive messages.
- Scheduled an update function that will leverage the Web Worker to create forces in the physics world based on user input.
- Triggered an initialize command within our actual Web Worker.

In order for any of the above to actually mean anything though, we'll have to actually implement the Web Worker in *Box2dWebWorker.js*.

Box2dWebWorker.js

First we'll start by importing an external script; specifically, we will import the Box2D.js script that houses all of the physics functionality we need.

```
importScripts('./Box2D.js');
```

Next, we'll add an event listener for incoming messages; this is the only *required* function that is common across essentially all Web Workers.

```
importScripts('./Box2D.js');

self.addEventListener('message', function (e) {
    if (e.data.msg === 'ApplyImpulse') {
        self.hero.j = e.data.j;
    } else if (e.data.msg === 'init') {
        self.init(e.data);
    }
});
```

As noted earlier, we will rely on a *msg* variable to determine which actions we will take. We're already familiar with the `'init'` *msg* from when we created the Web Worker initially. The `'ApplyImpulse'` message has not yet been implemented, but will be set when we are providing impulse / force data from the main application thread based on user input.

Let's take a look at the *init* function first.

```
        self.init(e.data);
    }
});

self.init = function (objects) {
    var fixtureDef, bodyDef, object, n;
```

A fairly simple beginning, `init` accepts objects that are passed in from the message. In this case, this will refer to our *walls*, *coins*, and *portals*. We also define four placeholder variables for fixtures, bodies, individual objects, and a counter; respectively. Next, we'll set some global, world values.

```
    var fixtureDef, bodyDef, object, n;

    /* Our world. */
    self.world = new Box2D.Dynamics.b2World(
        new Box2D.Common.Math.b2Vec2(0.0, 24.0),    /* Gravity. */
        true                                        /* Allow sleep. */
    );
    self.world.scale = 32.0;
    self.remove = [];
```

The first thing we do is create the world and provide a 2D vector that describes the direction of gravity. We also set a global *scale* value. This value is important because our physics world exists on a smaller scale (in our case 32 times smaller) than our screen dimensions. The reasoning behind this is that if we don't scale everything down in the physics world, we're going to be dealing with *massive* objects and the physics may not behave the way we want them to. For most applications, a value around 30 will be ideal but this may need to be adjusted.

We're also creating an empty array where we will store objects that we want removed from the scene. This will be used when the hero comes into contact with a coin and we want to remove that going. The reason we need this array is that we do not want to be removing objects in the middle of physics calculations. Instead, we will keep a queue of items to remove and do so inbetween calculations.

Next we'll set a number of global physics properties for fixtures.

```
    self.remove = [];

    /* Global properties. */
    fixtureDef                = new Box2D.Dynamics.b2FixtureDef();
    fixtureDef.density        = 1.0;
    fixtureDef.friction       = 0.0;
    fixtureDef.restitution    = 0.0;
    bodyDef                   = new Box2D.Dynamics.b2BodyDef();
```

It is important to note that by setting the `friction` to `0.0`, we are preventing the hero from being able to jump against a wall and *stick* to it. However, this also means that the hero will continually slide along the ground; we will need to address this issue ourselves (and we will.)

The restitution is a *bounce* value and has been set to `0.0` as well. When running into a wall (or falling) we don't want the hero bouncing (in this case), otherwise we could increase this value. We finish by creating a new body variable that will be reused when generating physics objects; we'll start with walls.

```
    bodyDef                   = new Box2D.Dynamics.b2BodyDef();

    /* Generate our walls. */
    for (n = 0; n < objects.walls.length; n = n + 1) {
        object                    = objects.walls[n];
        bodyDef.type              = Box2D.Dynamics.b2Body.b2_staticBody;
        bodyDef.position.x        = (object.x + object.width / 2.0) / self.world.scale;
        bodyDef.position.y        = -(object.y + object.height / 2.0) / self.world.scale;
        fixtureDef.shape          = new Box2D.Collision.Shapes.b2PolygonShape();
        fixtureDef.shape.SetAsBox(object.width / 2.0 / self.world.scale, object.height / 2.0 / self.world.scale);
        self.world.CreateBody(bodyDef).CreateFixture(fixtureDef).SetUserData({});
    }
```

Generating walls and generating coins will be very similar. In both cases, we cycle through the *TMX* resource's objects and, based on the dimensions/position of each object, we create a corresponding object in the physics world. By leveraging `b2_staticBody`, we indicate that these objects will *not* be affected by forces such as gravity and will remain stationary.

