Jan 26, 2019

Tutorial

This is a grounds up step by step guide to the making of our project with MelonJS.

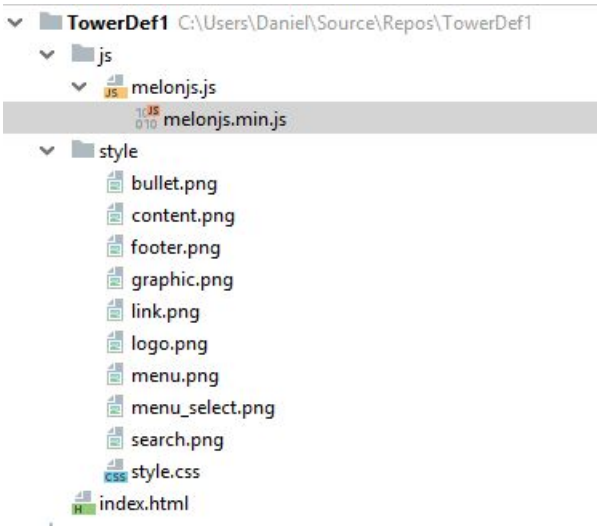
Goal: Render a Background Screen

Reference: Review the tiled_example_loader. Of course we could just load that example but what the use of that.

What you Will Need: Text Editor, Browser, the documentation example files from MelonJS (6.3.0 'melonJS-gh-pages').
What I am using: JetBrains WebStorm, Chrome v71.*, and the specific files in tiled_example_loader folder.

Create A Project Folder: TowerDef1
Create: index.html
Copy the style folder so that you have all the css and the images used in the css
Create Folder: js
Copy the melonJS.js to the js Folder.

Checkpoint 1: Project Structure



Note: I copied the melonjs.min.js and WebStorm recognized the association and created a sub item to melonjs.js

Now the index.html file content. You will see that I modified the index.html provided in the melonjs example but I stripped out the stuff so it can be added step by step. So copy the following HTML and we will rebuild it step by step so that we can explain what each step is doing when using the MelonJS. Honestly, you'll probably not copy it but go right to the end. But I think writing and reading this step by step will help build better mental models so that later you can jump in and extend your application with the framework more easily.

```
<!DOCTYPE html>
<html>
<head>
   <title>Tower Defense Program for CS 497 Spring 2019 Oregon State
University</title>
   <meta name="description" content="A tower defense program for the Maia Group" />
   <meta name="keywords" content="melonJS, HTML5, javascript, game, engine,
framework" />
   <meta http-equiv="content-type" content="text/html; charset=windows-1252" />
   <link rel="stylesheet" type="text/css" href="style/style.css" title="style" />
</head>
<body>
<div id="main">
   <div id="header">
       <div id="logo">
           <div id="logo_text">
               <!-- class="logo_colour", allows you to change the colour of the text
-->
               <h1>Tower Defense 1</h1>

               <h2 style="display:inline">A lightweight HTML5 game using the</h2>
                   <h1 style="display:inline">
                       <a href="http://www.melonjs.org"
style="display:inline">melon<span class="logo_colour">JS</span></a>
                   </h1>
                   <h2 style="display:inline">game engine</h2>

           </div>
       </div>
   </div>
   <div id="site_content">
       <div id="content">
           <h3>CS 497 Capstone Project</h3>
           <h4>Week 4 Content - Map Levels</h4>
           <span>Goal is to Display Different Levels using the following</span>
           <OL>
               <LI>A Tiled Map Resource</LI>
               <LI>MelonJS - Resource Loading Feature</LI>
               <LI>MelonJS - Rendering Feature</LI>
           </OL>
           <div id="jsapp">
<!-- We are going to put the application content here -->


           </div>
           <div id="info" style="width: 800px;">
               <div id="map" style="float: right;">
```

```
                </div>
            </div>
        </div>
    </div>
    <div id="content_footer"></div>
    <div id="footer">
        Copyright &copy; melonJS 2011-2015
    </div>
</div>
</body>
</html>
```

Then we are going to put our scripts in the jsapp div tag as indicated above.