In addition, both walls and coins are assigned a rectangular shape.

```
        self.world.CreateBody(bodyDef).CreateFixture(fixtureDef).SetUserData({});
    }

    /* Add our coins. */
    for (n = 0; n < objects.coins.length; n = n + 1) {
        object                    = objects.coins[n];
        bodyDef.type              = Box2D.Dynamics.b2Body.b2_staticBody;
        bodyDef.position.x        = (object.x + object.width / 2.0) / self.world.scale;
        bodyDef.position.y        = -(object.y + object.height / 2.0) / self.world.scale;
        fixtureDef.shape          = new Box2D.Collision.Shapes.b2PolygonShape();
        fixtureDef.shape.SetAsBox(object.width / 2.0 / self.world.scale, object.height / 2.0 / self.world.scale);
        object = self.world.CreateBody(bodyDef).CreateFixture(fixtureDef).SetUserData({
            tagName: 'coin',
            index: n
        });
    }
```

The primary difference when generating coins (aside from the source data) is that when we actually create the body, we call `SetUserData` and assign a `tagName` and `index`. This will help us keep track of *when* and *which* coins are collided with by the hero.

Speaking of the hero, let's add a hero to the physics world.

```
            index: n
        });
    }

    /* Add a box hero. */
    bodyDef.type            = Box2D.Dynamics.b2Body.b2_dynamicBody;
    bodyDef.position.x      = (objects.portals[0].x + objects.portals[0].width / 2.0) / self.world.scale;
    bodyDef.position.y      = -(objects.portals[0].y + objects.portals[0].height / 2.0) / self.world.scale;
    fixtureDef.shape        = new Box2D.Collision.Shapes.b2PolygonShape();
    fixtureDef.shape.SetAsBox(28.0 / 2.0 / self.world.scale, 28.0 / 2.0 / self.world.scale);
    self.hero = self.world.CreateBody(bodyDef);
    self.hero.CreateFixture(fixtureDef).SetUserData({});
    self.hero.j = [];
    self.hero.contacts = 0;
```

This time instead of defining a `b2_staticBody` we define a `b2_dynamicBody`. This means that the body we create *will* be affected by the world's physics.

In the Sprites and Tile Maps section, we only ever created a sprite for the *finish* portal. You can see here though, that we're using the data of the *start* portal to determine where our hero will be added to the world (no longer will the hero live in the bottom-left corner.)

We want the hero to be accessible from a number of locations within this Web Worker, so we create a reference to `self.hero` (`self` essentially means `this`.) We also create `self.hero.j` which will hold the X-Y impulse force acting on the hero.

Finally, as the hero will be starting in mid-air, we initialize the number of objects currently in contact with the hero to 0; this will help us keep track of whether the hero can currently jump. In this case, the hero will only be able to jump when in contact with ground/wall. Let's take a look at how we can listen for these collisions.

```
    self.hero.contacts = 0;

    /* Collision listener for coins, portals, etc. */
    self.listener = new Box2D.Dynamics.b2ContactListener();
    self.listener.BeginContact = function (contact) {
        /* Only our hero moves so it must be a hero collision. */
        self.hero.contacts++;

        /* If there is a collision, find if one of the objects is a coin and, if so, remove that coin. */
        if (contact.m_fixtureB.GetUserData().tagName === 'coin') {
            self.remove.push(contact.m_fixtureB.GetBody());
            self.postMessage({
                msg: 'remove',
                index: contact.m_fixtureB.GetUserData().index
            });
        } else if (contact.m_fixtureA.GetUserData().tagName === 'coin') {
            self.remove.push(contact.m_fixtureA.GetBody());
            self.postMessage({
                msg: 'remove',
                index: contact.m_fixtureA.GetUserData().index
            });
        }
    };
```

First, we create a ***new*** `Box2D.Dynamics.b2ContactListener` and then define the `BeginContact` function; this gets triggered on any new collision. If one of the objects in the collision is a coin, we will queue that coin up for removal.