The first script of course is the MelonJS script. Now remember as the HTML is loaded there is a certain order of execution and rendering. The major things to keep in mind are

1) DOM Elements don't render until the scripts preceding them have loaded.
2) Scripts are executed in order
3) When a script tag executes the DOM elements above it are available.

So the first tag to put in in the melonjs.js which we put in the 'js' directory. You could use the minified script but if you use the regular script then you can see or jump to the declaration of the created objects.

Insert the following in the "jsapp" div tag

```
<script type="text/javascript" src="./js/melonjs.js"></script>
```

Since we are using the 6.3.0 version of melonjs.js I will point out certain line numbers that illustrate what I think are key points to understanding the framework. The first is the creation of the 'me' object. The me object is created on line 26 of the melonjs.js (go ahead open it and look at it). Line 15 creates a function that is called with the parameter global. But if you look on line 42, right after the function is defined it is executed with the 'this' parameter. Which means that 'this' is just the 'window' object of the index.html. If the melonjs.js is executed in the Node.js environment then obviously the meaning of 'this' is different and that is handled in line 34. In the end line 41 me is added to the 'this' (because global = this) and therefore for the remaining of the scripts 'me' is now available to be accessed and used… Viola the framework is exposed to the rest of your application.

Now we need to have some resources for which we can display. And we get a sense from this example of how to begin to modularize your program into logical separate chunks.

Create file ressources.js

Create directory ‘data’
Create directory ‘data/maps’
Use “Tiled” to **create** map called “training_64x64.tmx”
Copy “training_64x64.tmx” to the ‘data/maps’ directory
Copy images associated with the training_64x64.tmx file to the ‘data/maps’ directory

MelonJs apparently is very very ‘name’ sensitive. So if the Tile Map Name, the associated Tile Set Images and the references to the g_resources (in the resources.js file cald in the me.loader.preload) are not named exactly right then your game crashes and you basically get Nothing.

We are going to talk about the nuances of creating the tile map. Because even though the Tiled Program is very powerful, its implementation into the MelonJS is very ‘technique’ sensitive.

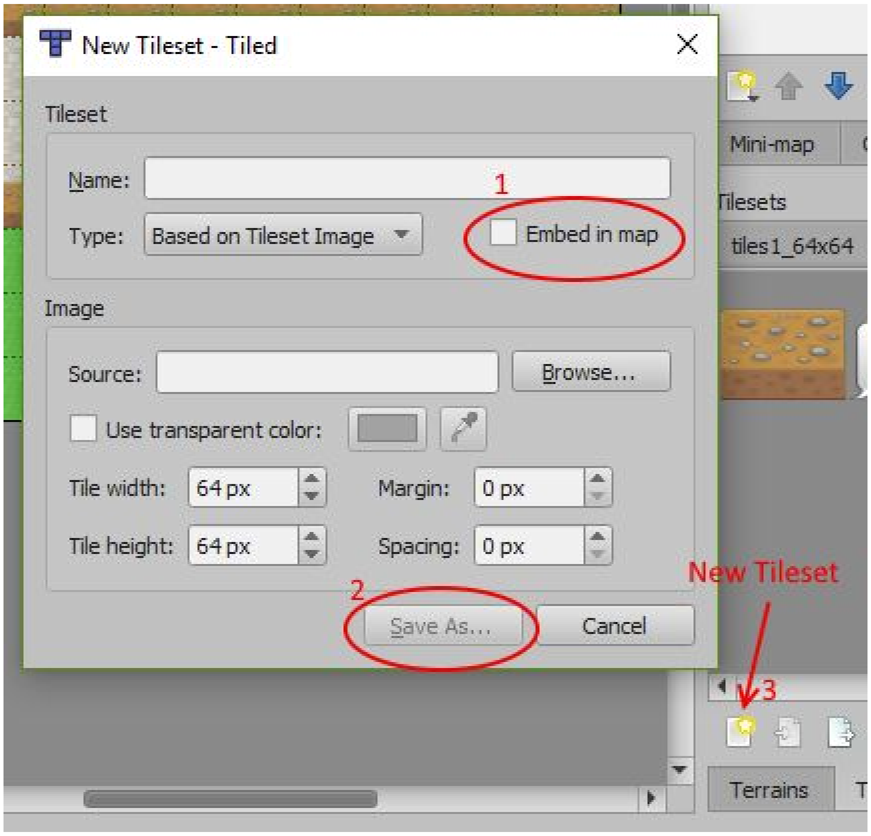
I’m going to make the assumption that you have just fiddled a bit with the Tiled program. You can save files and perhaps you found some images to use as a Tile Set. Patreon has created a pretty good Tiled Map Editor Tutorial Series that takes about 1.5 hours to complete. I recommend that you stop and complete that series right now. But if you have even just basic experience I think you can get through this tutorial.
(https://www.gamefromscratch.com/post/2015/10/14/Tiled-Map-Editor-Tutorial-Series.aspx)

I’m going to pause here and tell you that the Tiled program does take a bit of time to learn. I think I have the basics down in a few hours. What I found confusing was the relationships between the names of the tile sets, the image source names and the corresponding resources that are going to be exposed to the MelonJS ‘me’ object. So to minimize the confusion at the beginning I recommend that you just keep the images associated with the .tmx files, the tile set files (.tsx) and the .tmx files all in the maps directory. If you move any one of those items, the relative paths will not be correct and then MelonJS can’t find the images you need and you’ll be very confused as to what is exactly going wrong. When in reality it’s very simple… the file is moved.

Spend a little time working with Tiled and practice those breathing exercises that you need when you are feeling frustrated with things getting along too slowly.

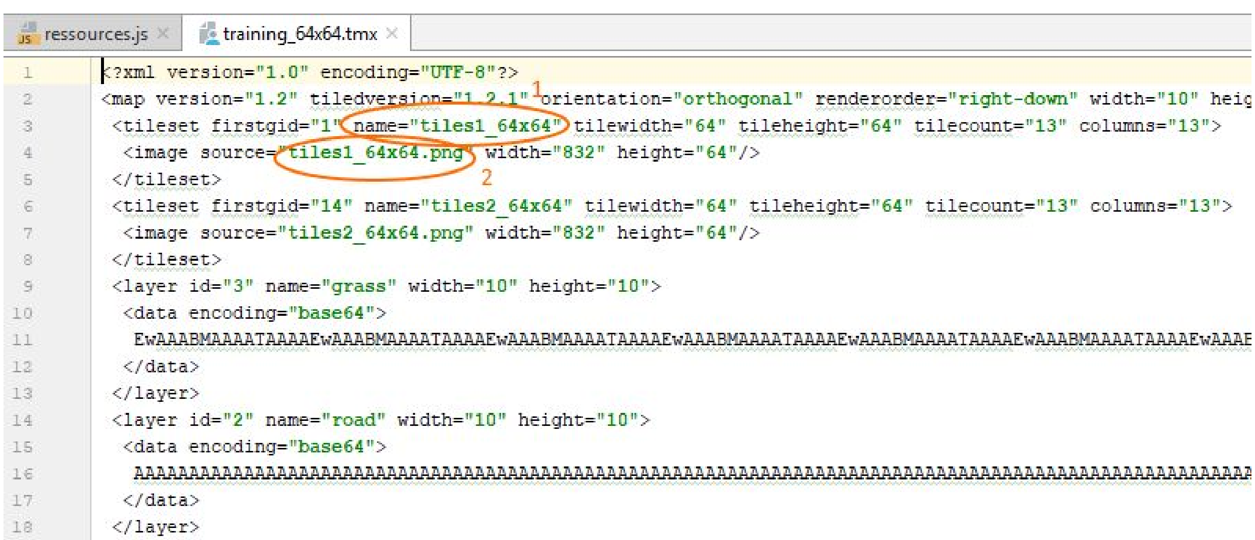
The Tiled file will be basing it’s tilesets on certain images of your choosing. If you open the .tmx file you will see that the .tmx just references the image and plucks the tiles from it and creates the associated tile set used. So it is important to make sure that the images used in the .tmx map are either in the same directory (or references the appropriate image)

How the .tmx file treats the tile sets depends on one little option when you create the tiled set.