Note that we also call `postMessage` to notify our main application thread of this removal. We do this because not only do we need to remove the objects from the physics world, but we also have to remove the sprites that we are rendering within the Cocos2d-HTML5 framework.

```
                index: contact.m_fixtureA.GetUserData().index
            });
        }
    };

    self.listener.EndContact = function () {
        /* Keep track of how many collisions are currently in effect for jumping purposes. */
        self.hero.contacts--;
    };
```

We also implement `EndContact` and decrement the collisions counter any time we are no longer touching an item. By incrementing on `BeginContact` and decrementing on `EndContact` we know when the user isn't touching anything (i.e. mid-air) and will use that to permit/deny the ability to jump.

```
        self.hero.contacts--;
    };

    self.world.SetContactListener(self.listener);
```

While defining the contact listeners is great, without actually setting the contact listener for the world, we would never see our functions trigger.

```
    self.world.SetContactListener(self.listener);

    setInterval(self.update, 0.0167);   /* Update the physics 60 times per second. */
    setInterval(self.cleanup, 0.0111);  /* Check for object removal 90 times per second. */
};
```

And finally, we make two separate calls to `setInterval`; the first will be responsible for updating the physics 60 times per second, and the second is responsible removing any queued objects from the world. The latter is performed more often to ensure it is addressed as early as possible once required. Let's take a look at how cleanup occurs.

```
    setInterval(self.cleanup, 0.0111);  /* Check for object removal 90 times per second. */
};

self.cleanup = function () {
    var n;

    /* Cycle through and remove all outstanding bodies. */
    for (n = 0; n < self.remove.length; n = n + 1) {
        self.world.DestroyBody(self.remove[n]);
        self.remove[n] = null;
    }
    self.remove = [];
};
```

As you can see, it's pretty straightforward in that all we're doing is cycling through any queued items, invoking the `DestroyBody` function on those objects, and then cleaning up.

The final function to take a look at is `self.update`. This is where the cycle-by-cycle physics calculations are performed, and fresh data is sent back to the main application thread. In the simplest form, this method would look as follows.

```
    self.remove = [];
};

self.update = function () {
    /* Process the physics for this tick. */
    self.world.Step(
        0.0167, /* Frame rate based on milliseconds. */
        20,     /* Velocity iterations. */
        20      /* Position iterations. */
    );

    /* Reset any forces. */
    self.world.ClearForces();
    self.hero.j = [0.0, 0.0];
};
```

Here, we're updating the forces in our physics world based on an assumed 60 frames per second and cleaning up after ourselves. However, we need to add a little more in order to make the hero behave in the manner that we want within our world.

```
self.update = function () {

    /* If the hero is not touching any floor, walls, or ceiling, eliminate any vertical (jumping) impulse. */
    if (self.hero.contacts === 0) {
        self.hero.j[1] = 0.0;
    }

    /* Apply the current horizontal and vertical impulse forces to our hero. */
    self.hero.ApplyImpulse(
        new Box2D.Common.Math.b2Vec2(self.hero.j[0], self.hero.j[1]),
        self.hero.GetWorldCenter()
    );

    /* If there is no horizontal impulse, set the horizontal velocity to 0; prevents any sliding on the frictionless ground. */
    if (self.hero.j[0] === 0) {
        self.hero.SetLinearVelocity(
            new Box2D.Common.Math.b2Vec2(
                0.0,
                self.hero.GetLinearVelocity().y
            )
        );
    }

    /* Cap the maximum horizontal velocities between -5.0 and 5.0. */
    self.hero.SetLinearVelocity(
        new Box2D.Common.Math.b2Vec2(
            Math.max(-5.0, Math.min(self.hero.GetLinearVelocity().x, 5.0)),
            self.hero.GetLinearVelocity().y
        )
    );

    /* Process the physics for this tick. */
    self.world.Step(
```

There are four key modifications we're making:

- If the hero is not in contact with any objects, prevent vertical impulse. This ensures the hero cannot jump again while in mid-air.
- Based on the adjusted impulse, we then apply those forces to the hero to get movement.
- If there is no horizontal impulse (supplied by touch inputs), set the horizontal velocity to 0. Since we removed friction from our static objects, this is how we make the hero stop moving.
- Finally, since impulse is constantly acting on the hero while the touch inputs are active, we want to prevent the hero from gaining too much velocity and we restrict horizontal velocity between -5.0 and 5.0 units.