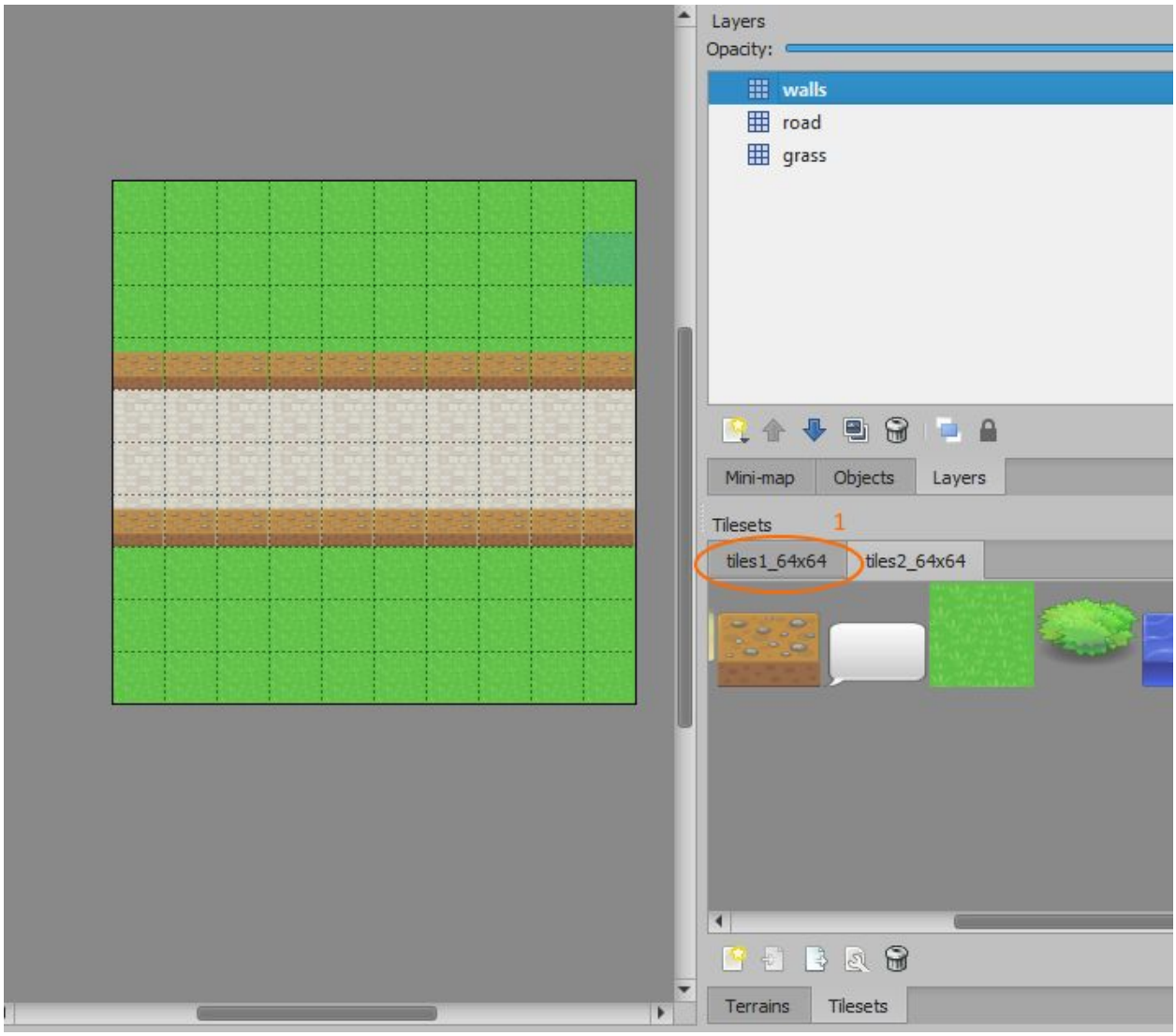


If Embed in map (item 1 above) is unchecked then after you click the browse and find the image, Button 2 changes to 'Save As' and the reference in the xml file becomes "`<tileset firstgid="1" source="tiles2.tsx"/>`"

Instead of what is listed below. While it probably can work. To KISS I took the Cute.tmx and changed the tiles to 64x64 and made a 10x10 tile map for easy math. (640x640px)



Above is the text from the .tmx file. Here the name of the tileset (1) correlates to the tab name in the below image (1). So you can see the associated file (2) is the image you need to make sure the tile map can find.



In the ressources.js File we are now going to create a g_ressources object.

```
/**
* ressources.js
*/

var g_ressources = [
   // images
   { name: "tiles1_64x64", type:"image",src:"data/maps/tiles1_64x64.png" },
   { name: "tiles2_64x64", type:"image",src:"data/maps/tiles1_64x64.png" },
   // TMX maps
   { name: "training_64x64",      type: "tmx",    src: "data/maps/training_64x64.tmx" }
```

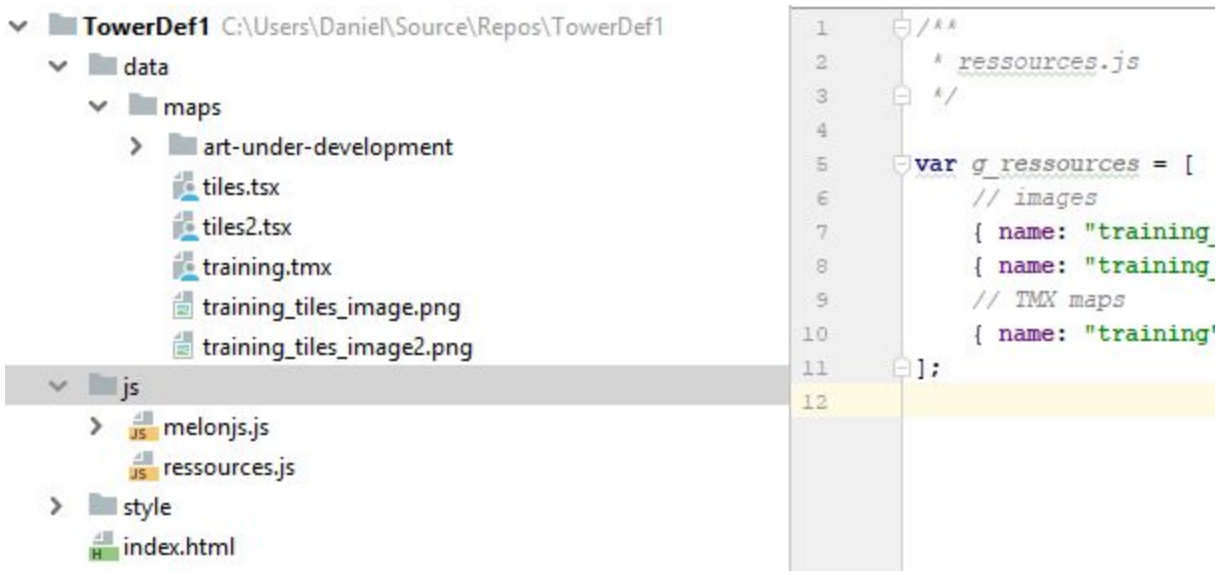
```
];
```

I used the tiles1_64x64.png in the same folder as the training_64x64.tmx map.
*** Important: Again, (learn from my stupid bug/mistakes) melon relies on the name property to look up the image. So make sure the 'name' matches the filename (without full path).

Add the ./js/resources.js file to the index.html

```
<script type="text/javascript" src="./js/ressources.js"></script>
```

Checkpoint 2: Project Structure



You have a simple index.html and ./js/ressources.js. Ignore "art-under-development". I just wanted to keep my scratch files close.

And the index.html's "jsapp" division should look like:

```
<div id="jsapp">
    <script type="text/javascript" src="./js/melonjs.js"></script>
    <script type="text/javascript" src="./js/ressources.js"></script>
</div>
```

In the next section we are going to create the main.js. For consistency we are going to put it in the ./js folder. It really doesn't matter. We just have to make sure we point to the right script in the index.html.