Following that, we perform the calculations as before and clean up, and the last thing we then need to do is actually send this data *back* to our main application thread so that it can be used to update the position and rotations of our Cocos2d-HTML5 sprites.

```
    /* Reset any forces. */
    self.world.ClearForces();
    self.hero.j = [0.0, 0.0];

    self.postMessage({
        hero: {
            x: self.hero.GetPosition().x * self.world.scale,
            y: -self.hero.GetPosition().y * self.world.scale,
            r: self.hero.GetAngle()
        }
    });
};
```

In this case we're not communicating a *msg* since the presence of a *hero* object will be indication enough. In this message, we supply the new xy-coordinates and rotation of the hero.

Up to this point, we've accomplished:

- Initializing our Web Worker.

- Performing physics calculations and supplying that data back to the main application thread.

The final code we must implement will be back in *SceneStart.js* and includes:

- Augment the scheduled *update* function to supply impulse information *to* the WebWorker.
- Augment the Web Worker's message listener to leverage the calculation data that is supplied to update our Cocos2d-HTML5 sprites.

SceneStart.js Revisited

First, let's start by fully implementing our scheduled update function to supply impulse information to our Web Worker. Until we implement touch controls, this impulse will always be zero, but this implementation will allow us to test our application as-is.

```
    update: function () {
        this.physics.postMessage({
            msg: 'ApplyImpulse',
            j: this.hero.j
        });
        this.hero.j[1] = 0.0; /* Reset vertical impulse to 0 after each frame (otherwise hero will fly away.) */
    }
});

var SceneStart = cc.Scene.extend({
```

Recall the `'ApplyImpulse'` message listener that we implemented in *Box2dWebWorker.js*; this is where those messages originate from. Basically, every frame we send the hero's current impulse to the WebWorker for its calculations. This impulse is affected by user input, which we have not implemented yet, so the current result is that no external forces (aside from gravity) will be acting on our hero.

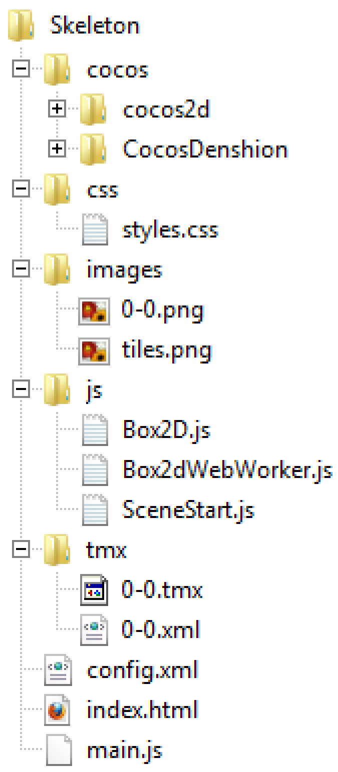
And the last thing we need to do is listen for incoming messages (sent by the Web Worker back to the main application thread) and apply the data that comes with it to our hero.

```
        this.physics.addEventListener('message', function (e) {
            if (e.data.hero) {
                /* If hero data exists, update our position and rotation. */
                _g.LayerStart.hero.setPosition(new cc.Point(
                    e.data.hero.x,
                    e.data.hero.y
                ));
                _g.LayerStart.hero.setRotation(e.data.hero.r / (Math.PI * 2.0) * 360.0);
            } else if (e.data.msg === 'remove') {
                /* If we need to remove sprites (i.e. hero ran into a coin), update our counter. */
                _g.LayerStart.removeChild(_g.LayerStart.coins.sprites[e.data.index]);
                _g.LayerStart.coins.sprites[e.data.index] = null;
                _g.LayerStart.coins.sprites.count = _g.LayerStart.coins.sprites.count - 1;

                if (_g.LayerStart.coins.sprites.count === 0) {
                    _g.LayerStart.finish.runAction(cc.FadeTo.create(2.0, 255.0));
                }
            }
        });

        this.physics.postMessage({
```

First we check for a `hero` variable, if it exists we know that we have new position/rotation data and we call the `setPosition` and `setRotation` functions on our Cocos2d-HTML5 sprite.