Create the file ./js/main.js

In the main.js we are going to do 2 things:

1. Define the game object
2. Execute the game.onload() function you just defined.

At this point, I'm going to just copy the code for the onload and loaded functions. Which is placed in the box below.

```
/*
* main.js
* A Basic Tiled Loader
*/

var game = {

   /**
    * initlization
    */
   onload: function () {

       // init the video
       if (!me.video.init(640, 640, {wrapper: "jsapp", scale:
me.device.PixelRatio})) {
           alert("Your browser does not support HTML5 canvas.");
           return;
       }

       // set all ressources to be loaded
       me.loader.onload = this.loaded.bind(this);

       // set all ressources to be loaded
       me.loader.preload(g_ressources);

       // load everything & display a loading screen
       me.state.change(me.state.LOADING);
   },

   /**
    * callback when everything is loaded
    */
   loaded: function () {
       // set the "Play/Ingame" Screen Object
       me.state.set(me.state.PLAY, new game.PlayScreen());

       // Set the Level
       me.levelDirector.loadLevel("training_64x64");

       // start the game
       me.state.change(me.state.PLAY);
   }
}; // game

//call the game.onload() function
me.device.onReady(function() {
```

```
    game.onload();
});
```

But look at the loaded function. There is a call to create a new game.PlayScreen()... But we don't have that function defined in our game object. PlayScreen is going to be put inside the ./js/play.js file.

Create the file ./js/play.js
Add a game.PlayScreen and make it inherit from me.Stage

Here is the Code to put in the play.js file

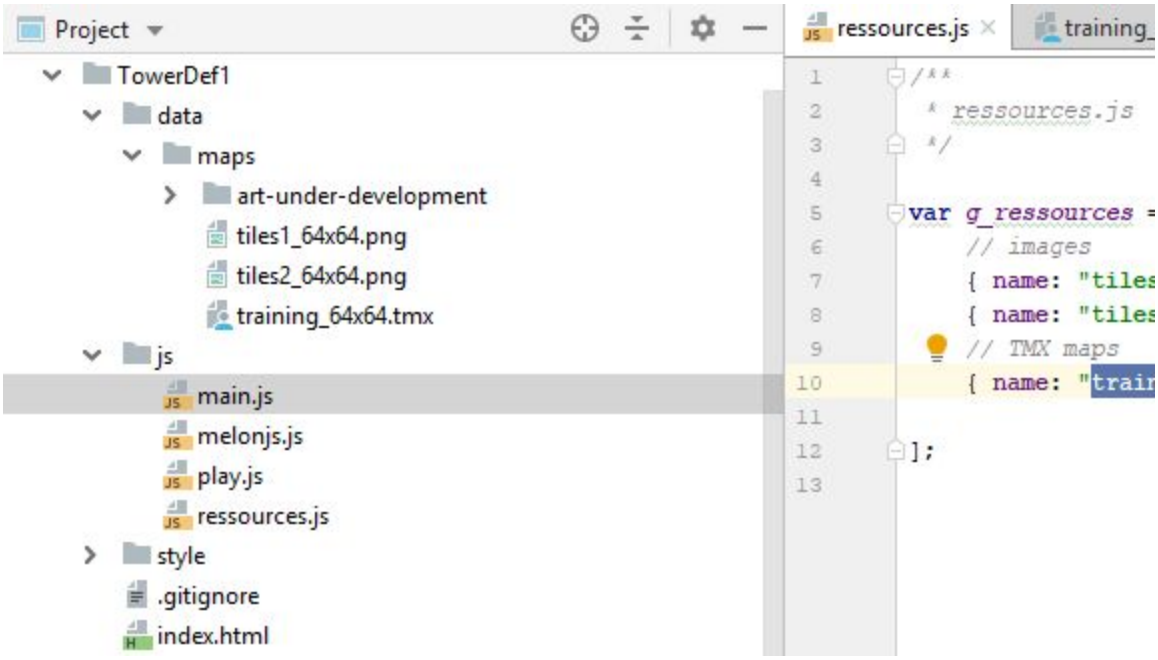
```
/*
* play.js
*
*/

game.PlayScreen = me.Stage.extend({
    /**
     *  action to perform on state change
     */
    onResetEvent: function() {
        // load a level
        me.levelDirector.loadLevel("training_64x64");
    },

    /**
     *  action to perform when leaving this screen (state change)
     */
    onDestroyEvent: function() {
        me.event.unsubscribe(this.handle);
    }
});
```

And of course, these new JS files and they need to be added to the index.html with play.js after the reference to the main.js. (Can you figure out why? Answer below Checkpoint 3)

Checkpoint 3: Project Structure



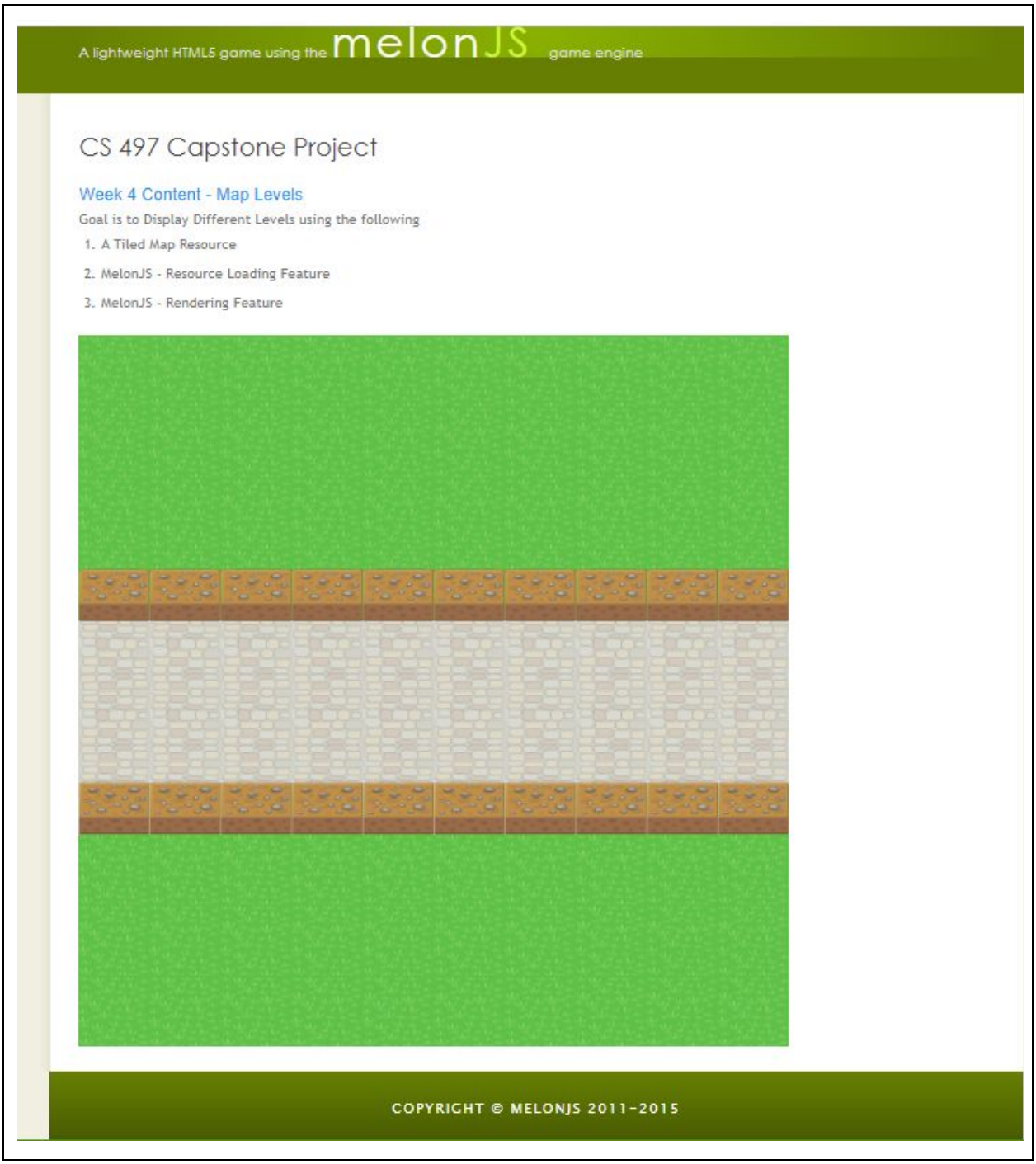
You have a simple index.html and ./js/ressources.js. Ignore “art-under-development”. I just wanted to keep my scratch files close.