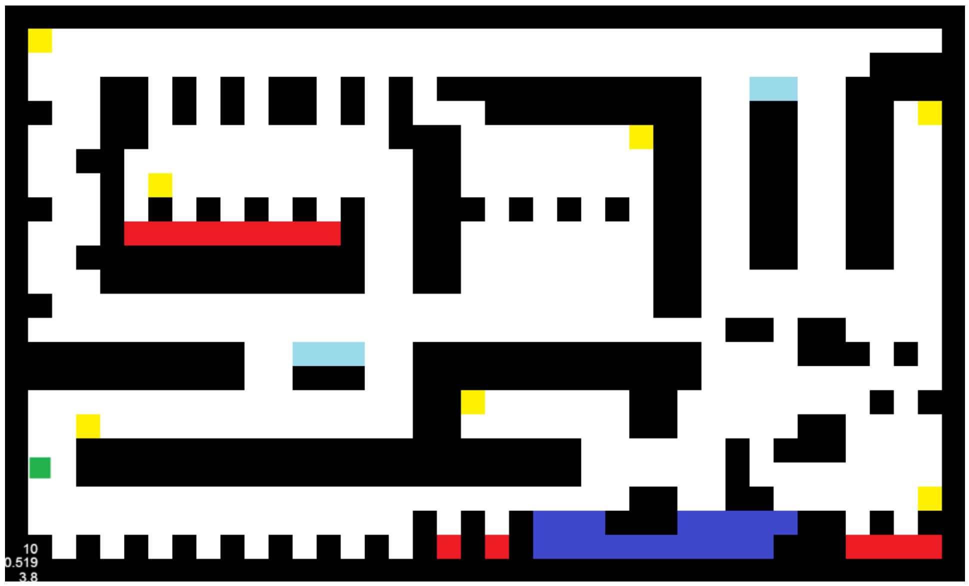
The only other message we could potentially get is a `'remove'` message. If so, we remove said sprite from the scene and cleanup any references to that object. We also decrement our coin sprite counter. If our counter reaches 0, we know that we've collided with / removed all the coins in our scene. If this is the case, we run a Cocos2d-HTML5 action to fade our *finish* portal into view.

The final file structure hasn't changed much, we've only added two (2) new files, however for reference this is what it now looks like.



```
Skeleton
  cocos
    cocos2d
    CocosDenshion
  css
    styles.css
  images
    0-0.png
    tiles.png
  js
    Box2D.js
    Box2dWebWorker.js
    SceneStart.js
  tmx
    0-0.tmx
    0-0.xml
  config.xml
  index.html
  main.js
```

And though we can't interact with the hero just yet, if you test this application with Ripple in your browser, you should see the hero start a few blocks above the platform in the bottom-left corner, and then fall down to be stopped by the ground.



The hero hovers momentarily before falling, at which point the ground stops the movement.

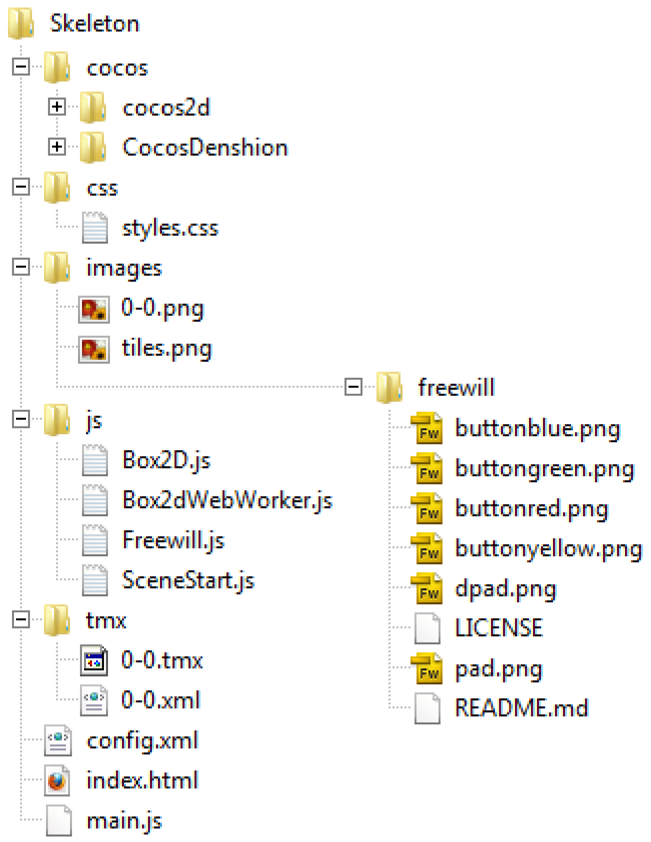
Implementing Touch Controls

If you've followed these tutorials all the way through, kudos to you fine developer. In this section we'll look at adding touch controls to our Cocos2d-HTML5 application. While Cocos2d-HTML5 does have its own touch implementation available to developers, I'm actually opting for a framework I developed called Freewill.js.

The reasoning for this is:

- The Cocos2d-HTML5 touch implementation (at the time of writing this sample) has some issues in being able to distinguish between touch points; direct HTML5 touch implementation does not. As this application requires multi-touch, we require a multi-touch capable framework.
- Freewill.js was written to accommodate a variety of scenarios so after coming this far, let's make the last steps as easy as possible.

To start off, let's download *Freewill.js* from the BoxQuest Github repository and save it to our *js* folder. We'll also need some assets that go along with *Freewill.js*, these can be downloaded from the same repository under the *images* sub-folder; copy the *freewill* folder to the *images* sub-folder of your project.



```
Skeleton
  cocos
    cocos2d
    CocosDenshion
  css
    styles.css
  images
    0-0.png
    tiles.png
    freewill
      buttonblue.png
      buttongreen.png
      buttonred.png
      buttonyellow.png
      dpad.png
      LICENSE
      pad.png
      README.md
  js
    Box2D.js
    Box2dWebWorker.js
    Freewill.js
    SceneStart.js
  tmx
    0-0.tmx
    0-0.xml
  config.xml
  index.html
  main.js
```

freewill *is a sub-folder of* ***images****.*

In order to leverage Freewill.js within our application, we'll need to make a few minor adjustments to our previous code. The files that will be affected are:

- index.html
- styles.css

- SceneStart.js

As in the previous sections, ***code in blue*** simply represents code that already exists as a means to more easily identify where the new code should be added.

index.html

Cocos2d-HTML5 allows us to specify the JavaScript files we will be using and the framework will load them for us at runtime. We already did this for SceneStart.js, but now that we're adding Freewill.js, we want to ensure the framework is aware.

```
                    appFiles: [
                        './js/Freewill.js',
                        './js/SceneStart.js'
                    ]
```

We also need to add a `<div>` overlay that will be receiving the touch input. For this, we'll create the basic element here then style it as needed through CSS.

```
    <body>
        <div id="freewill"></div>
        <canvas id="ccCanvas"></canvas>
```

styles.css

In order for the *freewill* overlay to receive the input, we'll want to ensure it is covering the entire screen while being in the foreground.

```
body {
    background-color: black;
    cursor: default;
    font-family: sans-serif;
    height: 100%;
    margin: 0;
    padding: 0;
    width: 100%;
}

#freewill {
    bottom: 0px;
    overflow: hidden;
    position: fixed;
    top: 0px;
    width: 100%;
    z-index: 999;
}
```

SceneStart.js

We're now ready to start using Freewill.js within our Cocos2d-HTML5 application and begin by instantiating a freewill object.

```
        this.addChild(this.finish, 1);

        /* Load freewill. */
        this.freewill = new Freewill({
            container: document.querySelector('#freewill')
        });
```

Here we reference the `<div>` that our touch listener events will be assigned to. We can now start adding Joystick and Button controls to our application. First, we'll set up a Joystick control to handle our hero's movement.

```
        /* Load freewill. */
        this.freewill = new Freewill({
            container: document.querySelector('#freewill')
        });

        /* Add a Joystick to control movement. */
        this.freewill.move = this.freewill.addJoystick({
            imageBase: './images/freewill/buttonblue.png',
            imagePad: './images/freewill/buttonblue.png',
            fixed: true,
            pos: [0.0, 0.0],
            trigger: [0.0, 0.0, window.innerWidth / 2.0, window.innerHeight],
            opacLow: 0.0,
            opacHigh: 0.0
        });
```

Generally, when we add a Joystick, we can supply a base image and a pad image; the pad follows the touch point while the base represents where the joystick is located. In our case, this Joystick will be completely hidden since we do not want any images interfering with the viewable screen. As such, we pick `buttonblue.png` for both `imageBase` and `imagePad` images as it has no impact on the overall look.

We set `fixed` to ***`true`*** because, again, since our Joystick is not visible, we don't actually need to be updating its position. On that note, we set the position of the Joystick to coordinates `[0.0, 0.0]` for the very same reasoning.

The `trigger` is what is important for this control scheme. The `trigger` dictates the area in which touch events will be registered. We're overriding the default behavior to specify that *anywhere on the left-half of the screen* the `move` Joystick should register touch events.

Finally, we set both the low and high opacities to 0 to ensure the Joystick does not appear, and we save a reference to our Joystick as ***`this`***`.freewill.move`. This reference is important as it will now allow us to implement our own custom *onTouchStart*, *onTouchMove*, or *onTouchEnd* functionality. In our case, we only need to implement *onTouchMove* and *onTouchEnd*.

```
        /* Add a Joystick to control movement. */
        this.freewill.move = this.freewill.addJoystick({
            imageBase: './images/freewill/buttonblue.png',
            imagePad: './images/freewill/buttonblue.png',
            fixed: true,
            pos: [0.0, 0.0],
            trigger: [0.0, 0.0, window.innerWidth / 2.0, window.innerHeight],
            opacLow: 0.0,
            opacHigh: 0.0
        });

        /* If there is movement, update our horizontal impulse force to that of the Joystick's horizontal position. */
        this.freewill.move.onTouchMove = function () {
            _g.LayerStart.hero.j[0] = this.velocity[0];
        };
```

Each Freewill.js Joystick has an XY-velocity based on where the user's finger is in relation to the base of the Joystick. In our case, if the Joystick `trigger` detects movement, we will retrieve the velocity of the Joystick and set the hero's horizontal impulse to reflect that movement.

```
        /* If there is movement, update our horizontal impulse force to that of the Joystick's horizontal position. */
        this.freewill.move.onTouchMove = function () {
            _g.LayerStart.hero.j[0] = this.velocity[0];
        };

        /* When we release the Joystick, ensure the impulse force is 0. */
        this.freewill.move.onTouchEnd = function () {
            _g.LayerStart.hero.j[0] = 0.0;
        };
```

In order to prevent the hero from running after the Joystick is released (i.e. finger is raised off the screen), we reset the hero's impulse to 0 when the Joystick `trigger` detects this release. Because our scheduled update supplies the hero's impulse data to our Web Worker every frame, this is all we need in order to implement horizontal movement.

However, we also want our hero to have the ability to jump. For this, we don't need Joystick behavior but instead we need Button behavior. Freewill.js provides this as well.

```
        /* When we release the Joystick, ensure the impulse force is 0. */
        this.freewill.move.onTouchEnd = function () {
            _g.LayerStart.hero.j[0] = 0.0;
        };

        /* Add a Button to control jumping. */
        this.freewill.jump = this.freewill.addButton({
            image: './images/freewill/buttonblue.png',
            fixed: true,
            pos: [0.0, 0.0],
            trigger: [window.innerWidth / 2.0, 0.0, window.innerWidth / 2.0, window.innerHeight],
            opacLow: 0.0,
            opacHigh: 0.0
        });
```

Similar to the Joystick, we are creating a completely transparent Button so the `image`, `fixed`, and `pos` values are more or less arbitrary. This time, we set the `trigger` to the right-half of the screen and again both opacities set to 0 (i.e. hidden.)

```
        /* Add a Button to control jumping. */
        this.freewill.jump = this.freewill.addButton({
            image: './images/freewill/buttonblue.png',
            fixed: true,
            pos: [0.0, 0.0],
            trigger: [window.innerWidth / 2.0, 0.0, window.innerWidth / 2.0, window.innerHeight],
            opacLow: 0.0,
            opacHigh: 0.0
        });

        /* When the user touches the screen, we initiate a jump (vertical impulse.) */
        this.freewill.jump.onTouchStart = function () {
            _g.LayerStart.hero.j[1] = -9.0;
        };

        this.schedule(this.update);
```

Finally, we take our Button object and implement `onTouchStart` behavior; in this case, we add a vertical impulse anytime `onTouchStart` occurs and the Web Worker will determine whether it is valid (i.e. are we touching ground/wall) and make the appropriate physics calculations.

You are now ready to test this with the Ripple Emulator in your browser! The one downside of this being a multi-touch game is that most PCs have only one mouse input, meaning you can only move *or* jump which is never ideal in a platformer.

The base BoxQuest project does have basic keyboard controls implemented and can be tested / used as a reference if you want to fully test within a browser. The two (2) required changes are contained within SceneStart.js (look for the commented code.)

Questions

By all means feel free to reach out to Erik Oros if you have any questions about this guide, the implementation, or anything else BlackBerry development related. Maybe you just have something to show off! You can usually find me through one of these channels:

- BlackBerry Development forums: **oros**
- Twitter: **@WaterlooErik**
- IRC (Channel: #BlackBerryDev): **ErikOros**