And the index.html’s “jsapp” division should look like:

```
<div id="jsapp">
   <script type="text/javascript" src="./js/melonjs.js"></script>
   <script type="text/javascript" src="./js/ressources.js"></script>
   <script type="text/javascript" src="./js/main.js"></script>
   <script type="text/javascript" src="./js/play.js"></script>
</div>
```

(answer: If play.js scripted before then the script would try and add the PlayScreen Object to game. But game would not be defined because game is defined in main.js. Hence an error.)

A this point and after some debugging. I get a screen that shows one level of the tilemap.



I'm going to end this tutorial by adding the Key bindings. First let's shrink the viewport down so that using the keys to scroll the image around.

In the main.js change the 640 to 320.

```
if (!me.video.init(320, 320,....
```

Maind

There are 3 steps to the key binding… Step 1 adding the binding to the me object. We can do this after the DOM has loaded everything and the game object is loaded. So in the main.js we will add the key bindings to the game.loaded object.

```
loaded: function () {
   // set the "Play/Ingame" Screen Object
   me.state.set(me.state.PLAY, new game.PlayScreen());

   // Set the Level
   me.levelDirector.loadLevel("training_64x64");

   // enable the keyboard (to navigate in the map)
   me.input.bindKey(me.input.KEY.LEFT,  "left");
   me.input.bindKey(me.input.KEY.RIGHT, "right");
   me.input.bindKey(me.input.KEY.UP,    "up");
   me.input.bindKey(me.input.KEY.DOWN,  "down");
   me.input.bindKey(me.input.KEY.ENTER, "enter");

   // start the game
   me.state.change(me.state.PLAY);
}
```

This just now says that the game object is going to receive these inputs AND we can use the defined terms “left”, “right”... etc to reference the key.

Step 2 - Have PlayScreen Subscribe to the event
In the play.js file on the game.PlayScreen.onResetEvent object we are going to add what they call a ‘subscription’ to the event. So whenever a new level is loaded the ResetEvent will be called and the keys will be sent to the new PlayScreen Object.

```
onResetEvent: function() {
   // load a level
   me.levelDirector.loadLevel("training_64x64");
   // subscribe to key down event
   this.handle = me.event.subscribe(me.event.KEYDOWN, this.keyPressed.bind(this));
}
```

Step 3: You can see we just assigned a call back (‘keyPressed’) to this event. But we need to now define this function. Again we are in the game.PlayScreen object so the defnition of this.keyPressed goes there. Here is the whole game.PlayScreen object in the play.js. You can see we just use the tilewidth property to move the viewport.

```
/*
* play.js
*
*/
```

```
game.PlayScreen = me.Stage.extend({
    /**
     *  action to perform on state change
     */
    onResetEvent: function() {
        // load a level
        me.levelDirector.loadLevel("training_64x64");
        // subscribe to key down event
        this.handle = me.event.subscribe(me.event.KEYDOWN,
this.keyPressed.bind(this));
    },
    /**
     * update function
     */
    keyPressed: function (action /*, keyCode, edge */) {

        // navigate the map :)
        if (action === "left") {
            me.game.viewport.move(-(me.levelDirector.getCurrentLevel().tilewidth /
2), 0);

        } else if (action === "right") {
            me.game.viewport.move(me.levelDirector.getCurrentLevel().tilewidth / 2,
0);
        }

        if (action === "up") {
            me.game.viewport.move(0, -(me.levelDirector.getCurrentLevel().tileheight
/ 2));

        } else if (action === "down") {
            me.game.viewport.move(0, me.levelDirector.getCurrentLevel().tileheight /
2);
        }

        if (action === "enter") {
            me.game.viewport.shake(16, 500);
        }

        // force redraw
        me.game.repaint();
    },
    /**
     *  action to perform when leaving this screen (state change)
     */
    onDestroyEvent: function() {
        me.event.unsubscribe(this.handle);
    }
});
```

Final: