ZIPAPP GAME PLATFORM

MSc Computer Science Summer Project 2013

Author:
Dolan Miu

Supervisor:
Dean Mohamedally

AUGUST 21, 2013
UNIVERSITY COLLEGE LONDON
This report is submitted as part of requirement for the MSc Computer Science degree at UCL.

Contents

Abstract: Currently, Windows 8 and Windows Phone game development is lacking a template architecture that can allow novices to quickly build their first games. Using JavaScript and HTML5, a 2-dimensional game engine is made to allow such novices or non-developers to create games for Windows 8 and Windows Phone. With the template, tools and instructions provided, a game engine can be linked with Microsoft's ZipApp service as three templates, each as a different game genre for the user to create.

The game engine based on MelonJS requires minimal coding to get started, and uses WYSIWYG editor 'Tiled' for easy creation of game levels and menu screens. Tiled is an open source, free to use tile editor for use for making games. It has been extended to not only create tile maps, but also spawn objects, add lights, add enemies, alter camera properties, add finish lines and much more.

Three templates for leaning games development with Visual Studio using the ZipApp metaphor were created to demonstrate this capability. Full documentation and tutorials were made to assist the user with these templates through the game creation process.

Thorough testing with unit tests and various end users which meet the criteria were conducted to ensure a well-polished product.

1 Introduction

This report is a critical evaluation of the project undertaken during the summer of 2013 as part of the MSc Computer Science program at University College London.

The main aim of this project was to deliver and design a 2D game creation platform of high quality which will meet the needs of a programming novice or a student, but also give sufficient flexibility for experienced users. It will allow people to create games with ease by using pre-programmed assets which users can customise to create personalised games. This platform will be for Microsoft's ZipApp app creation service, a service which gives people an advantage at app creation. Currently, user difficulty is an issue within this service which will be rectified.

The following chapter's describes the approach and outlines the key parts for the project, outline the key parts of the project and demonstrate my approach to it. It was a great opportunity to develop a game framework for a large audience of developers.

1.1 The concept

(The concept behind the game platform is simple) Due to the rising popularity of mobile devices, 3.16% of the world population in July 2013 currently use mobile devices (w3schools 2013), and hence the rising popularity in the usage of mobile apps. The concept of the game platform will therefore address this opportunity. The numbers of casual gamers are rising, a potential market which many developers are hoping to exploit but lack the experience or knowledge to begin or create good quality games. This game platform is proposing to eliminate these problems and allow users to create games with ease. The framework is user-friendly, has a WYSIWYG editor, abstracting all the complex details from the developer.

ZipApp has proposed an idea which will extend its present service of providing utility app templates to games for the Windows 8 and Windows Phone. The Windows Store was released on February 2012 and now contains over 100,000 Apps (Muchmore 2012), a promising business venture for developers.

Knowing JavaScript is key in app and web development in this technological climate, especially with the introduction of HTML5 canvas and the decline of Adobe Flash. With technological advances and growth of popularity of the internet, numerous frameworks are being created for JavaScript. The rising importance of this programming language needs to be emphasised for potential web developers as many successful websites are using JavaScript to portray responsive, unique, and interactive websites. It is supported by all modern browsers, supports functional programming, can be used for client and server, and supports objects and inheritance.

With the rise of tablet PCs and mobile computing, aspiring developers are increasingly targeting these platforms but do not know where to begin, or would like an easy way to quickly prototype a game concept. Simple yet intuitive templates for different game genres are implemented within a game engine which allows novices and beginners to explore ideas without difficulty.

The ZipApp service helps users to start creating Apps by requiring them to provide details regarding their app and select 3rd party plugins they would like. The end result is a downloadable template App with the setup completed, giving an efficient start to the project.

1.2 Objectives

1.2.1 Creation of the platform

The main goal for the project is to produce a platform for novices to create specific genres of games on. The game genres decided on are a platformer, fighting, and a horror genre. Three demos of the genres using my platform will be created. Minimal amount of coding is needed and pre-built templates for the corresponding genres will be made to be put on the ZipApp website for the end user to easily utilise.

1.2.2 Library Implementation

The main bulk of the programming will be creating features users can deploy into their game. Objects such as water, physics, particles, camera styles, dynamic lighting, and controls will be added to give the user an array of tools to create with.

1.2.3 Helper tool

To reduce programming due to using the platform for the user, a helper tool will be developed to aid with the majority of the programming and handle with loading project files, creation of levels and aid plugin design.

1.2.4 ZipApp website integration

On the ZipApp website, the end user can select an option to begin the creation of a game, and then select a genre for the game which will prompt a download for the platform and the template file for the specified options. Once downloaded, the user can begin producing after following easy instructions.

To reduce the difficulty further, tools which will generate initialisation files and configuration files for the game will be available. This will increase the speed of development and reduce the lines of code needed to be written by the user.

1.2.5 Plugin Interface

ZipApp offers a variety of third party features which can be added to the App. Currently there are YouTube, Facebook and Twitter plugins amongst others. A plugin interface would have to be developed in the game platform in order to enable them to be imported into the project. This will allow developers with more experience to extend the platform, providing features not yet on the current platform. This will empower the community; boost its user base, the platform's potential, and thus its popularity. Examples of plugins would be a multiplayer socket plugin for the easy creation of multiplayer online games by abstracting away the complexity. This is already present in Unity with its Asset Store. Users purchase plugins for their game which will enhance their game with minimal effort from user.

1.2.6 Tutorial and Documentation

In order for the platform to be used effectively, the end user must have examples, and a well-documented API for them to use. Without these, the end user will have little idea on how the platform works. They would not know what the various commands, objects and keywords do, leading to inefficiency and deter people from using the platform.

1.2.7 Three Templates

Three templates will be made, each with a different genre. A platformer, stealth horror espionage, and a fighting genre will be available for the user to play and in conjunction allow extension and development of their own version of the template. The templates I would produce will be fully editable and documented so the user can utilise its full potential.

2 Background

2.1 Software, libraries and frameworks

2.1.1 Photoshop

Photoshop is the industry standard for graphics editing created by Adobe Systems. It will be used to create game sprites, backgrounds and various images. Its competitors include GIMP, Paint.NET, PaintShop Pro, and PixelMator. Due to its popularity, Photoshop's file format '.PSD' is widely integrated into many of its competitors. The PSDs of the assets I create will be included in the project template to allow the end user to edit and modify as they please.

2.1.2 MelonJS

MelonJS is a 2-dimensional JavaScript game framework which has been in development since 2011(MelonJS 2013). MelonJS, being free and open source, but most importantly used freely for commercial purposes by anyone who wishes to use it, is the foundation of the project. It provides image, sound, and animation libraries for JavaScript for creating websites and for games. However in addition to these features, MelonJS includes native support for tile maps which is within the requirements. The cumbersome part of using MelonJS is there is a lot of superfluous code which needs to be written, which is cumbersome to the user. A potential solution would be to program an application which will create the code for the player automatically.

2.1.3 Tiled

Tiled is an open source, free tile-map creator which is released under the MIT licence. Most 2D game levels are displayed using a tile map, a 2 dimensional array containing tile objects. The data structure contains information about the terrain, as well as references to dynamic objects such as the player and enemies. This tile map is stored in an open source XML format (.TMX), so a reader/writer for the file format can be created by anyone. It is also very user-friendly and easy to use, actively developed with a large active community supporting the program. MelonJS support Tiled as its main tile map editor, thus increasing its support availability. Tiled being the main tile map editor of MelonJS and used by other game engines such as AndEngine is a testimony of the popularity of itself(Rogers 2012).

2.1.4 Visual Studio 2012

The IDE used to develop is Visual Studio 2012, primarily for its advanced JavaScript debugging features. It simplifies development for the users whilst also providing an organised architecture for the development of templates since the next alternative to deploy Windows 8 Apps would be to unnecessarily use PowerShell scripts along with another IDE. It does not require the users to import their projects into a workspace before continuing, meaning less work for the user, quick downloads, and mobility as project files can be easily transferred across USB drives. There is also a lot more community support for building Windows 8 Apps with Visual Studio 2012 than any other solution.

It is the official tool Microsoft produced to create Windows 8 and Windows Phone Apps. ZipApp is also a Microsoft service, encouraging the use of Visual Studio. It also includes a feature which allows deployment to the Windows Store (Microsoft's App Store) very easily.

This will enable people using the platform to easily create and distribute apps without the need of additional third party tools.

2.1.5 GitHub

To manage and backup the projects source code, GitHub will be used: a service which enables the upload of the projects files to a private repository online to be stored. It automatically backs up the project each time an upload (formally termed a 'Push') is made to GitHub. This repository can be accessed by anywhere with permission, for example development could occur on a primary computer, and when it is time to present the product, the latest revision of the project can be accessed from GitHub to the laptop to show the audience. This is chosen over other competitors such as BitBucket because GitHub provides the necessities for open-source development and ease of use, building around the concept of "social coding".

2.2 Competition and similar apps

Many game platforms and frameworks released allow developers to construct their projects, each one with their positives and negatives but posing strong competition. The project must therefore have a strong unique selling point to compete against them. Unity will be the strongest competitor because it offers WYSIWYG tools with aesthetic visuals for minimal experience and effort, but most of their premium features including deploying to mobile devices is disabled unless their expensive licence is bought.

2.2.1 Unity3D

Unity3D is a 3D game development ecosystem created for the professional developer or the artist. It gives the user the necessary tools and an intuitive interface to begin development easily. It has multiple valuable features including DirectX 11 rendering, a plethora of supported file types and deployment to all major platforms including Windows 8 and Windows Phone, and allows scripting in JavaScript or C#. Additionally, pre-programmed scripts coding for specific functions can be bought from the asset store for the less experienced users. It unfortunately does not support true 2D graphics and has limited bitmap manipulation.

2.2.2 Game Maker

This is a 2D game engine with a long history designed for professionals and novices in mind. It has a graphical user interface for easy level creation and object behaviour with additional support for scripts using their own GML scripting language. This engine can also deploy into many platforms including Windows 8 and Windows Phone.

2.2.3 Delta Engine

Delta Engine is an open source multiplatform game engine. The programming language used is C# or C++. It requires a licence for commercial purposes, but is free for PC development.

2.2.4 CraftyJS

CraftyJS is a JavaScript/HTML5 game framework designed to work on all web browsers. It supports inheritance and sound, animation and effects similar to CreateJS. It has a state manager also and has a very similar set of features with its competitors.

2.2.5 CreateJS

CreateJS is a suite of modular libraries and tools which complement each other to develop interactive content on open web technologies using HTML5. The CreateJS suite consists of EaselJS for graphics manipulation and drawing, SoundJS which enable sound playback with

various settings for panning, delay and other dynamics, and TweenJS which provides animation on the loaded bitmaps.

2.2.6 LimeJS

LimeJS is an open source free HTML5 JavaScript game framework for developing JavaScript games for touch screens or desktop browsers. Requiring python knowledge to compile the framework together and there are no other resources for this apart from the developers API, the user accessibility is limited. It is harder to develop 3rd party plugins for this framework as the compiled JavaScript is not human readable.

2.2.7 ImpactJS

ImpactJS is a paid for HTML5/JavaScript game engine for desktop browsers and mobile browsers. It boasts its own map editor similar to Tiled, its proprietary Ejecta service and source code upon purchase. Ejecta is a quick and easy way to port JavaScript games into iOS platforms, which is attractive to those not familiar with Objective C.

3 Requirements and Analysis

3.1 Problem Statement

ZipApp is a Microsoft service which allows the creation of Windows 8 Apps quickly and easily, providing the resources required to begin development. Once an app template is created, it can be extended by using third party plugins from the website, such as social media plugins and YouTube. The App is fully themed with the modern Windows 8 Metro appearance to be consistent with the operating system and Windows Phone. Finally, the full source code can be downloaded by the developer. They have features which aid users to create utility apps or tools, but not one which enable users to easily create games.

A system will be created which will enable novices with little or no programming experience to quickly get started in making games as easily as possible with the possibility of extending it with plugins. Tutorials and heavy documentation will be written to ensure the user has a strong foot to start developing. A growing market for apps and a recent large interest for mobile gaming begin to appear with the advance in computing power for mobile devices.

3.2 User Requirements

Due to the longevity of the requirements list, it was decided to place them at the appendix of this project. Below is a short summary of the requirements.

Must have requirements

- Menu components such as buttons, labels, switches
- Animated/Non-animated sprite support
- Playback of sounds and music
- A method to store level data
- Parallaxed/Non-Parallaxed Backdrops
- Ability to save and load
- Ability for keyboard and mouse input
- Work on Windows 8 devices and Windows Phone.
- Scalable resolution without distortion of game to support multiple platforms

Should have requirements

- Key bindings
- Particle effects engine
- Water effects
- Surround sound
- Lighting effects e.g. shadows, highlights
- Physics
- AI for in-game characters
- Camera/Map scrolling
- Reward System e.g. points/score
- Tutorials on how to use the game engine.

Could have requirements

- Ability to set preferences of the game template on the ZipApp website prior to download. E.g. Game title, how many buttons

Want to have requirements

- Stock sprites and sounds

3.3 Use Case Specifications

Use cases are a fundamental technique for object orientated analysis. The use cases outline the potential actors which will use the system. As again, due to the length of the use cases, it will be placed in the appendix for further reading.

The actors for the system are as follows:

- Plugins Developer
- Developer
- Student/Novice Developer

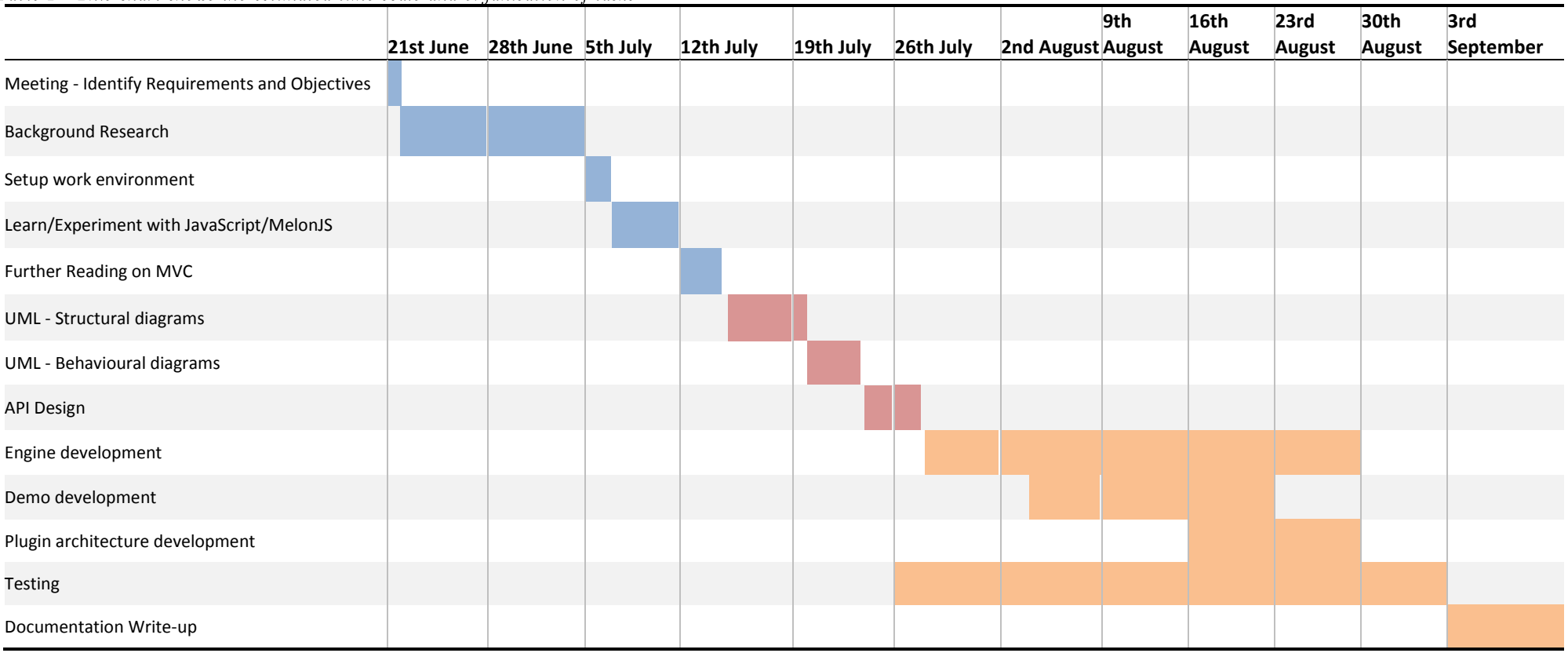
3.4 Visual User Experience

The target audience for this project is primarily beginners to programming; therefore creating something which does not require the user to delve in to the complexity of how a game works in order to make a game. The engine will adopt a Unity-like method of abstraction by using a scene editor and component architecture. Game objects are flexible and can be extended in anyway by adding 'components' to that game objects such as sprites and sounds(Xie 2012). In addition to this, programming will be made to be as minimal as possible, and instead be generated with a tool which takes in user input.

4 Project plan and timetable

4.1 Gantt chart

Table 1 - This chart shows the estimated time scale and organisation of tasks



5 Architecture and design

5.1 Software development lifecycle

Software development undergoes a number of phases. Different models of software development have very similar phases, but the arrangement of these phases influence the way software is developed and consequently affects the quality of the end product.

The choice of software development lifecycle was the V-Model (Verification and Validation model), an extension to the Waterfall model used for small to medium sized projects where the requirements are clearly defined and fixed(Davis, Bersoff et al. 1988). The V-Model and the Waterfall model are the oldest models in software development.

Each phase must be completed before a next phase begins, similar to the Waterfall model. But each phase in the software development cycle is paralleled by a phase in the testing cycle; hence the name verification and validation. Because testing is conducted at the same time, defects can be found at the initial stages of development, which they can be corrected easily. Waterfall development has testing after development, which means requirements must be certain, as correcting defeats after development is a costly process. V-Model is best suited for test activities as these activities start parallel along with each phase of requirement – design and implementation.

Advantages:

- V-Model introduces testing early
- Clear, simple and easy
- Testing and test design happen before development, saving a lot of time, and thus much higher success rate over the waterfall model.

Disadvantages

- Doesn't account for change – very rigid
- There are no early prototypes of the software produced as it is only developed during the implementation phase.
- If there are any changes to the product halfway, tests along with requirements have to be updated.

The verification stages are of the following:

- Requirements analysis: the information about the system is gathered from the end user.
- System design: create a fundamental design of the software.
- Architecture design: interface relationships, database tables, and dependencies are created
- Programming: the entire system is broken up into small modules, which later are assembled to form system(Satalkar 2011).

The validation side is of the following:

- Unit testing: this is where the small modules created are tested to ensure they are good to use.
- Integration testing: the small modules are then integrated into the larger system where errors in interfaces and interaction between modules are determined.
- System testing: testing is carried out in the perspective of the end user and is tested against the system specification.
- User acceptance testing: testing it against the requirements specification. Ensures the software developed meets the acceptance criteria. The tests are conducted in a simulated real time environment(Satalkar 2011).

5.2 General Architecture

The architecture of the system would be composed of 3 main tools. Visual Studio for compiling, Tiled for level creation, and the MelonJS Helper to automate programming. They will be used to generate files for each other to use, so each of the tools must be able to read and write JavaScript which can be read and not give errors. The game framework architecture will be built as a layer on top of the MelonJS base.

5.3 Diagrams

5.3.1 Structure diagrams

Class diagrams were developed in order to show the static view of an application. Class diagrams is not only used for victualing, but it also describes different aspects of the system and how it is connected together(Schmuller 2004). UML Class diagrams were made for the game library and the MelonJS Helper tool which is placed in the appendix.

5.3.2 Behavioural diagrams

Behavioural diagrams show what must happen in the system. The behaviour of a system is illustrated using diagrams and can describe functionality of software systems. Use case diagram is created in the appendix to show what happens in the system when communicating between MelonJS, the programs, and the user.

5.3.3 Interaction diagrams

Sequence diagrams are used to show the interactions between objects in a sequential order that those interactions occur. Sequence diagrams were made for the MelonJS Helper tool and are placed in the appendix.

5.4 User centred design

User centred design is an approach to design that grounds the process in information about the people who use the product. The user's needs, wants and limitations are given thorough attention throughout each stage of the design process(Lowdermilk 2013).

For the game framework, user centred design applies as the user would create all their games through the Tiled editor. The MelonJS Helper will have to incorporate user centred design for it to be effective as a tool to simplify tasks. If the tool is poorly designed and thus hard to use in itself, it is a failure. When an object is placed in the Tiled editor, the properties of the object must be self-explanatory and link closely with the documentation provided and be consistent across all objects. For example, changing the colour of an object should be the same when changing the colour of another object in the properties, and cannot be spelled differently.

Tasks should be simple in structure. Attention must be paid to the psychology of the user, such as the limits on how much a person can hold in memory at one time, and how many

active thoughts can be pursued at once(Norman 1998). The MelonJS Helper tool changes the complex task of repetitive code actions to one click, whilst Tiled will simplify designing levels down to a graphical editor.

We cannot rely on our memory for certain kinds of information therefore providing mental aids to a product are necessary. For example, a notepad may be used to store phone numbers or a grocery list, or a watch to provide the time.

Conversely, there are dangers to oversimplification. A task which is oversimplified can lead to loss of control. For instance, driving an automatic car may seem convenient, but however some people prefer to be able to monitor the gear shifts of a manual car to feel in control. Frozen foods may simplify cooking, however for some, it destroys the essence of the task – Automate but keep much the same(Norman 1998). To avoid over simplification, there is always the option to access the code to make personal adjustments. Each object will also have a large numbers of properties to tweak.

Some tasks can seem complex because of the manual skill required. By restructuring the task into one which requires a different skill, the task can be easier and user friendly. For example, tying a shoelace is a difficult skill to learn, with the invention of Velcro, the nature of task has changed, removing the complex sequenced of skilled motor actions, replacing it with a simpler action, much easier to learn(Helen Sharp 2002).

Assume that any error that will be made will be made – design for error. With this in mind, the user can go back on track to the intended flow of the program quickly. Dialog messages allow the user to know what has happened to reverse the unwanted outcome. Dialogs will be placed throughout the MelonJS Helper tool, if there is a problem, and exceptions with detailed explanations will be given if there is a compile error in the game templates.

Standardising should be the last step to consider if all else fails. By standardising actions, outcomes, layouts and displays, the user only needs to learn how to use a function once, and then they can use the same feature throughout, no matter how arbitrary the standardised mechanism. For example, a QWERTY keyboard, car controls, or clock. One can use any other QWERTY keyboard after learning how to use their own.

5.4.1 Personas

John Smith

Game Design Student at the Royal College of Art, Inspiring game designer.

Background

- 20
- Single
- Ambitious
- Outgoing
- Comfortable using a computer
- Very proficient with creative packages such as Adobe Photoshop or Autodesk Maya

Motivations

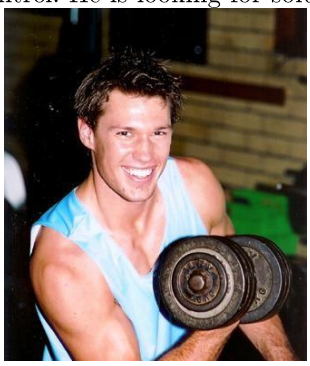
- Playing really good indie games
- Viewing spectacular game concept art
- Getting brawny in the gym

Frustrations

- Playing poorly made games
- Learning programming

John is a frequent gamer who studies how to make the resources for games, such as the 3D models, 2D sprites and the music composition. However, he finds he cannot put his ideas onto platforms without the help of a programmer. He normally works with a programmer, but now he wants his independence, he does not want to rely on anyone. He is starting to learn programming, but is struggling. Nevertheless, he is eager to get started making games.

He is on his way to his final year, and wants to create a game for one of his projects without to depend on unreliable computer scientists and have full control. He is looking for solutions.



Harry Jones

Studied Computer Science at Imperial College. Programmer at the Goldman Sachs, developer.

Background

- 30
- Married with two children
- Ambitious
- Likes to spend time with family
- Very fluent using a computer
- Very proficient with development environments such as Visual Studio, NetBeans, Eclipse.
- Knows C++ C#, Java, JavaScript

Motivations

- Getting his wages every month
- Seeing his two beautiful daughters
- Pictures of luxury cars he will soon own
- Promotion at his job

Frustrations

- Being told off by his boss
- Being told off by his wife

Harry is a full time employee at one of the world's most famous banks. He works long hours, with long commutes with a family to take care of also. He has been a developer for over 8 years and has a lot of development experience.

He loves gaming and would love to improve his portfolio by creating some games. However, he doesn't have enough time with his career and family to juggle. He would like a tool which would allow him to create a game easily without spending too much time on the low level parts.

5.5 Design Patterns

Developers often structure their code intentionally with a design pattern, instead of letting their patterns emerge gradually during development. There is nothing wrong with either approach, but however there are benefits of using design patterns discussed further(Smith 2009).

A pattern is a reusable solution that can be applied to commonly occurring problems in software design. Patterns are not an exact solution, but is there to only provide a solution scheme(Pree and Sikora 1997). Patterns aid the solving of a design problem. The benefits of using patterns are as follows:

- Patterns are solid approaching to solving issues in software development.
- Patterns can be easily reused and serve as an out of the box solution that can be adapted to suit the needs of developers.
- Patterns are also expressive; they contain a structure and vocabulary to the solution that can help express rather large solutions elegantly.
- Reusing patterns assist in preventing minor issues that can cause major problems in the supplication development process.
- Certain patterns can decrease the overall file size of the code by avoiding repetition(Osmani 2012).

There are 4 types of design patterns, structural, creational, behavioural and concurrency patterns. There are different design patterns available in each category of design pattern such as the Constructor the Pattern, the Module Pattern, the Singleton Pattern, the Observer Pattern, the Prototype Pattern, the Facade Pattern, the Factory Pattern and the Decorator Pattern(Osmani 2012) to list a few.

The Module pattern was the main pattern of choice for development. This is a pattern which emulates the concept of classes, something which is non-existent in JavaScript. It includes both private and public methods and variables in a single object. This stops function name conflicts and protects variables and methods from leaking into the global scope. This results in a clean solution for shielding logic why only exposing the interface used by other parts of the application(Pree and Sikora 1997).

The Prototypal patter also has similar features, it allows for public and private methods and inheritance. However there is no standard built-in way of writing code that will look the same no matter which developer wrote it. There is also no standard way to do protected and private variables and methods. Some developers may not like how the libraries are written or choose the write in a different way. The Module pattern has a fixed way of representation, so all developers can understand and relate to the code. It is easier for object-orientated developers to develop as it shows strong similarities(Coplien 2003).

The Singleton pattern restricts the instantiation of a class to a single object. It differs from a static class such that static methods can only be ran from static classes, whereas a singleton is treated as an object, it can be parsed as an argument etc. A singleton's initialisation can be delayed because it may require some extra information not available during initialisation time – where static classes are initialised. Singletons are used when lots of objects require a single object repeatedly such that dependency injection becomes cumbersome. A good example of a valid singleton is a logging class. Every class needs it, and there only needs to be one logger(Rainsberger 2001).

A Singleton object was used in the MelonJS helper. There is a settings object which contained all the directories to all the relevant project files. All of the parsers and screens needed to use the settings object, and there only needed to be one instance of it. It is also used in the game framework as a 'GameManager' object. This class stores data, handles music in a level and there can only be one instance. A Singleton was also used in the HUD layer of the three templates. MelonJS handles the HUD using a Singleton, which if it does exist when attempting to spawn another, it returns the current one. Items can be added to the HUD layer afterwards.

A Façade pattern presents an outward appearance to the application to hide away complexity. It provides a convenient high level interface which simplifies the API being presented to other developers, which improves usability(Stefanov 2010).

It was used in the MelonJS Helper when generating the files. A single method would generate a string representation of the file ready to be written. Unobtrusively, it is calling various methods to format and generate the different variables and methods for each object, screen or plugin.

Likewise, it is also used in the game framework. The 'MainPlayer' player class has a function called 'bindKey()' which binds the selected key to an event. This method calls various methods including Melon's 'me.input.bindKey()' to bind the key to MelonJS and generates a unique ID for a key event so other keys performing the same function in a different object will not clash. This is important for multiplayer games, demonstrated in the fighting game template.

Factory pattern is a creational pattern concerned with the concept of creating objects. It differs because the creation of objects does not require a constructor. Instead, a Factory can provide a generic interface for creating objects, where the type of object to be created is selected(Smith 2012). The factory pattern is useful when objects or component setup is complex; or if an object needs to be generated depending on the conditions; or when many small objects needs to be created with same properties(Osmani 2012).

This pattern was used for particle effects, where a 'particle factory' is made to generate many particles over time. The user specifies the type of particle emitted, at the rate, decay time, and the gravity the user wishes. This can be seen in the platformer template as 'fire'.

This was also made in the character select screen of the fighter template. The user selects a champion and a player is created depending on what character they chose.

The MelonJS Helper has examples of its use. The parser has an 'addCodeBlock()' function which creates either 'Line', 'CodeBlock' or 'Unrecognised Line' for statements, loops and if statements or comments respectively.

5.6 Model View Controller/ViewModel

5.6.1 Model View Controller

A Model-View-Controller or MVC is an architectural design pattern, a method of organising and splitting up the files that make up an application into sensible parts rather than having a huge lump of code. It was created by Trygve Reenskaug in 1979, the creator of the SmallTalk language(Upton 2007).

Models are the data elements in a system, such as databases or excel documents. In this system, it is objects which represent the underlying data. It performs operations on the data to add meaning to it. The model should be unaware of the view and the controller.

Views show the state of the model; displaying the information across to the users. This would be the user interface for an application. For example, in Microsoft Excel it would be the spread sheet, or for a website, it would be the underlying HTML/CSS(Krasner and Pope 1988).

The controller links the view and the model together. The model and the view do not know the existence of each other and communicate via the controller. It offers options to change the state of the model whilst controlling application flow and provide the dynamic data to the views. In a website, this would be the underlying PHP server code which obtains data from its databases and parses it into HTML/CSS presentation layer – the View(Upton 2007). When a button is clicked in the view, the view asks the controller that such button is clicked, and receives data from the model before returning it back to the view.

The advantages of using architectural patterns include:

- When updating an application, it is clear whether the changes to the model/controller or the view, creating easy maintenance.
- Writing unit tests are easy due to the decoupling of the model and views.
- It allows different developers to work on the logic and the UI at the same time(Burbeck 1992).

5.6.2 Model View ViewModel

The MelonJS Helper was created with the use of Windows Presentation Foundation (WPF) for which the Model View ViewModel (MVVM) architectural pattern was defined for. It has found popularity in the Adobe Flex community as an alternative to the MVC pattern.

Like with MVC, the Model in MVVM domain specific data or information the application will be working with. It holds information, but doesn't handle behaviour. It is the Classes and Objects in Visual Studio.

The view formats such data and displays it on screen for the user to interact with. It should not have much computing involved. The view isn't responsible for handling state; it is there to keep in sync with the ViewModel. In Visual Studio, it is the .XAML window pages, written in the Extensible Application Mark-up Language (XAML).

The ViewModel is a specialised controller that acts as a data converter – converting model information to view information and passing commands from view to model. In Visual Studio, this would be represented as a partial class of the XAML page(Shneiderman 2005).

MVVM has a nice data binding feature which provides a loose coupling between the model and the view and removes the need to write any lines of code in the ViewModel and syncs the values between the two in real time(Smith 2009).

6 Implementation

At the start of the implementation stage, the intended route was to create an original game framework. EaselJS was used for the display of graphics on screen, whilst SoundJS was used for the playback of sound, and a custom base game object class was made which could be extended to create other objects. The libraries mentioned are part of the CreateJS suite. The CreateJS suite was initially chosen because of its long development history, well documented API, its extensibility and its huge community and industry support(Skinner 2013).

After a lot of research, the plan to create an original game framework was discontinued as time could be spent on other important aspects which will improve the user experience, rather than focusing on the fundamentals. This will in turn produce a more polished and ready product.

MelonJS was the choice for the base framework for the game template. This is chosen because of its long history in development, its friendly active community, its Object Orientated architecture and most importantly, its native support for Tiled – the free open source tile map creator. MelonJS itself is also open source and is under the MIT licence, which means it can be freely distributed or used under any condition, even for commercial purposes.

Tiled is very important because it will give the user a graphical WYSIWYG (What You See Is What You Get) experience for creating games. Tiled not only lays out the layout of a level, but also allows the placement of game objects such as the player, enemies, and collectables.

MelonJS is built on top of to allow most of the creation of the game inside Tiled alone, removing almost all the programming needed. An additional MelonJS Helper tool is developed in C# to further aid development.

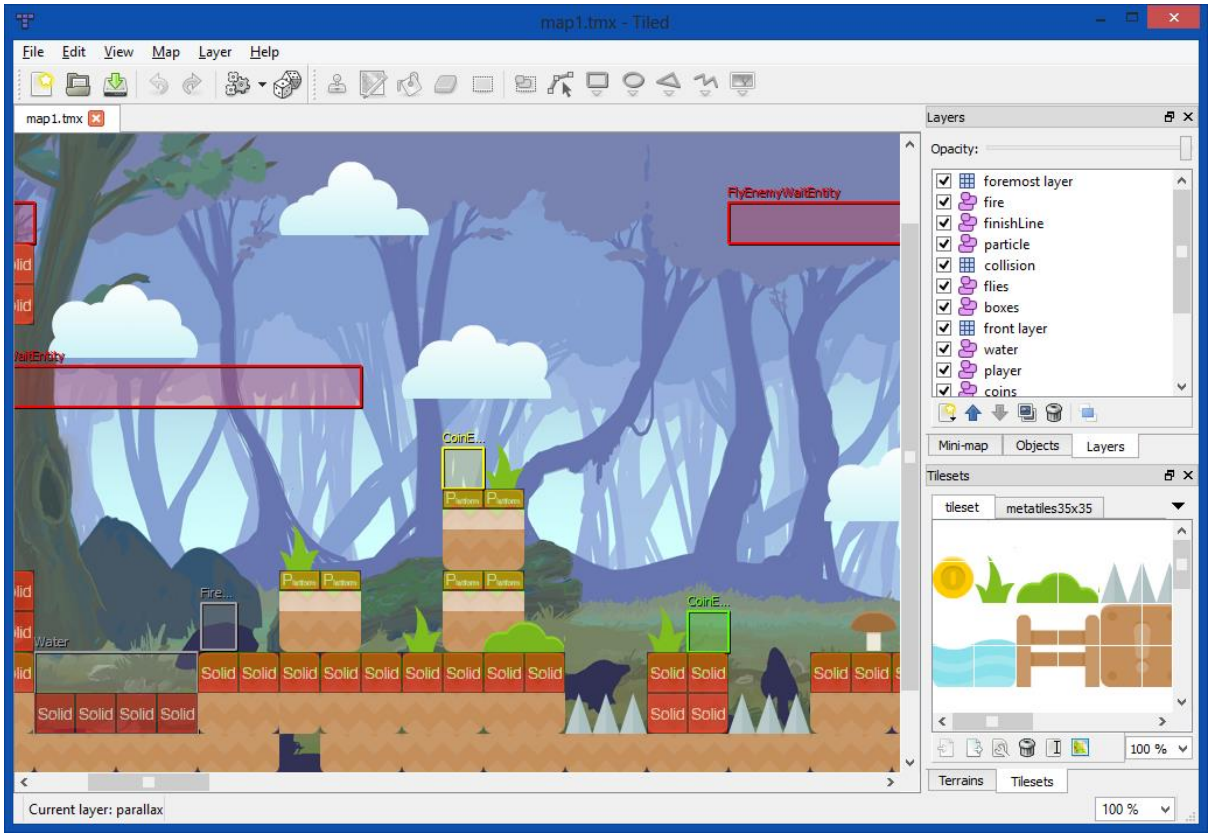


Figure 1 - The Tiled map editor

6.1 MelonJS Helper

This is a tool developed to simplify the development process for the user when using Melon. MelonJS requires multiple repetitive detailed codes to initialise simple features such as Levels, Objects, Assets and Plugins. To improve the development experience for the user and further remove fewer lines for the user to code, a dedicated helper tool with GUI is implemented. This tool will generate all the code required by Melon by requesting data about the program such as the assets folder, level details, plugin details, and files.

The advantage of this is that the developer no longer has to manage such mundane tasks, which will cause preventable errors if a user neglected to add an asset file or object. It will also help the Tiled WYSIWYG editor as every object created will be usable in Tiled without extra labour.

The helper is developed in C# as JavaScript cannot access files on locally due to security enforcements and standards (Stefanov 2008). C# can use the File IO imports which make file manipulation very straightforward.

There are three helper sub-tools available, a resource helper, a screen helper, and a plugin helper. The next sections will outline what each one does and how they were made.

The user interface was aided with the help of the Elysium library(Vishnyakov 2013); a free open source library, released under the MIT licence. It enables aesthetically simple UI design which resembles the modernised Windows 8 environment.

6.1.1 Settings tab

The Settings tab is the hub to store all the URLs to the folders and files for the project. The resources require the assets folder, the entities folder, and a resource.JS file to write on. The

Screens require the play.JS file in the project template. The plugins require the plugins.JS file in the project template. A static Settings class was made so that other objects can pull data from it. The settings can then be saved by pressing the 'Save Settings' button on the bottom. This creates an XML file in the same directory as the executable containing the relevant settings. When the executable starts again, it will search for this XML file, and if it exists, the executable will preload all the settings back. This will therefore not require the user to perform simple mundane task of filling in the file directories every time they launch the helper.

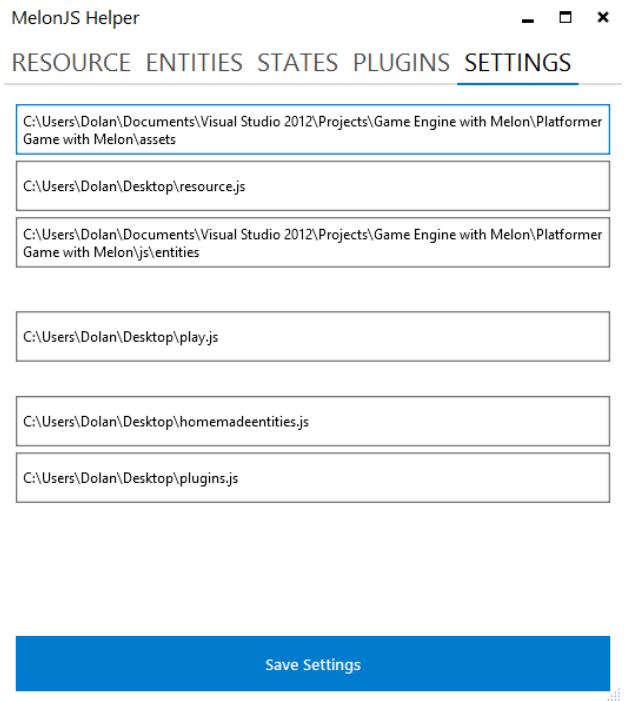


Figure 2 - Settings tab of the MelonJS helper

6.1.2 Resource helper

This tool is a subset of the tool made to abstract MelonJS resource handling. MelonJS handles its resources in an eccentric way as the user must import the files into the project (or upload them into the server in a case of a website), and then the user must reference them in an array which gives its file location, its file type, and its reference name. This is tedious for the user, impeding development progress, and complicating the process.

Similar to resource handling, MelonJS requires the user to add all the objects created by the user into their 'entity pool' in order for it to be used with Tiled. It requires the object name and a reference name. As before, it is tedious for the user to add the object into the entity pool every time an object is created.

As a solution to this, a dedicated resource.JS generator was created to deal with the handling of resource files and game objects by first requesting the projects asset folder (where all the external resources such as images sound files are located), then scanning the directory including sub directories for files. It scans the extension of each file and decides what type of file it is and where it resides.

The generator then requests the folder to all the JavaScript files. It scans all JavaScript files, and uses a regex to find any line which begins with the 'game.' prefix; which indicates a game object to be added. The reference to it is the name of the object minus the 'game.'

prefix. For example, 'game.DynamicLight' would be the object name, and 'DynamicLight' would be its reference.

Finally, the current resource.js file is selected in the Settings tab. When the user clicks 'Create resource.js', it will generate the resource.js file containing all the references to the assets and objects. This will append the current resource.js file.

Every time the user creates a new object within Visual Studio, the user must open the resource helper, scan the resources and generate a new resource.JS file. Although seemingly tedious, this takes up less time, simpler and will guarantee no human errors.

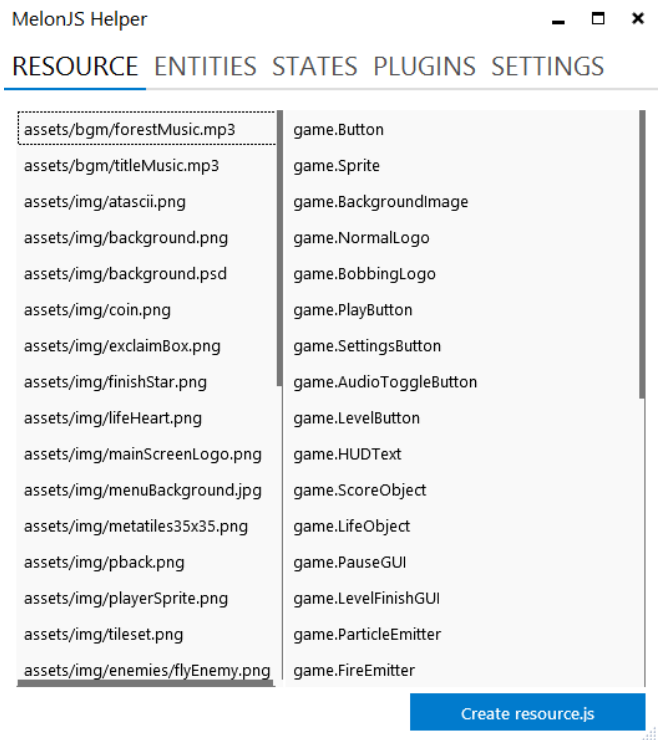


Figure 3 - Resource tab in the MelonJS helper

6.1.3 Level helper

The level helper tool carries out a similar function to the resource generator. Levels in Melon are generated by extending the 'me.ScreenObject' class which will create a screen. Screens are 'views' players see and interact within a game. With each screen, one can draw polygons, draw text, or images, but most commonly, one would want to load a level within the screen. The screen must be associated with a game state of the system with an extra line of code. A game state exists to handle different events differently and draw something different on the screen. Each state will handle its own events, update the game world, and draw the net frame on the screen(Rogers 2012).

By requesting the user to enter details of each screen such as the level it loads, the name of the screen and the state the screen is in, it is possible to generate a screen file.

Each Screen is stored in its own Screen class which has methods, could have methods inside them which define the Screen. The screen's method's 'ToString()' functions were overwritten to generate the JavaScript representation of the class, which is called in the screen's 'CreateJS()' method which generates a string array of lines needed to be written to disk.

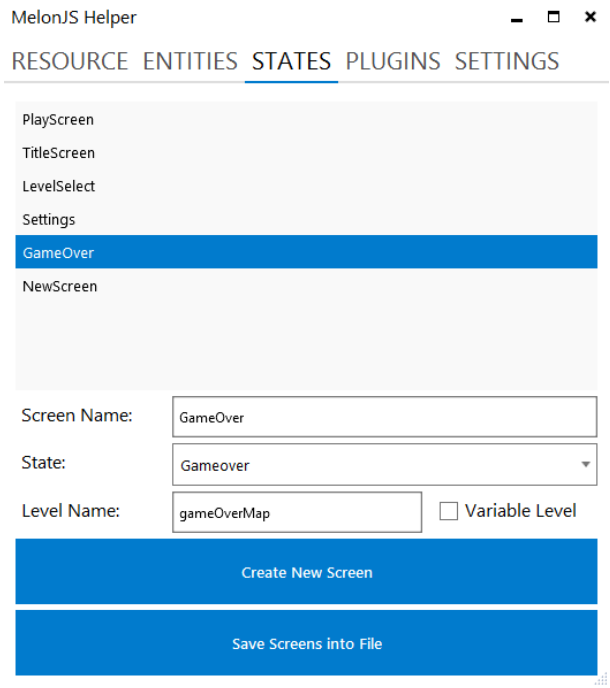


Figure 4 – States tab in the MelonJS helper

6.1.4 Plugins helper

One of the requirements of the project is to provide plugin support. This is already been utilised by MelonJS with their own plugin architecture since version 0.9.5. However, this is not very comprehensive to non-programmers or to newcomers to MelonJS. Therefore a helper is developed which will generate the fundamentals to the plugin for the user to start creating. As with the Screens file, the Plugins file stores lots of Plugin objects which represent the plugin in code form. A plugin contains methods which each has their own 'ToString()' method, which creates a JavaScript depiction of the plugin. It is called when generating a plugin.JS file which is called by the Click event of the 'Save Plugins' button. The following code shows what is generated:

```
(function () {
	var NewPlugin = me.plugin.base.extend({
		version: "0.9.5",
		init: function () {
			this.patch(asdas, asdas, function() {
				//Write your plugin here!;
			});
		}
	});
	me.plugin.register(NewPlugin, "NewPlugin");
})();
```

Now the user only has to write plugins by replacing the "//Write your plugin here!" line.

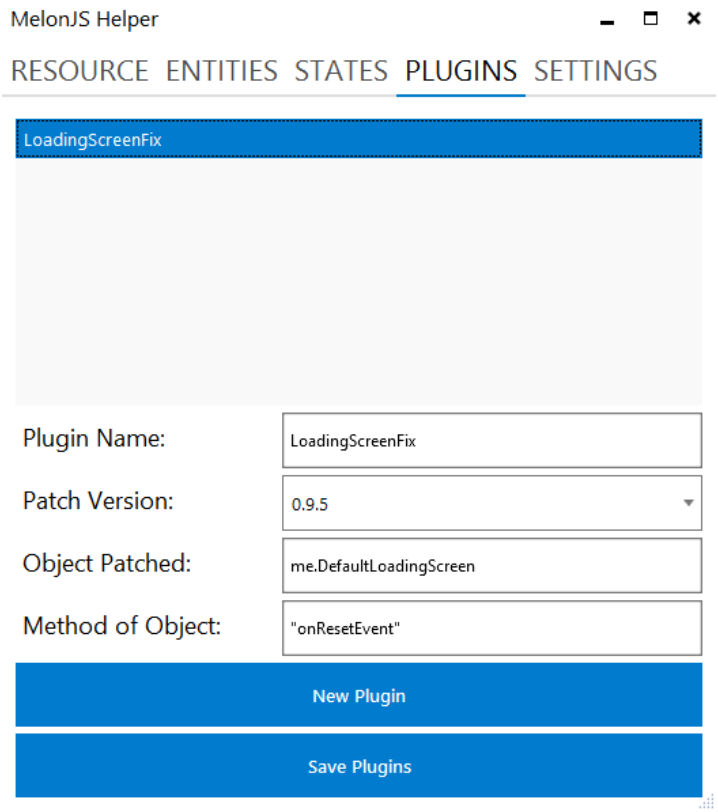


Figure 5 - Plugins tab in the MelonJS helper

6.2 Library implementations

6.2.1 Initial thoughts

CreateJS was very attractive at first as it would provide a solid foundation to work on, offering graphic, sound and animation libraries, which can be easily modified to be suitable for games. Inheritance in the game engine deriving from one base class, the Object and 'GameObject' was desired. The Object class will have methods all objects should have. For example is equal to, to string and clone. The 'GameObject' class what every object on the stage should have. For example, adding a sprite to the object, add sounds it can play, its coordinates, and showing/hiding the object on the stage.

A popular tile map creator called Tiled was used to create '.TMX' files for the game levels. An XML reader was designed for the '.TMX' file type. When the tile map was read, a level object was created with the relevant attributes. The level then searches for the tile sets associated with the tile map and loads them in. A nested for loop was iterated for each tile in the tile map to position the correct sprite for the tile. Each tile was created using CreateJS' createJS.Bitmap() function and underscoreJS' _.clone() function. Cloning is needed because displaying the same sprite twice using CreateJS will just shift the sprite to a new location, so many instances are needed. These individual new Bitmaps were grouped into a two dimensional array which scalar values can be applied to it to shift positions or scale, giving the illusion of a camera.

A menu system was made with the aid of JQuery, and HTML. The JQuery script can retrieve values a user entered into a form on ZipApp and pushed directly into the game itself. For example, the game title and buttons.

6.2.2 Tile maps

To store level data for 2D games, tile maps are used. A tile is a unit of floor space in a game which may (or may not) contain an image which when stitched together with many tiles creates a larger image (tile map) representing the level. An image which contains many tiles stitched together is called a tile set. Historically, this was done to save RAM on storing large levels by re-using a number of smaller graphics and increase real-time rendering performance (Burchard 2013), and easy creation of levels. Most tile maps are of an XML type format, so an XML parser is needed. The parsed information then fetches relevant tiles from the tile sets and recreates the level.

The advantages of using tile map are for the extensibility. Instead of creating huge bitmaps for large levels, tiles make it possible to construct seemingly endless levels on the fly using a few tiles that are repeated to produce a larger image(Rettig 2012). Tile maps save both memory and time as the same tile can be rendered multiple times. This is why tiles are used in geographic mapping applications; it is unnecessary to load the whole map at once, so only the portion of interest is loaded(Rogers 2012). Collision detection using tile maps doesn't use many resources, a technique MelonJS uses, discussed in later sections.

A tile map consists of three components: the map, the tile set and an image which contains the textures for the corresponding tiles. Each tile has its own properties, such as its ID, which corresponds to the position in the map, and a global ID, which identifies the tile texture from the tile-set that are displayed. Tiles are mapped into layers, each of which is a complete map of the 2D area covered with tiles(Rogers 2012).

Each layer is drawn on a different depth layer, so the stacking of layers is important. The higher the layer in the stack, the more "foreground" it will become. For example, a cloud layer should be placed in the top most layers, whereas the ground should be on the lowest layer. Placing the cloud layer below the ground layer will imply the ground is higher than the clouds. Because of this, the lower layers including the objects contained become initialised first, and the top layers become initialised afterwards. This causes problems such as objects depending on other objects becoming initialised before it can be initialised.

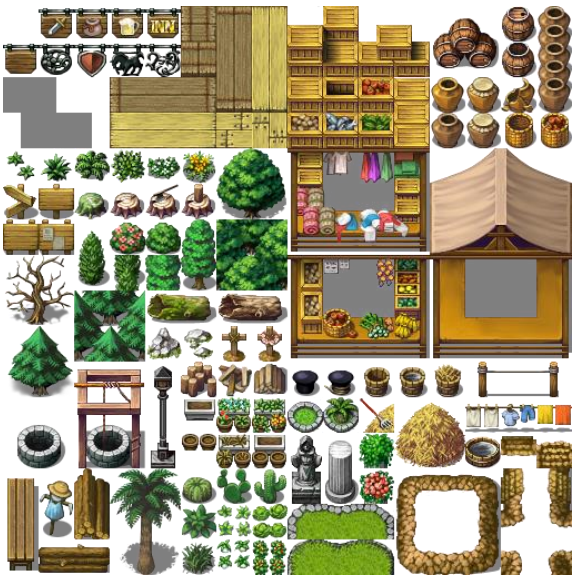


Figure 6 – Example of a tile set(Planet 2010).

A working tile map loader was made which loaded various tile maps. This was done with parsing XML in JavaScript. A parser was written using an XMLHTTPRequest object which

scans through the GIDs of each layer to get their respective tiles and coordinates and creates a 2D array of numbers. The numbers are references to the tile image in the tile-set. The parser continues scanning and fetches key words such as map width, tile height, orientation of the map etc. A level object with all its properties scanned including the 2D integer array representing the level is returned.

To display the loaded maps, EaselJS from the CreateJS suite was used. Each tile was rendered as a new bitmap object, which its image was found by the number it contained in the array element; and its position specified by the array column and row number multiplied by the tile width.

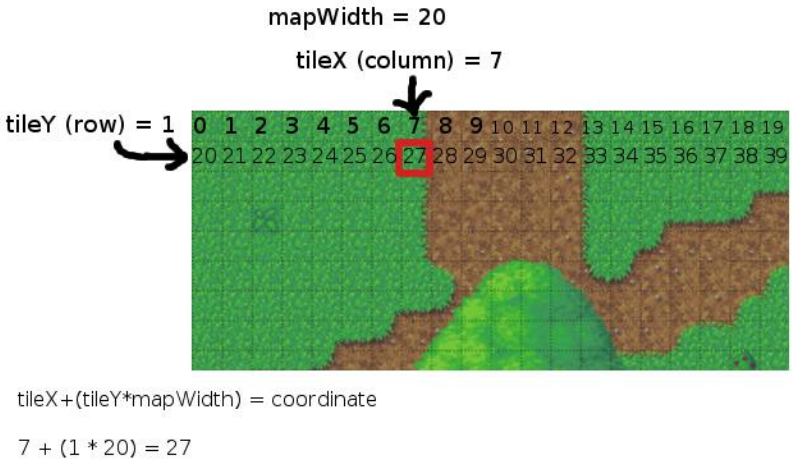


Figure 7 - A tile map showing its GID with the formula of how to convert this to a coordinate(Bruner 2013)

6.2.3 Collision detection

A lot of the collision detection is handled by MelonJS already through the use of the collision layer in Tiled. This is very intuitive for the user since he/she only needs to 'paint' the walls of a level in a graphical user interface.

A real time collision detection algorithm(Yang, Cheng et al. 2005)

Recent advances in real time collision detection(Yoon, Kim et al. 2010)

Collision detection for deformable games(Teschner, Kimmerle et al. 2005)

Figure 8 - Collision boxes in the Tiled editor program.

However, the problem arises when collision detection is for top down view games. There is an issue where if there is no gravity, collision detection between vertical walls is unstable and objects will clip through at certain angles. This is an easy bug to execute and is a key weakness in the game platform. Therefore, a custom collision detection method for the vertical direction was created to fix this problem.

The method works by grabbing to suitable x and y coordinates needed for consideration. If the player is traveling left (velocity in the x direction is negative), then it will obtain the left most coordinate of its collision box, which is defined by MelonJS. If the player is travelling right (velocity in the x direction is positive), then it will obtain the right most coordinate of its collision box. The y coordinate is the top of the collision box. Once the coordinates are obtained, it searches for that tile given the coordinates. If that tile is a collision block, then its velocity in the x direction is 0.

This method fixes most of the problems, however if the object is significantly bigger than the tile size of the level, the method will not work correctly if the vertical wall is smaller than the height of the object. On the other hand, it is unlikely that objects will be much bigger than the tile size in top down view games. (Adams 2009)

6.2.4 Water

The water created was for side-on view 2D games such as platformers. The water responds appropriately to forces which the player or objects apply. The water also responds appropriately to the magnitude of the velocity of the object in question. For example, a greater drop into the water will create a bigger splash than simply walking through the water. This was implemented by emulating physics. The water is simulated by splitting a rectangle into rows of columns. The more columns there are the higher the resolution of the water. The water has a dampening constant, tension constant and a spreading constant. If a body touches the water, the speed of the incoming body is used determine how much the initial displacement of the column is. The column displaced corresponds to where the body is. This initial displacement causes the adjacent columns to displace triggering a cascade of displacements, emulating water ripples. At each update tick, the adjacent displacements are dampened with appropriate tension for accurate simulation. This is very computationally demanding, so for higher frame rates a low count for the columns per unit length was chosen. Particles of water are created at the point of impact to simulate splashing. These particles are coloured vector circles to increase performance.

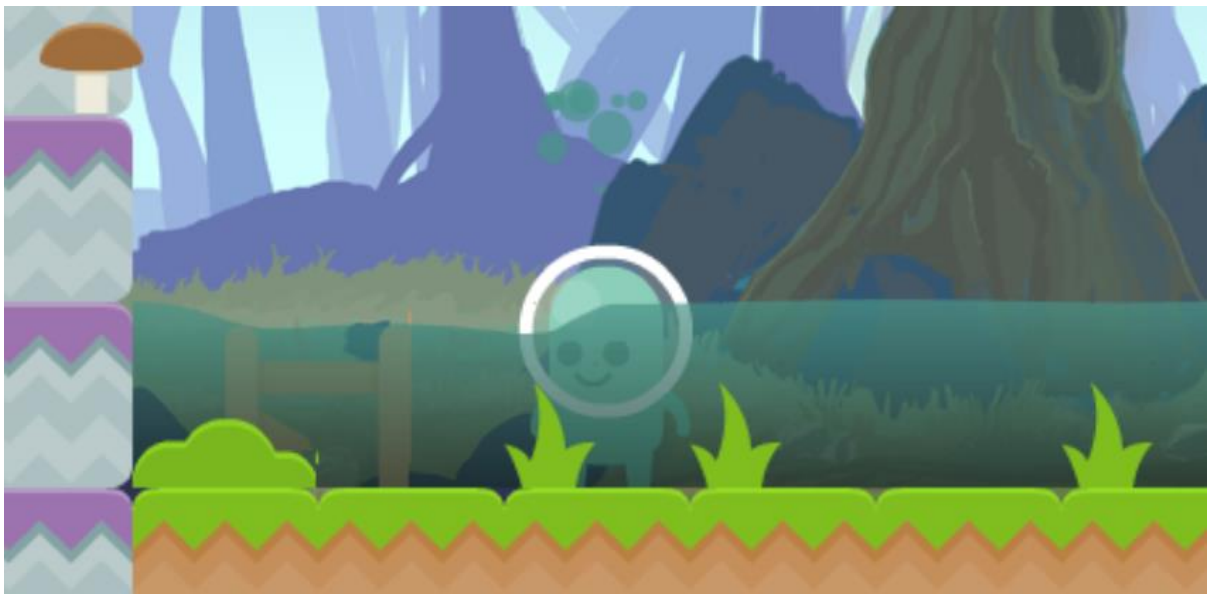


Figure 9 – The dynamic water in the platformer demo created.

6.2.5 Enemy

6.2.5.1 Platformer enemies

The default enemies created for platformer style games are all decided within Tiled. The enemies stroll back and forth in a horizontal straight line. To create an enemy, the developer creates a block and names it the enemy object. The width of the box indicates the path the enemy will travel. A 500px box will cause the enemy to travel 500px left and right. The sprite then needs to be specified. All the sprite flipping is done automatically for the user. Gravity is automatically applied to these enemies, along with collision detection.

Other enemies such as flying enemies can be made which inherit the basic enemy object. These are enemies which are not affected by gravity. Another enemy is a wait enemy. These enemies do not conventionally move back and forth, but instead have a probability to stop and ponder before moving off again. This makes the game less systematic and more challenging as the enemies are no longer predictable.

6.2.5.2 Stealth Game

The enemies in the stealth game are more sophisticated. It uses the A* algorithm for movement and ray-casting for detection. These are explained in the later sections. The enemies also have different states it can be in, neutral, suspicious and alerted. The enemies move around in a pre-defined shape sketched in Tiled. This shape is a polygon which can contain as many sides as possible. This shape is then converted to a valid path using the A* algorithm in which the enemy will follow. The direction of which the enemy is facing is found by taking the difference in it's before and current positions. This direction is used in the ray casting and the mini-map, which is discussed later. By knowing the direction the enemy is facing, it is possible to limit its field of view, so the enemy can't see the player from the behind.

If a ray can be cast from the player to the enemy and it's within its field of view, then the enemy can see the player. At first the enemy will be in the suspicious state and path-find its way to the player, if the enemy is close enough to the player, the enemy will be in the alerted state. The enemy will run towards the player much more quickly and only regain back to the neutral state if the player hid for a certain amount of time. The enemy gives up and returns back to its patrol route.

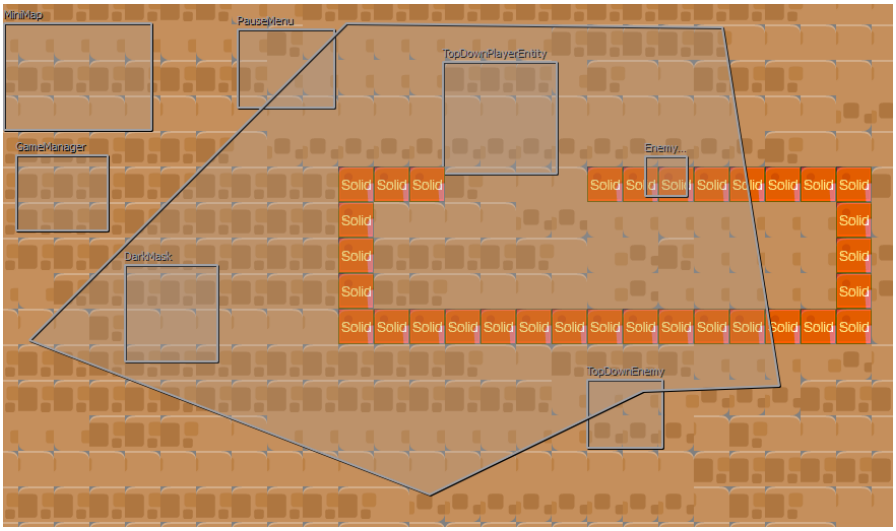


Figure 10 - A level in the Tiled editor, showing all the elements in the level

In the figure above, the path of the 'TopDownEnemy' can be seen. This demonstrates clearly that although the polygon path intersects the wall, the A* algorithm can resolve a path which would not go through the wall, but instead go around the wall.

6.2.6 Particles

Many polygons from the particles displayed on the screen will definitely produce some form of lag and therefore proved to be a problematic process. However, this was solved by using primitive SpriteObjects in MelonJS and simple vector polygons and keeping the polygon count to a minimum. Initially, collision checking on the particles were included (water particles would create ripples themselves), but this lowered the frames per second substantially. As a result, collisions on particles wasn't included, however, it hardly reduces the visual effect.(Kuryanovich, Shalom et al. 2012)

For an emitting particle source for example a fire, a loop was added which will fire a particle every certain amount of time passed. This particle will have a life span of a couple of seconds and destroys itself. For example, the fire displayed on the platformer demo produced a fire particle every 500ms with a life span of 2 seconds.

6.2.7 Lights

One of the ways to achieve realism in lights for games is to add flickering. This does not apply to some lights, but for fires and candles, this was required. To approach this, a vector circle was drawn with a radial gradient from the centre to the edge, from semi translucent orange to transparency. This creates an appealing aura which compliments other objects mimicking an effect known as diffuse lighting. A random value for the alpha of this object was given at each update frame to simulate flickering.

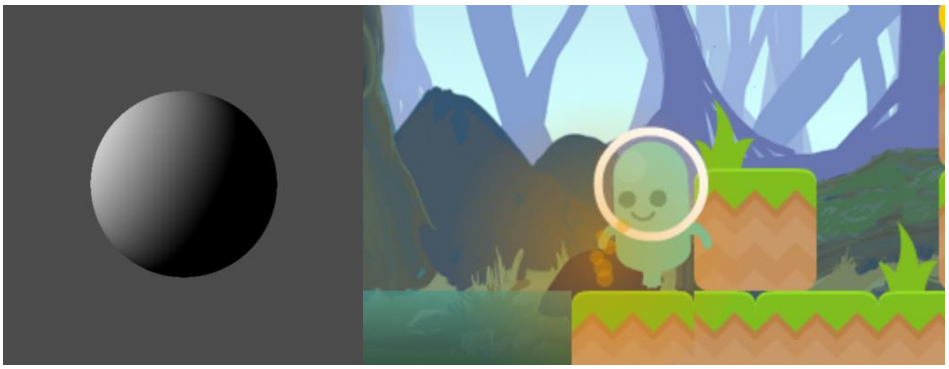


Figure 11 – From left to right; (a) Example of an object under diffuse lighting; (b) The game platform mimicking diffuse lighting

6.2.8 Path Finding

Path finding is a very fundamental part of top down games. It allows entities to travel to different points on the map avoiding potential obstacles. Dijkstra's Algorithm is a path finding algorithm made in 1956(Johnson 1973). This algorithm uses weighted nodes to signify the amount of 'energy' it takes to travel between each nodes. Each problem consists of a start node and an end node (source and sink)(Dijkstra 1959).

Heuristics are techniques used to estimate the solution, which is not guaranteed to be optimal but will speed up the process. By adding heuristics into Dijkstra's Algorithm, the A* algorithm is formed. Comparing this tedious process (going through all the nodes like Dijkstra), it is much quicker and thus is often used in games and other high performance applications. (Zhan 1997)

There are many heuristics for A* algorithm including Manhattan, birds eye, gateway heuristic, boundary heuristic and recently a new heuristic was found which claims to be better than the Manhattan heuristic – the portal heuristic(Yngvi Bjornsson 2006).

The Manhattan heuristic was used however due to its simplicity and low memory usage – crucial for games. Very accurate routes aren't required for this purpose. In a study for most of the main heuristics, the Manhattan heuristic used the least amount of memory, came the slowest. The Portal heuristic used the most memory and was 5th fastest and the Boundary heuristic used moderate amounts of memory and was the fastest(Yngvi Bjornsson 2006).

The Manhattan heuristic works by counting how far each piece is from the goal. The sum is the number of moves possible. It is called the Manhattan heuristic based on the way Manhattan city floor plan is laid out(Lester 2004).

The A* algorithm for the game platform was done by first analysing the collision tile layer. For every tile, a node is created which contains its parent, its cost, its f value, its g value, its h value, its visited state, and its closed state. If it is empty (no walls present), its cost is zero. If it is not empty (there is a wall), then its cost is 1.

The search algorithm takes the start node and end note coordinates and finds the correct nodes appropriately. A heap is created which will hold the current nodes it has traversed. The start node is pushed onto the heap. The current node is created by popping the most recent node off the heap, which in this case is the start node. The neighbours are then found for this particular node. There can be a maximum of 8 neighbours, 4 if diagonal neighbours are not considered. For each neighbour, if it has not been visited, or its neighbours calculated g score is less than the neighbour's current g score then, its g score is newly calculated by simply adding the current nodes g score to the neighbour nodes g score.

Its heuristic score (h score) is found by using the Manhattan heuristic. The parent of the neighbour is set to the current node. Its f score is calculated. Its visited state is set to true. The neighbour nodes are now pushed onto the heap.

This is repeated until the current node in question is equal to the end node, which signifies a path is found. By tracing back the parents of each node starting from the end node, the path can be retrieved.

6.2.9 Dynamic Light

To enhance realism in a game, it is important to include elements which emulate lights in reality as closely as possible. A light which is blocked by objects and cast shadows are a few of the necessary attributes. In the stealth game, this was needed due to aesthetic reasons as well as for the game mechanics also.

The way this is achieved is through the use of ray-casting. Ray-casting is a well-known method to determine if a line between point A and B can be drawn without obstruction. developed first by Arthur Appel in 1968(Appel 1968). Rays are casted from a central point "the eye" to all points, and find the closest object which blocks that path of the rays. This is computationally unacceptable, so some heuristics, assumptions and simplifications must be applied. Ray-casting has many uses such as optimising rendering speeds by only rendering the faces which the user can see – back face culling(Wimmer, Giegl et al. 1999).

Id Games first introduced the first games and ray-casting on the PC platform through their game Doom(Kushner 2002). Lines from every pixel on the screen were extended from the player's position in the direction the player was facing. When it strikes a surface, the pixel

corresponding to the line on the players screen is painted the colour of surface the line hit. This technique will not waste resources on rendering surfaces which will never be shown(Kushner 2002). Doom makes a simplification to only ray cast along on plane, this immensely speeds up performance and is truly modelled as a 2D game illustrated in figure 8b. This is close to a solution the project requires.

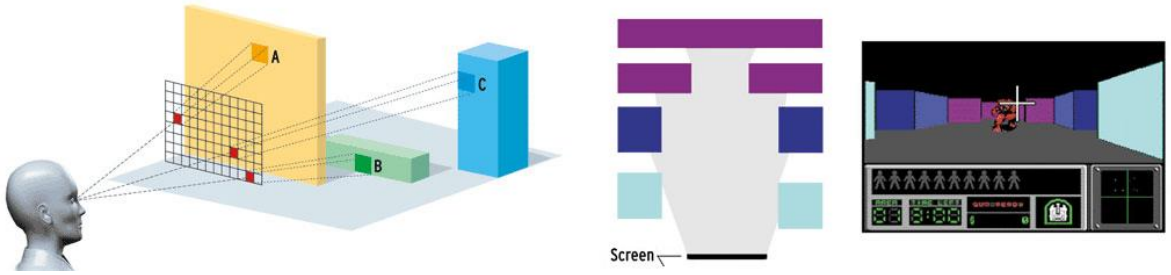


Figure 12 - Diagram demonstrating Ray-casting. From left to right: (a) 3D view of ray casting. The grid is the screen with lines firing off. The cuboids represent objects. Pixels are only sourced when needed. (b) A 2D simplification used in Doom. (c) The rendered Doom game.(Kushner 2002)

Giertsen devised an optimisation to ray-casting by the use of coherency to speed up rendering. He performs a sweep of a mesh in three-space, using a sweep-plane perpendicular to the floor, thus, the viewing plane is the x-y plane, parallel to the floor. A diagram of the orientation of shown below.

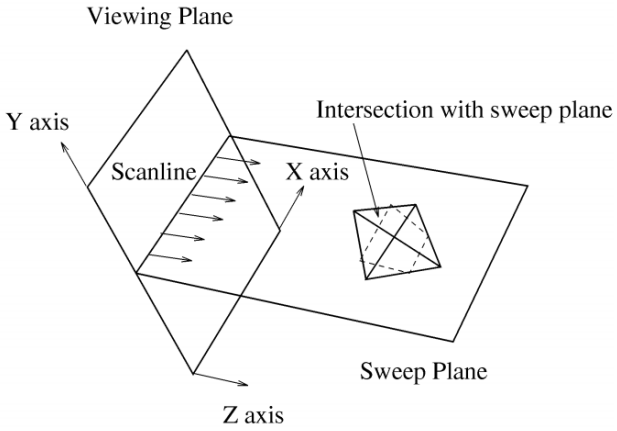


Figure 13 - Giertsen's ray casting method showing the sweep plane perpendicular to the y axis, used in sweeping three-space(Giertsen 1992)

Firstly the algorithm takes all the vertices in the world coordinate system to the view coordinate system using matrix transformations, and then the vertices are sorted by y-coordinate and put in a priority queue array. He then creates an active cell list (ACL) which represents the sweep status – a list of the current cells currently intersected by the sweep line. While the priority queue array is not empty, pop the queue. If it is a vertex, then update the ACL (add/delete the cells incident on the vertex. If it isn't a vertex, it will render the scan-line. By sweeping three-space, it reduces the ray-cast problem into a two-dimensional cell sort problem(Giertsen 1992).

An algorithm was initially developed inspired by Claudia's Lazy Sweep method(Silva and Mitchell 1997) and Amit Patel's 2D Visibility Algorithm(Patel 2012) to make use of a sweep-plane approach. A sweep-line is swept across the plane 360 degrees around a point, much like RADAR. The algorithm works by casting lines away from a central point and if it hits a solid object, the line stops, if it doesn't, then it continues until it reaches the end of the map.

This was optimised to only cast rays to the edge of walls or blocks. As for every two rays make one triangle, it is possible to fill these triangles to simulate light. The algorithm has a sweep function which scans 360 degrees to find the endpoints of the walls, and calculate which wall is nearest for each ray to decide which point it stops at.

It gave the desired results, but due to the speed of the algorithm, it unfortunately cannot be used in a game. Therefore a script called illuminated.js is used instead which is designed to give aesthetically pleasing lights.(Renaudeau 2012)

Illuminated.js requires polygons to be parsed to the light elements in order for it to calculate the shadow casting. This requires an algorithm in its self and is discussed below.

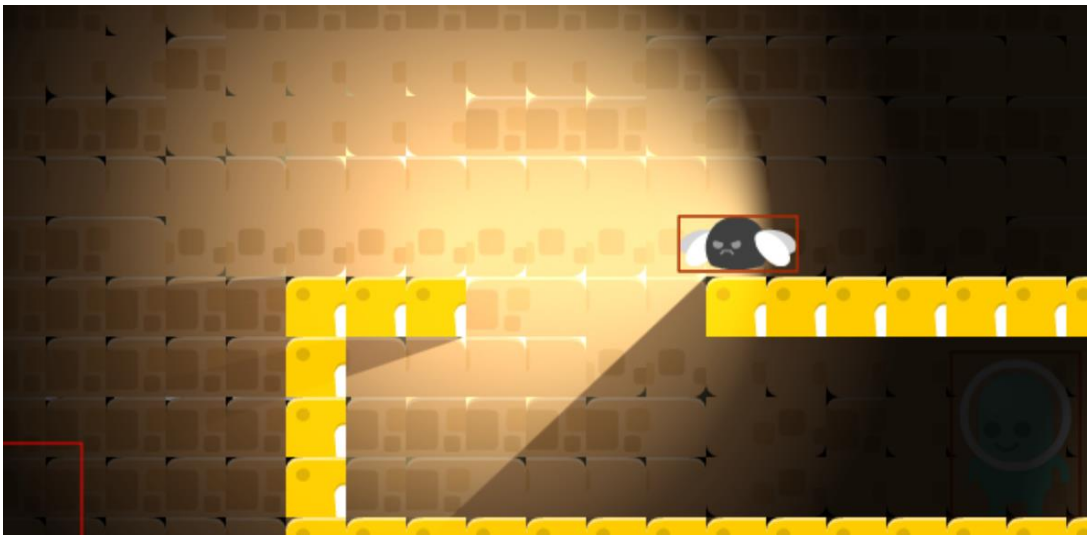


Figure 14 – An early screen shot of the top down stealth game demo. It shows an enemy emitting a dynamic light

6.2.10 Level Tracing

The initial idea was to create a square for every wall tile in the level. Although, this did work to a certain degree, it had numerous problems. It was slow because there were many vertices and edges for the level trace (four multiplied by the number of tiles), and it had artefacts such as light leakage. This is a bug in illuminated.js where the ray casting algorithm cannot resolve tightly packed objects, and instead will produce slits of lights through each wall tile. It looks unpolished and diminishes the immersion of the game.

Therefore, an algorithm which minimises the amount of vertices needed to represent walls was developed. This increased the frame rate and removes the artefacts. It works by searching through all the collision tiles in the level. If it hits one, it has already been traced before it will skip to the next tile. If it is not, then the algorithm will commence. The algorithm has three parts, the adding part, the sorting part, and finally the reduction part.

The adding part of the algorithm focuses on adding the appropriate vertices to the 'vertex pool'. For each tile, it will add up to four vertices depending if it's already added or not. If it's already added, it will skip. This will remove all redundant vertices and thus a complete polygon for the wall segment. Upon running the code, all the vertices are in mixed up positions producing a very ugly glitch-like polygon.

The sorting part of the algorithm sorts the vertices into an order which will places the vertices in the correct orientation. It works by exploiting a mathematical observation. If one were to follow an object clockwise, their right hand will be touching its 'walls' and the left hand will always be free, and vice-versa for anti-clockwise. This is true for every object and

is therefore the reason why hugging the right side of a wall in a maze will eventually solve it. Following this principle, for each vertex, its neighbours were found, and will return the neighbour which has its left neighbour free (clockwise algorithm). Repeating this for every vertex will sort it accordingly and produce a nice result; however it is inefficient because it is using multiple vertices for a single straight edge.

The reduction part of the algorithm firstly finds the first vertex of each wall segment. Then it steps to the next vertex and notes down the change in direction from the start vertex. It repeats this until there is a change in direction, in which the vertex before the direction change is added to the vertex pool. This is iterated until the whole shape is produced.

For my 35 block example, it reduces the vertex count from 140 to 12, saving massive computational time.

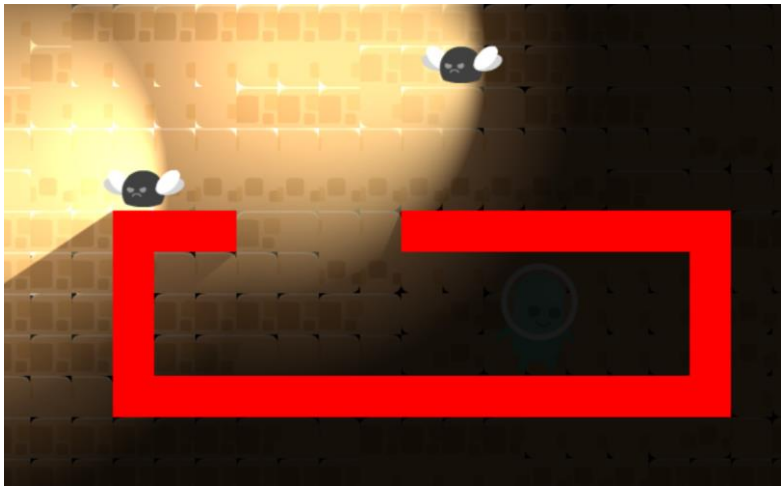


Figure 15 - The polygon generated by the algorithm highlighted in red

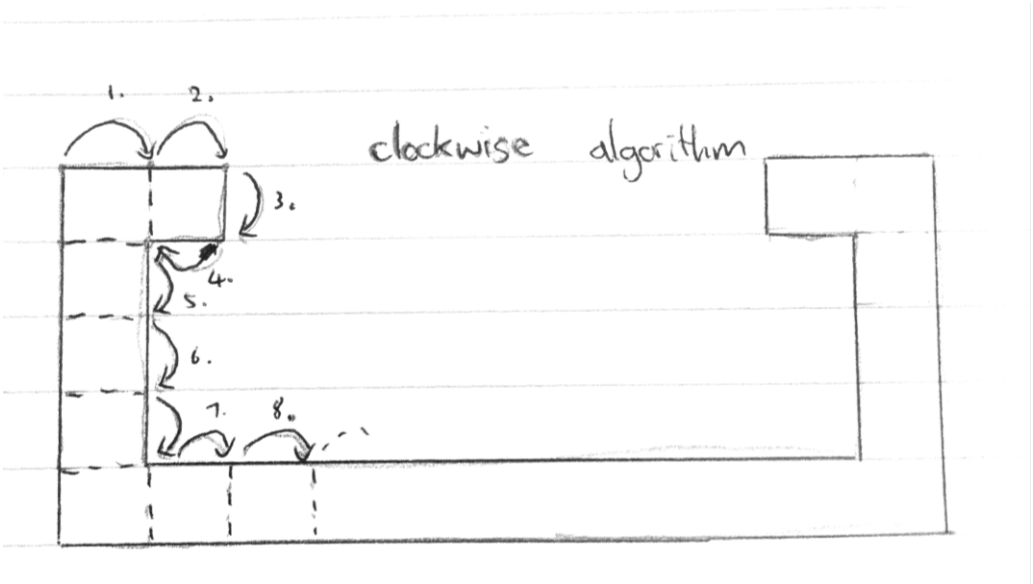


Figure 16 - An illustration of how the algorithm works. The numbers represent the steps it goes through

6.2.11 Hiding and Detection

The ray-casting algorithm developed initially for the dynamic lights before switching over to illuminated.js was not all lost as there is another use for it which would not consume much resource. A ray is casting from each entity onto the player. If the ray is unblocked by walls

and other obstacles, then it is assumed that the enemy can 'see' the player. This works by comparing each wall segment (four segments per wall) collides with the line between the player and the enemy. If it does, it signifies that the enemy cannot see the player as there is a collision. This is a slow algorithm with O(n) speed. This means the speed of the algorithm slows down linearly depending on how many walls there are in the system(Zimmerman 2004).

Another touch added was to limit the angle of which the entities can see by calculating the angle two points create and determining if that is within the allowed range. By doing this, a field of vision or 'vision cone' is created for the entity.

To limit the distance in which the entity can see, the Pythagoras of the entity's position and the player position is calculated and if it is below a certain number, then the entity can see the player.

6.2.12 Mini-Map

It is important in some games to have a mini map to give a sense of context of the positioning of the enemies and objectives. Fitting everything on the screen would be impractical, but without the mini-map, it will confuse the player.

It works by scanning the collision layer of the map, then for each collision tile, it creates a 1px by 1px square and positions it with the pseudo formula below:

$$position.x = \frac{tile\ position.x}{tile.width} + minimap's\ position.x - \frac{players.position.x}{tile.width}$$

$$position.y = \frac{tile\ position.y}{tile.height} + minimap's\ position.y - \frac{players.position.y}{tile.height}$$

This will give a reasonably scaled outline of the level. The end term of the expression is to shift everything slightly in order to make everything relative to the centre (the player's location). For the enemies, the same process is done as with the tiles. However, a vision cone is drawn on the screen with its origin at the point. The draw method takes a direction argument; the direction in which the player is facing. This is important because it resolves where the vision cone should be facing, to imply where the enemies are looking at.

The player is simply a dot in the middle of the mini-map screen. Parameters for the mini-map such as the colour of the walls, colour of the enemies and colour of the player can be changed within the Tiled object.

JavaScript Canvas has a method called 'clip'. It allows shapes drawn on canvas to only be shown within a certain polygon; similar to masking in Photoshop. With this, a box is made with the size of the mini map which hides everything outside it and shows everything inside it. To contrast the mini-map and the game screen to make the mini-map more visible, the clip box is filled with a semi-transparent black colour.

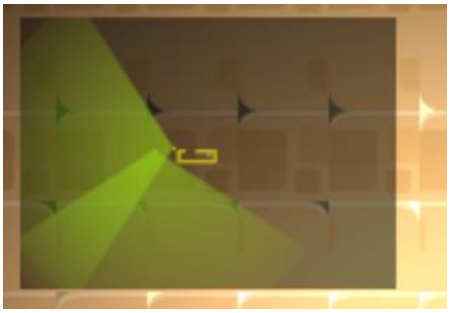


Figure 17 - The Mini-Map showing a level's walls, the player and two enemies

6.2.13 Finish Lines

To further make the developing experience easier for the end user, pre-set finish line objects were created. These objects transport the player to a certain map once the player comes into contact with it. The transportation code is already done within MelonJS, but it requires coding to use. Therefore, an object is created which takes the level name as an argument. The finish line can then be created, positioned, and be given the map to be transported to in the Tiled editor. In the platformer demo, the finish line took the user into the level finish map, which displays the score. It doesn't necessarily need to be another level of the game.

Different finish line objects have also been created which inherit off the one base finish line object, such as the floating/bobbing finish.

6.2.14 Camera

The original method to control the in MelonJS is to set the viewport to track a certain entity. This was found to be very limiting as extra camera effects could not be done. A wrapper camera class called 'MultiCam' was made to address this issue. It is an object which can be placed on stage which when given certain parameters, it produce the desired camera effect. MelonJS' camera is locked onto the MultiCam object. Thus when the camera object is moved, MelonJS will shift its view also. It means the MelonJS camera can be manipulated by simply shifting the position of the MultiCam object.

The camera can track multiple entities or one entity by specifying the 'entities' property in Tiled. If one entity is given then it will track the one entity; if multiple entities are given separated by a space and/or a comma, it will track the middle point of those entities. To find the middle of a large number of points is computationally demanding, so algorithms such as Convex Hull have been developed to obtain an outer polygon outlining an edge which surrounds a set of points.

The camera had a smoothness parameter which when given will ease out the camera movement, causing the camera to tail the entities exponentially slower as it gets closer to the entities.

Default behaviour for the camera can be obtained by specifying the 'deadZone' per parameter. This is the value of which dictates when a camera moves depending on how near the edge of the screen the tracked entity is. A smaller value will cause the camera to move if the entity is shifted slightly off centre; a large value if the object is really off-centre. A value of 0 will lock the camera onto the entity. The axis of which the camera can be tracked is specified by the axis property in Tiled. Horizontal, vertical and both axis tracking can be specified by the axis property in Tiled. This runs an internal MelonJS command.

6.3 Three Templates

By using the libraries, Tiled and the MelonJS Helper, three templates of different action genres were created. Each one of these games was made by first creating the levels needed with the use of Tiled. Objects are then added into Tiled by creating a box and specifying the name of the object in the name field in its properties. Additional properties can be added depending on the object. For example, game controls, speed of enemies and sprite etc. They then were imported into Visual Studio 2012. Assets and other game files were also imported into Visual Studio 2012. The MelonJS Helper tool then helps generate the resource.JS file required to load the files into MelonJS by requesting the games asset directory and entities directory. States then need to be created using the MelonJS Helper by specifying what screen it displays, the state it's in, and the name of the state.

After saving the states and generated the resource.JS reference file, MelonJS will automatically load the screens and the assets into the engine and the game will be ready to run.

6.3.1 Platformer template

A platformer game was created which has a protagonist wandering around in a jungle environment. It utilises particle effects, water, enemies and various screen effects. The goal is to get to the end finish line.

6.3.2 Stealth Espionage template

A stealth espionage game was created in which the goal of the game is to avoid the enemies. The enemies have AI and would chase if it sees the player using A-Star path-finding. A mini-map can be seen on the top which will give a representation of where the enemies and the walls are. Collisions have been remade

6.3.3 Fighting template

A fighting game was made which demonstrates the MultiCam multiple entity tracking feature. Two fighters are on stage which the MutliCam tracks. The objective is to knock the player off the edge of the map. The more damaged the player, the more easily the player can be knocked off the stage. When this happens, the player loses a life. When all lives are lost, the other player wins the game.

7 Testing

Validation is a process designed to increase confidence that a program functions as intended. With validation comes testing: a critical part of the system life cycle which will improve and polish the product. The desired outcome of testing is a guarantee that a program satisfies its specifications. Testing can only show the presence of errors, not their absence(Olan 2003).

Test driven development (TTD) is a desirable and effective way to develop software. It involves designing test cases first, and then developing the classes which will satisfy the tests. Classes are implemented incrementally, with tests performed whenever a change is made. The advantages of TTD are that it makes you consider what you really want from the code, thus saving writing unnecessary code, thus simplifying; it creates a detailed specification; reduces time in rework; less time is spent debugging; informs the developer whether the last change has broken previous code. (Beck 2002)

7.1 Unit Testing

The growth of object oriented programming has influenced the way software testing is approached. Unit testing is where individual units of source are put under a tests with control data and usage procedures to see if they working appropriately. It checks if the unit outputs the expected result from the data given. In object orientated programming, a unit would an entire interface such as a class, or it can be as small as a method(Xie, Taneja et al. 2007). Unit tests are normally written to ensure it meets the design and works as intended. Its implementation can range from being manual (pencil and paper), to being fully automatic.(Lindley 2010).

Automated tests are essential in software development. Each unit test can be ran on a unit of code repeatedly automatically without human interaction, so when something goes wrong, it can be instantly known.(McFarland 2011)

A testing framework is required to carry out the testing. Writing a testing framework from scratch tailored towards the project would be ideal, but it would require a lot of work to cover all the details and special requirements of testing JavaScript code(Lindley 2010). Due to the time restraint of this project, a well-developed testing framework will be used.

There are many JavaScript testing frameworks, such as JSUnit, RhinoUnit, crosscheck, jsspec, spec, Screw.unit and Jasmine etc. Jasmine was chosen because of its smooth integration to Behaviour driven design.

Testing the game framework was limited due to its integration with MelonJS. Everything created for use with MelonJS such as 'ObjectEntities', 'HUD_Items' become instantiated through MelonJS and interact with other MelonJS objects in the MelonJS environment. Because of this, and the requirement for 'events' to occur within the game world, testing was limited. Static and helper classes however were tested.

7.1.1 MelonJS Helper Testing

MelonJS Helper was written in C# so a C# testing framework was used. NUnit Framework was used because of its integration into Visual Studio and one click test feature with

Refactor. Firstly a new project was created which would contain the tests. Then the NUnit framework DLL was referenced in the project.

With Refactor and NUnit, it was possible to run the tests within Visual Studio while working on the code simultaneously. Tests were written by first calling the [Test] tag before a method. This allows NUnit to know that it's a test. Then various methods were implemented which tests certain aspects of the unit.

Finally an Assert.Pass() or Assert.Fail() was called at the end which will determine whether the unit passed or failed the test. Tests were created for the most applicable units of the system including the XML parsers. Tests written are included in the appendix. To create a final representation of the test outside of Visual Studio, the tests were compiled into a DLL which was then opened with the NUnit standalone executable. This executable runs test independently of Visual Studio and produces a result shown below.

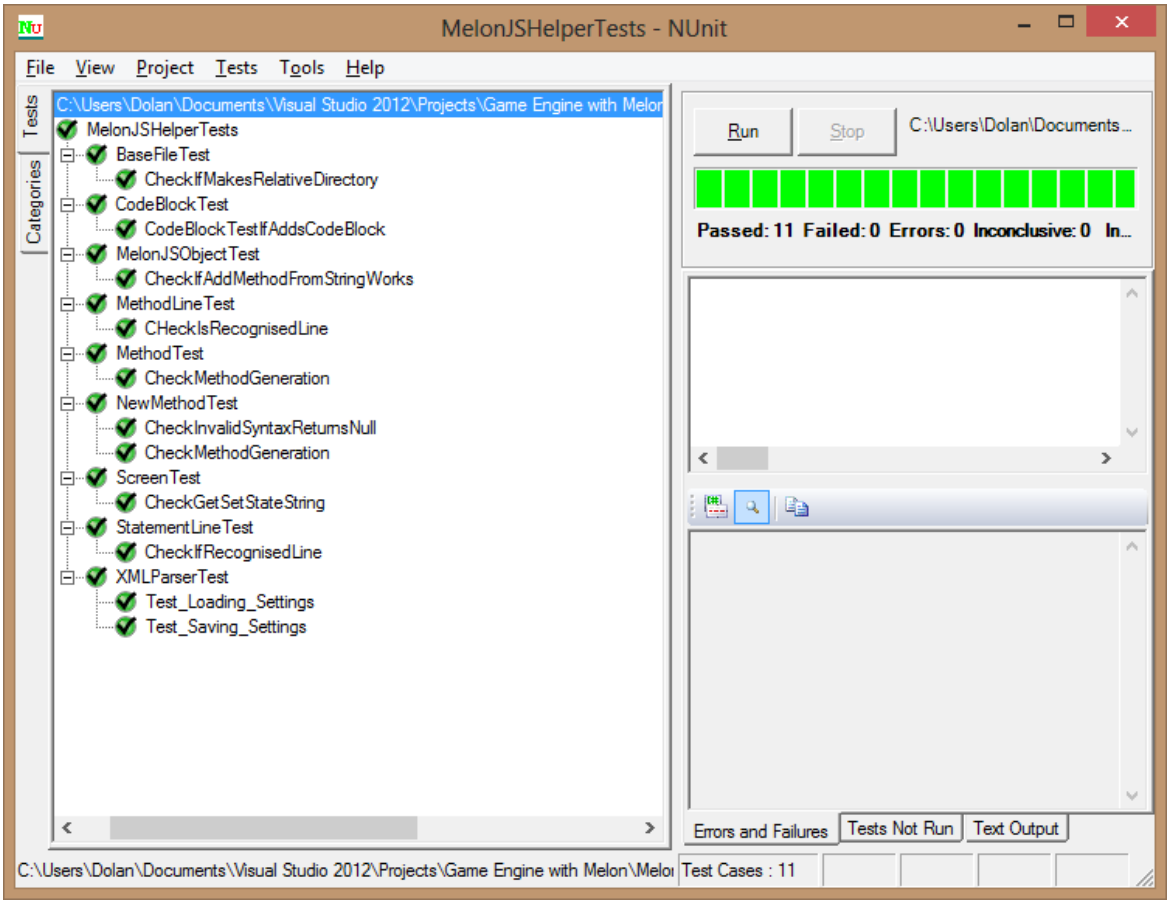


Figure 18 - NUnit Test and the results

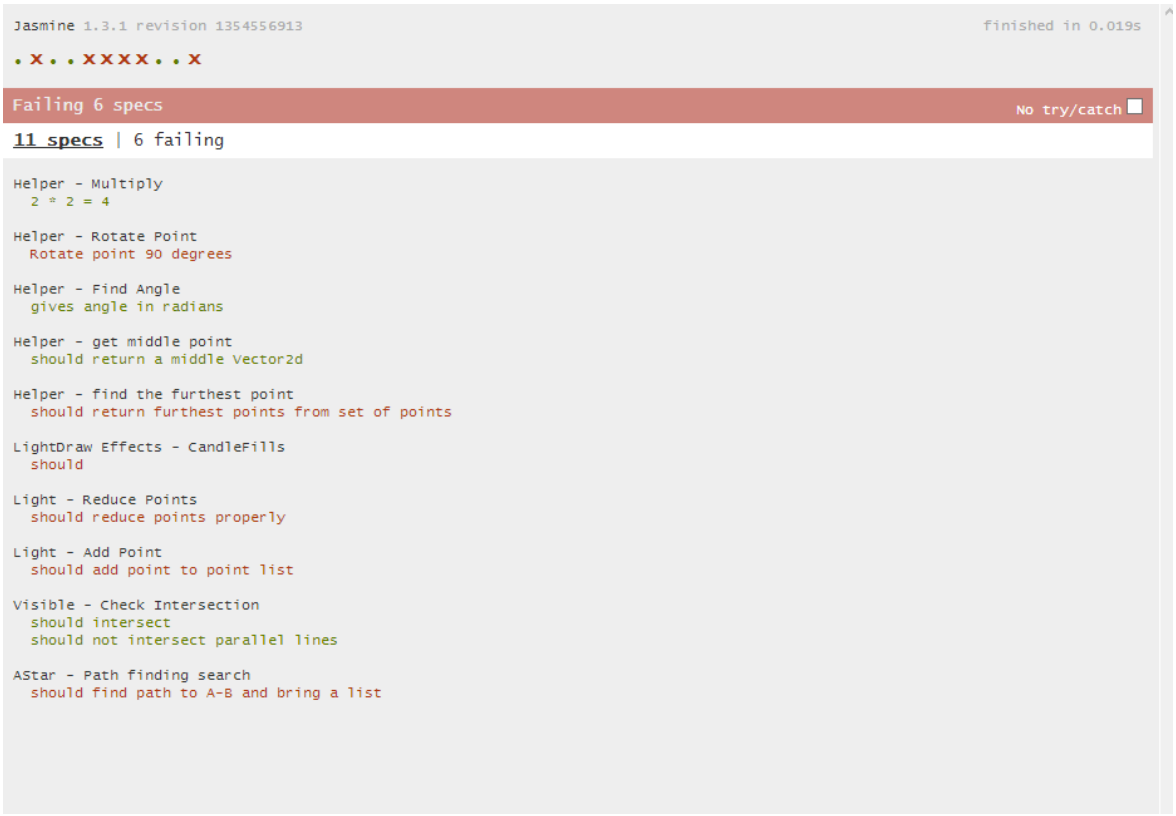


Figure 19 - Jasmine test results for JavaScript game framework

7.2 Integration testing

Integration testing happens after unit testing and before system testing in the V-Model. After making each critical section in MelonJS and MelonJS Helper, an integration test would be done to ensure that system worked together as it should have done. Various files would be made to see if MelonJS recognises it, and see if MelonJS could be recognised by the MelonJS Helper.

7.3 Acceptance tests

Acceptance tests were the last tests to be done. A few people were gathered which fit the description of the end user. They were invited to test the framework and the MelonJS Helper tool to see if they like it. Overall their response was positive, but however they were displeased with the layout of the helper of which a user mentioned he 'did not understand what the list boxes were for' in the resource tab. Changes were then made to correct these problems.

7.4 Tutorial and Documentation

Documentation for both the MelonJS Helper and the framework was written. SandCastle was used to generate the documentation in C#, whereas JSDoc was used to generate html JavaScript documentation. However due to the size, it was decided to be placed it on a CD.

A tutorial was developed to show how users would use this framework to create games. It guides through step by step of how to create each feature, and requires zero lines of code.

8 Conclusion and Evaluation

8.1 Summary

This chapter will outline the closing of the project with its highlights and its issues along with a critical evaluation of the outcome and possible future improvements. Over the past three months, new languages and software packages were explored to deliver a product which I believe is of a good releasable standard. Having no experience in JavaScript development was a challenge at first and took up a lot of time, but soon it was grasped and development was of good progress.

In the beginning of the project, aims and objectives were defined which will mould the project in its correct directions. The next sections will discuss if the aims and objectives of the project were met with a brief summary of the results.

8.2 Objectives

8.2.1 Creation of the platform

The main goal for the project is to create a platform for novices to create certain genres of games on. The game genres decided on are platformer, fighting, and a horror genre. Three demos of the genres using my platform will be created. To make it easier for the end user, minimal amount of coding is needed and pre-built templates for the corresponding genres will be made to be put on the ZipApp website.

One of the objectives set out at the beginning was to create a platform which will allow novices to easily create games on. A solution was developed to allow the creation of games easily without much need of programming and knowledge on games. This was done by abstracting the programming logic and MelonJS from the design of the game.

The MelonJS Helper tool aided with the routinely programming for the creation of levels, plugins, and the loading of resources. It also acts as a visual representation of the respective objects, allowing the level and plugin properties to be edited without delving into code – reducing chances of error.

With the addition of Tiled, all levels, maps and objects can be created with ease without touching a single line of code. Properties of objects are specified in Tiled to allow for customisation, such as its sprite, its speed, and many other factors.

8.2.2 ZipApp website integration

Unfortunately, because ZipApp is a closed source program, ZipApp integration could not be achieved at this time. However, a few mock images of how it would've looked like is done in order to give a visual representation of the intended outcome.

8.2.3 Plugin interface

A plugin interface is already written for MelonJS as of version 0.9.5. Instead, an intuitive tool was developed which will allow the end user to create and manage templates within the tool, automatically adding it to the MelonJS plugin pool. Once the plugin is made within the

tool, the user can continue working on it by opening the plugin file and write the body of the plugin.

8.2.4 Three templates

Creating the templates using the framework made was a success. All the games works as intended, and it stuck to the specification. The platformer game was noticeably the most polished game out of the three. The stealth game was the most technologically advancing, and the fighter is somewhere in between. All the games made require not a single line of code to get started with, and with the MelonJS Helper tool, making a game from the templates is an easy process.

8.3 Critical Evaluation

8.3.1 Overall Architecture

The overall architecture of this project is good; however, the bridge between MelonJS and the game seemed chaotic. Tiled, MelonJS Helper, Visual Studio, MelonJS just to deploy a game. Having said this, it will be easier than coding without aid. For the future, a single standalone application can be made instead of the many variety of tools needed perhaps extending on the MelonJS Helper tool idea further.

8.3.2 Future Development

YoYo's GameMaker (discussed in the Background section) has a game object creator which requires zero line of code. Multiple events can be added to objects, and within each events, actions can be added. These actions take the form of drag and droppable icons. Code can be constructed this graphical fashion and can be visualised by the user more easily. For example, an object could have a 'collide event', which when triggered will have a 'play sound' action, and a delete action. Inspired by game makers intuitive object creation tool, work has begun in making such tool which will take this graphical representation and translating it to code. However due to time limitations, this was never completed. This is an idea for future development which would be implemented for future releases.

As MelonJS is primarily a game framework for the web, the project could be ported so it works on web browsers as well, which will expand the audience, as not everyone owns a Windows 8 or Windows Phone.

8.4 Final Words

JavaScript has become a key player in the evolution of the internet. It was a good opportunity spent to have learnt the technologies. It was an enjoyable experience to make a product with huge potential. One of the aims of the project was to utilise JavaScript and HTML5 to create. Although some goals failed, such as integration with ZipApp, it was still a valuable learning experience.

9 References:

Adams, E. (2009). Fundamentals of Game Design, New Riders Publishing.

Appel, A. (1968). Some techniques for shading machine renderings of solids. Proceedings of the April 30--May 2, 1968, spring joint computer conference. Atlantic City, New Jersey, ACM**:** 37-45.

Beck (2002). Test Driven Development: By Example, Addison-Wesley Longman Publishing Co., Inc.

Bruner, N. (2013). "Parsing and Rendering Tiled TMX Format Maps in Your Own Game Engine." from http://gamedev.tutsplus.com/tutorials/implementation/parsing-tiled-tmx-format-maps-in-your-own-game-engine/.

Burbeck, S. (1992). "How to use Model-View-Controller (MVC)." Applications Programming in Smalltalk-80.

Burchard, E. (2013). The web game developers cookbook, Addison Wesley.

Coplien, J. O. (2003). Software design patterns. Encyclopedia of Computer Science, John Wiley and Sons Ltd.**:** 1604-1606.

Davis, A. M., E. H. Bersoff, et al. (1988). "A strategy for comparing alternative software development life cycle models." Software Engineering, IEEE Transactions on **14**(10): 1453-1461.

Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs." Numerische Mathematik **1**(1): 269-271.

Giertsen, C. (1992). "Volume visualization of sparse irregular meshes." Computer Graphics and Applications, IEEE **12**(2): 40-48.

Helen Sharp, T. R., Jenny Preece (2002). Interaction Design, John Wiley & Sons, Inc.

Johnson, D. B. (1973). "A Note on Dijkstra's Shortest Path Algorithm." J. ACM **20**(3): 385-388.

Krasner, G. E. and S. T. Pope (1988). "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80." J. Object Oriented Program. **1**(3): 26-49.

Kuryanovich, E., S. Shalom, et al. (2012). Particle Systems. HTML5 Games Most Wanted, Apress**:** 107-132.

Kushner, D. (2002). "Software: the wizardry of Id." IEEE Spectr. **39**(8): 42-47.

Lester, P. (2004). Heuristics and A* Pathfinding. A* Pathfinding for Beginners.

Lindley, C. (2010). jQuery Cookbook, O'Reilly Media.

Lowdermilk, T. (2013). A Developer's Guide to Building User-Friendly Applications, O'Reilly Media.

McFarland, D. S. (2011). JavaScript & jQuery: The Missing Manual, O'Reilly Media.

MelonJS (2013). "melonJS - A lightweight HTML5 game engine." from http://www.melonjs.org/.

Muchmore, M. (2012). "13 New Features in Windows 8 Consumer Preview." from http://www.pcmag.com/article2/0,2817,2400967,00.asp.

Norman, D. A. (1998). The design of everyday things. London, MIT.

> This bible on the cognitive aspects of design contains examples of both good and bad design, and simple rules that designers can use to improve the usability of objects as diverse as cars, computers, doors, and telephones.

Olan, M. (2003). "Unit testing: test early, test often." J. Comput. Small Coll. **19**(2): 319-328.

Osmani, A. (2012). Learning JavaScript Design Patterns.

Patel, A. (2012). 2D Visiblity Algorithm.

Planet, V. R. (2010). tileb.png. http://vxresource.files.wordpress.com/2010/03/tileb.png. **512 x 512**.

Pree, W. and H. Sikora (1997). Design patterns for object-oriented software development (tutorial). Proceedings of the 19th international conference on Software engineering. Boston, Massachusetts, USA, ACM**:** 663-664.

Rainsberger, J. B. (2001). Use your singletons wisely - Know when to use singletons, and when to leave them behind.

Renaudeau, G. (2012). "Illuminated.js – 2D lights and shadows rendering engine for HTML5 applications." from http://greweb.me/2012/05/illuminated-js-2d-lights-and-shadows-rendering-engine-for-html5-applications/.

Rettig, P. (2012). Professional HTML5 Mobile Game Development, Wrox.

Rogers, R. (2012). A Hands-On Guide to Building Your First Android Game, Addison-Wesley.

Rogers, R. (2012). Learning Android Game Programming, Addison Wesley.

Satalkar, B. (2011). Waterfall Model Vs. V Model.

Schmuller, J. (2004). Teach yourself UML in 24 hours, Sams Publishing.

Shneiderman, B. (2005). Designing the user interface, Addison Wesley.

Silva, C. T. and J. S. B. Mitchell (1997). "The lazy sweep ray casting algorithm for rendering irregular grids." Visualization and Computer Graphics, IEEE Transactions on **3**(2): 142-157.

Skinner, G. (2013). "Getting started with the Flash Professional Toolkit for CreateJS."

Smith, J. (2009). WPF Apps With The Model-View-ViewModel Design Pattern. MSDN Magazine.

Smith, J. M. (2012). Elemental Design Patterns, Addison Wesley.

Stefanov, S. (2008). Object-Oriented JavaScript. Birmingham,, Packt Publishing.

Stefanov, S. (2010). JavaScript Patterns, O'Reilly Media, Inc.

Teschner, M., S. Kimmerle, et al. (2005). "Collision Detection for Deformable Objects." Computer Graphics Forum **24**(1): 61-81.

Upton, D. (2007). CodeIgniter for Rapid PHP Application Development. Birmingham, PACKT Publishing.

Vishnyakov, A. (2013). "Elysium." from http://elysium.asvishnyakov.com/Downloads.cshtml.

w3schools (2013). "Mobile Devices Statistics." from http://www.w3schools.com/browsers/browsers_mobile.asp.

Wimmer, M., M. Giegl, et al. (1999). "Fast walkthroughs with image caches and ray casting." Computers & Graphics **23**(6): 831-838.

Xie, J. (2012). Research on Key Technologies Base Unity3D Game Engine. The 7th International Conference. Melbourne, Australia**:** 695-699.

Xie, T., K. Taneja, et al. (2007). Towards a Framework for Differential Unit Testing of Object-Oriented Programs. Proceedings of the Second International Workshop on Automation of Software Test, IEEE Computer Society**:** 5.

Yang, B., X. Cheng, et al. (2005). A real-time collision detection algorithm for mobile billiards game. Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology. Valencia, Spain, ACM**:** 294-297.

Yngvi Bjornsson, K. H. (2006). "Improved Heuristics for Optimal Pathfinding on Game Maps."

Yoon, S.-e., Y. J. Kim, et al. (2010). Recent advances in real-time collision and proximity computations for games and simulations. ACM SIGGRAPH ASIA 2010 Courses. Seoul, Republic of Korea, ACM**:** 1-110.

Zhan, F. B. (1997). "Three Fastest Shortest Path Algorithms on Real Road Networks: Data Structures and Procedures." Journal of Geographic Information and Decision Analysis **1**: 70-82.

Zimmerman, K. S. a. E. (2004). Rules of Play: Game Design Fundamentals, The MIT Press.

10 Appendix

10.1 Supporting documentation and diagrams

10.1.1 User Requirements List

10.1.1.1 Must have requirements

Number	**Unique ID**
Title	Name of the unique requirement in natural language
Type	Functional / Non-functional - Product / Organizational
Description	Top level description of the requirement

Number	**1**
Title	Menu components
Type	Functional - Product
Description	Buttons, labels and switches etc. must be implemented in the game so that the end user can create their own personalised menu system for their game.

Number	**2**
Title	Animated and static sprite support
Type	Functional - Product
Description	Sprites can be animated or non-animated. The user will be able to use both of them to create the intended effect for the game. For example, static for walls, and animated sprites for enemies.

Number	**3**
Title	Playback of sounds and music
Type	Functional - Product
Description	Sounds should be able to be played upon request for background music, or sound from objects. E.g. footsteps or gun shots. The sounds should be able to be manipulated in basic spatial ways

Number	**4**
Title	A method to store level data
Type	Functional - Product
Description	Levels must be stored in a format which can reproduce the level once it is loaded up after being stored, it must be able to store any kind of level configuration.

Number	**5**
Title	Parallax and non-parallaxed planes
Type	Functional - Product
Description	Backgrounds must be able to move slower than the level, and the foreground elements must move faster than the level. This will give the illusion of depth. The user can specify how much parallax there is on each plane, including zero parallax.

Number	**6**
Title	Saving and loading
Type	Functional - Product
Description	The game must be saveable and loadable to continue the user's progress through the game. The saved file must be able to represent that.

Number	**7**
Title	Keyboard and mouse input
Type	Functional - Product
Description	When playing with a PC, it must be able to accept commands from a keyboard and/or mouse.

Number	**8**
Title	Touch input
Type	Functional - Product
Description	When deploying on the Windows store for Windows 9 tablets and Windows Phone, it must support input from the touch screen. This is also a requirement from Microsoft which if it is not met, the App will not be deployable to the store.

Number	**9**
Title	Work on Windows 8 and Windows Phone
Type	Functional - Organisational
Description	The templates must be tested on both Windows 8 and Windows Phone to ensure they work on these platforms.

Number	**10**
Title	Scalable resolution
Type	Functional - Product
Description	The resolution of the game should be able to resize without distorting the game graphics. Many Windows devices have different resolutions, the template must be compatible with all device resolutions.

10.1.1.2 Should have requirements

Number	**11**
Title	Keyboard and mouse input
Type	Functional - Product
Description	When playing with a PC, it must be able to accept commands from a keyboard and/or mouse.

Number	**12**
Title	Particle effect engine
Type	Functional - Product
Description	Particle effects enhance the realism of a game, to immerse the player further.

Number	**13**
Title	Dynamic water
Type	Functional - Product
Description	Water which reacts and ripples to solid bodies enhances the realism of a game to a great degree.

Number	**14**
Title	Surround sound
Type	Functional - Product
Description	To further immerse the player into the game.

Number	**15**
Title	Lighting effects
Type	Functional - Product
Description	To further immerse the player into the game. For example, shadows and highlights

Number	**16**
Title	Physics
Type	Functional - Product
Description	Basic physics which will add mechanics to the game. For example, gravity, newton's third flaw, collision etc.

Number	**17**
Title	Simple AI for in-game characters
Type	Functional - Product
Description	Certain types of games require other characters in game which the player can interact with. For example, enemies or helpful assistants

Number	**18**
Title	Camera/Map Scrolling
Type	Functional - Product
Description	To track the player avatar when it moves across the level

Number	**19**
Title	Reward System
Type	Functional - Product
Description	By adding a points or score system, the player will feel a sense of achievement when collecting coins or killing enemies.

Number	**20**
Title	Tutorial on how to use the platform
Type	Functional - Product
Description	Documentation or video instructions on how to create a game using the platform.

10.1.1.3 Could have requirements

Number	**21**
Title	Ability to set preferences of the game template
Type	Functional - Product
Description	On the ZipApp website prior to download, the user could configure the game to his/her liking. For example, the game title, amount of buttons, type of game, assets etc.

10.1.1.4 Want to have requirements

Number	**21**
Title	Stock sprites and sounds
Type	Functional - Product
Description	Custom made sprites and sound assets could be added to the game at the end to aid the end user to create games.

10.1.2 Use Case Tables:

Use Case: Creating and downloading a template	
ID	1
Brief	Actor is able to select appropriate template
Primary Actors	User
Secondary Actors	None
Preconditions	Must have signed up with ZipApp
Main Flow	1. Actor Flow starts with user creating a new App on the ZipApp website 2. System displays a form with app specific details and template type. 3. Actor selects a general template to work with "Game" 4. Actor fills in app specific details 5. System then produces a list of game genres the actor can create 6. Actor then selects a genre he wants 7. Actor selects plugins he want included in the game 8. System creates the template and produces a download link 9. Actor downloads the template files
Exception Flow	System validation fails due to missing app specific details System requests user to type in missing details Go back to Main Flow #4
Exception Flow	System validation fails due to actor didn't select game genre System requests user to select genre Go back to Main Flow #3
Post conditions	Actor is able to un-zip template file and opens the template .SLN in visual studio.
Trigger	If actor is logged in, and wants to create an app

Use Case: Changing genre of game	
ID	2
Brief	Actor able to change genre of game
Primary Actors	User
Secondary Actors	None
Preconditions	Must have created an app already
Main Flow	1. Actor logs in to ZipApp 2. System produces list of created templates 3. Actor selects previously created template 4. System displays the template properties, including a list of game genres and currently selected genre 5. Actor then selects a genre he wants 6. System updates the template details
Exception Flow	Actor does not change genre of game Actor saves properties of template System alerts nothing has been changed and goes to Main Flow #6
Post conditions	Actor is transported back to the properties screen for the current template.
Trigger	If actor has logged in and is editing a template.

Use Case: Downloading the template file	
ID	3
Brief	Actor able to download game template file
Primary Actors	User
Secondary Actors	None
Preconditions	Must have created an app already
Main Flow	1. Actor logs in to ZipApp 2. System produces list of created templates 3. Actor selects previously created template 4. System displays the template properties, including a list of game genres and currently selected genre 5. Actor then selects the download button 6. System brings a download link for the template 7. Actor confirms the download 8. System begins downloading the file onto local computer.
Exception Flow	Actor does not download template Actor saves properties of template System alerts nothing has been changed and goes to Main Flow #6
Post conditions	Actor is able to un-zip template file and opens the template .SLN in visual studio.
Trigger	If actor has logged in and is editing a template.

Use Case: Creating a level using the WYSIWYG editor	
ID	4
Brief	Actor able to use WYSIWYG editor to create the game level
Primary Actors	User
Secondary Actors	None
Preconditions	Must have downloaded the template already
Main Flow	1. Actor un-zips the template files 2. System displays a list of project files 3. Actor selects the level editor exe within the project files 4. System opens up the editor
Exception Flow	Actor does not have a compression program installed System will prompt the actor to download one online Actor downloads and installs the compression programs Actor returns to Main Flow #1
Post conditions	Actor is able to explore and use the editor to his liking.
Trigger	If actor has downloaded and un-zipped the template

Use Case: Creating objects using the editor	
ID	5
Brief	Actor able to use the objects made in the IDE in the editor
Primary Actors	User
Secondary Actors	None
Preconditions	Must have downloaded the template already
Main Flow	1. Actor creates an object within the editor 2. System displays object on the stage 3. Actor references the object with the same name referenced in Visual Studio 4. Actor fills in properties of the object 5. System updates the object 6. Actor saves the map
Exception Flow	Actor does not reference the object with the correct name (through typo or referencing a non-existent object) Actor goes to Main Flow #3 to attempt to correct.
Post conditions	Actor is able to see the object within the game.
Trigger	If actor is already using the map editor, and click the create object button.

Use Case: Create custom game objects	
ID	6
Brief	Actor is able to create custom game objects
Primary Actors	User
Secondary Actors	None
Preconditions	Must have downloaded the template already
Main Flow	1. Actor extends a previous base object 2. Actor writes appropriate methods 3. Actor saves code 4. Actor opens the resource.js generator 5. System opens up the resource.js generator 6. Actor fills in asset and object details for the app 7. Actor clicks the generate button 8. System generates a resource.js file 9. Actor creates the object by following use case 5.
Exception Flow	Actor writes a syntax error upon extending the object. System will throw an error, displaying the line in which it occurred. Return to Main Flow #2
Post conditions	Actor is able to continue creating the level
Trigger	If actor is currently using the IDE

Use Case: Create plugin for template	
ID	7
Brief	Actor is able to create custom plugins using the helper tool
Primary Actors	User
Secondary Actors	None
Preconditions	Must have opened the helper tool, and have set the plugin directory in the tool
Main Flow	1. Actor creates new plugin 2. Actor specifies plugin name 3. Actor specifies version of plugin 4. Actor presses button to create plugin 5. System creates a plugin.js file template for use.
Exception Flow	Actor does not fill in appropriate details System will throw an error, displaying the line in which it occurred. Return to Main Flow #1
Post conditions	Actor is able to continue to use the tool
Trigger	If actor is currently using the tool

Use Case: Create level for template using tool	
ID	8
Brief	Actor is able to make levels in the tool
Primary Actors	User
Secondary Actors	None
Preconditions	Must have opened the helper tool, and have set the plugin directory in the tool
Main Flow	1. Actor creates new level 2. Actor specifies level name 3. Actor specifies the state of the level 4. Actor clicks button to generate level 5. System produces screens.js file
Exception Flow	Actor does not fill in appropriate details System will throw an error, displaying the line in which it occurred. Return to Main Flow #1
Post conditions	Actor is able to continue to use the tool
Trigger	If actor is currently using the tool

10.1.3 Class Diagrams

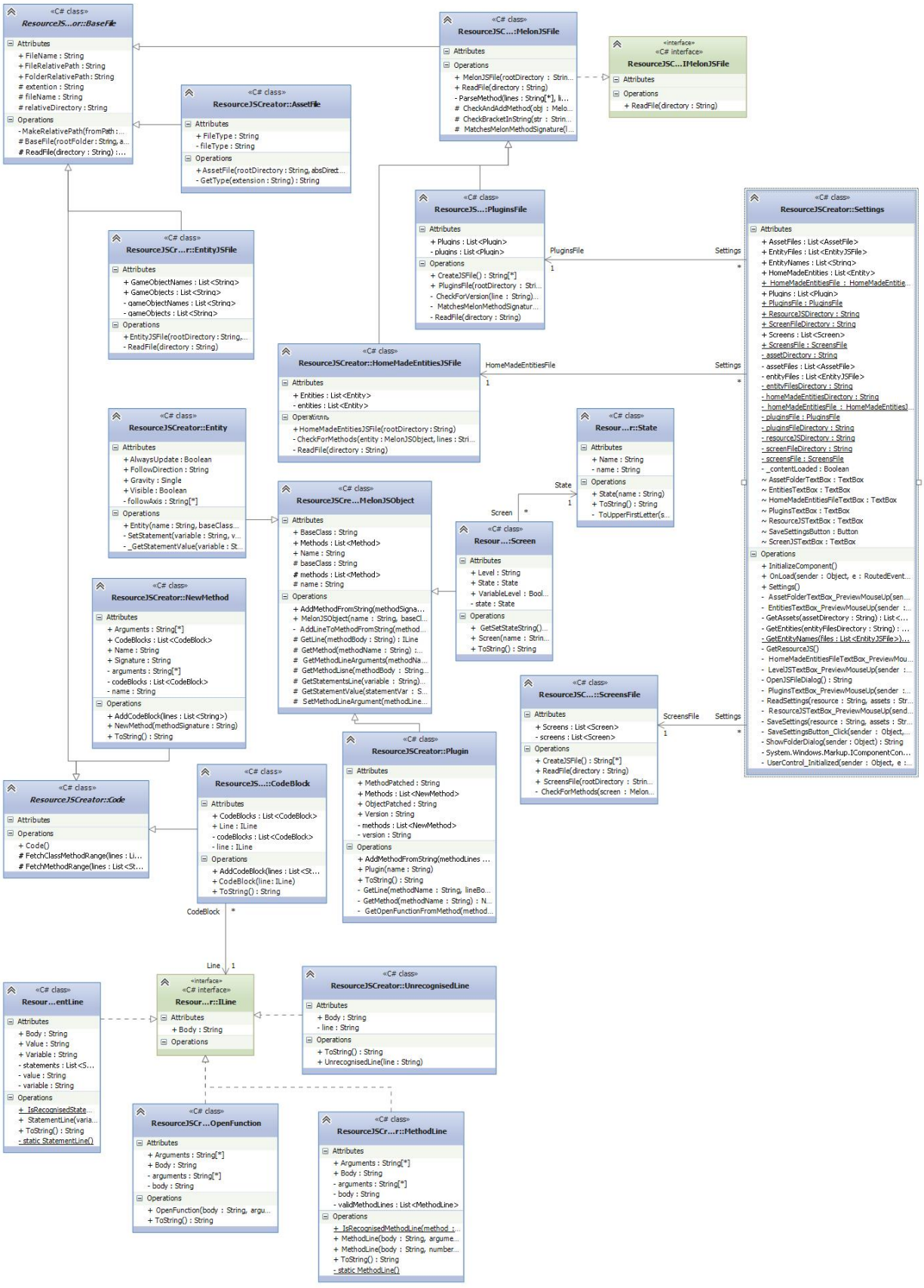
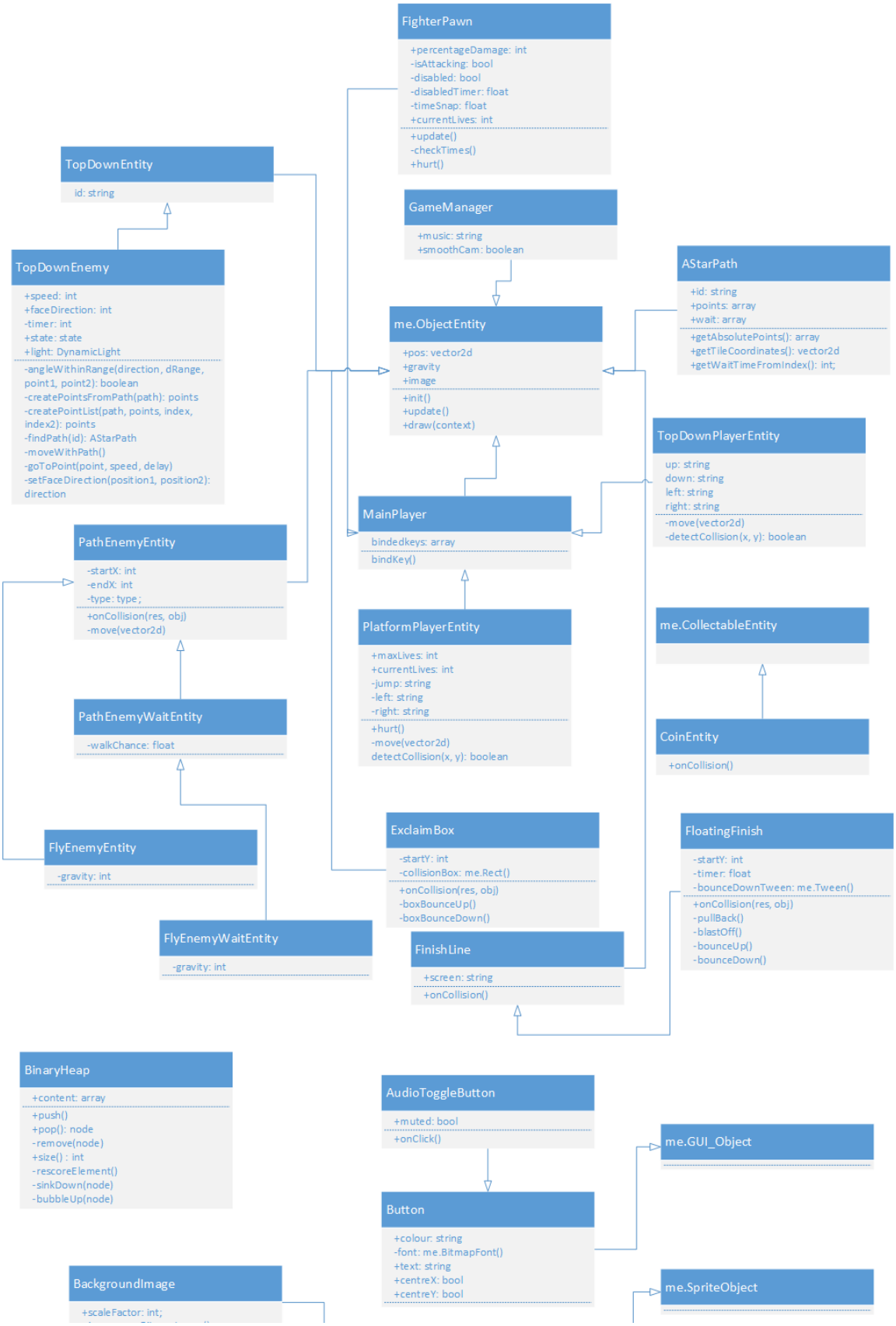


Figure 20 – UML Class diagram for the MelonJS Helper tool


FighterPawn
+percentageDamage: int
-isAttacking: bool
-disabled: bool
-disabledTimer: float
-timeSnap: float
+currentLives: int
+update()
-checkTimes()
+hurt()
TopDownEntity
id: string
GameManager
+music: string
+smoothCam: boolean
AStarPath
+id: string
+points: array
+wait: array
+getAbsolutePoints(): array
+getTileCoordinates(): vector2d
+getWaitTimeFromIndex(): int;
TopDownEnemy
+speed: int
+faceDirection: int
-timer: int
+state: state
+light: DynamicLight
-angleWithinRange(direction, dRange, point1, point2): boolean
-createPointsFromPath(path): points
-createPointList(path, points, index, index2): points
-findPath(id): AStarPath
-moveWithPath()
-goToPoint(point, speed, delay)
-setFaceDirection(position1, position2): direction
me.ObjectEntity
+pos: vector2d
+gravity
+image
+init()
+update()
+draw(context)
TopDownPlayerEntity
up: string
down: string
left: string
right: string
-move(vector2d)
-detectCollision(x, y): boolean
MainPlayer
bindedkeys: array
bindKey()
PathEnemyEntity
-startX: int
-endX: int
-type: type;
+onCollision(res, obj)
-move(vector2d)
PlatformPlayerEntity
+maxLives: int
+currentLives: int
-jump: string
-left: string
-right: string
+hurt()
-move(vector2d)
detectCollision(x, y): boolean
me.CollectableEntity
PathEnemyWaitEntity
-walkChance: float
CoinEntity
+onCollision()
ExclaimBox
-startY: int
-collisionBox: me.Rect()
+onCollision(res, obj)
-boxBounceUp()
-boxBounceDown()
FloatingFinish
-startY: int
-timer: float
-bounceDownTween: me.Tween()
+onCollision(res, obj)
-pullBack()
-blastOff()
-bounceUp()
-bounceDown()
FlyEnemyEntity
-gravity: int
FlyEnemyWaitEntity
-gravity: int
FinishLine
+screen: string
+onCollision()
BinaryHeap
+content: array
+push()
+pop(): node
-remove(node)
+size() : int
-rescoreElement()
-sinkDown(node)
-bubbleUp(node)
AudioToggleButton
+muted: bool
+onClick()
me.GUI_Object
Button
+colour: string
-font: me.BitmapFont()
+text: string
+centreX: bool
+centreY: bool
BackgroundImage
+scaleFactor: int;
me.SpriteObject

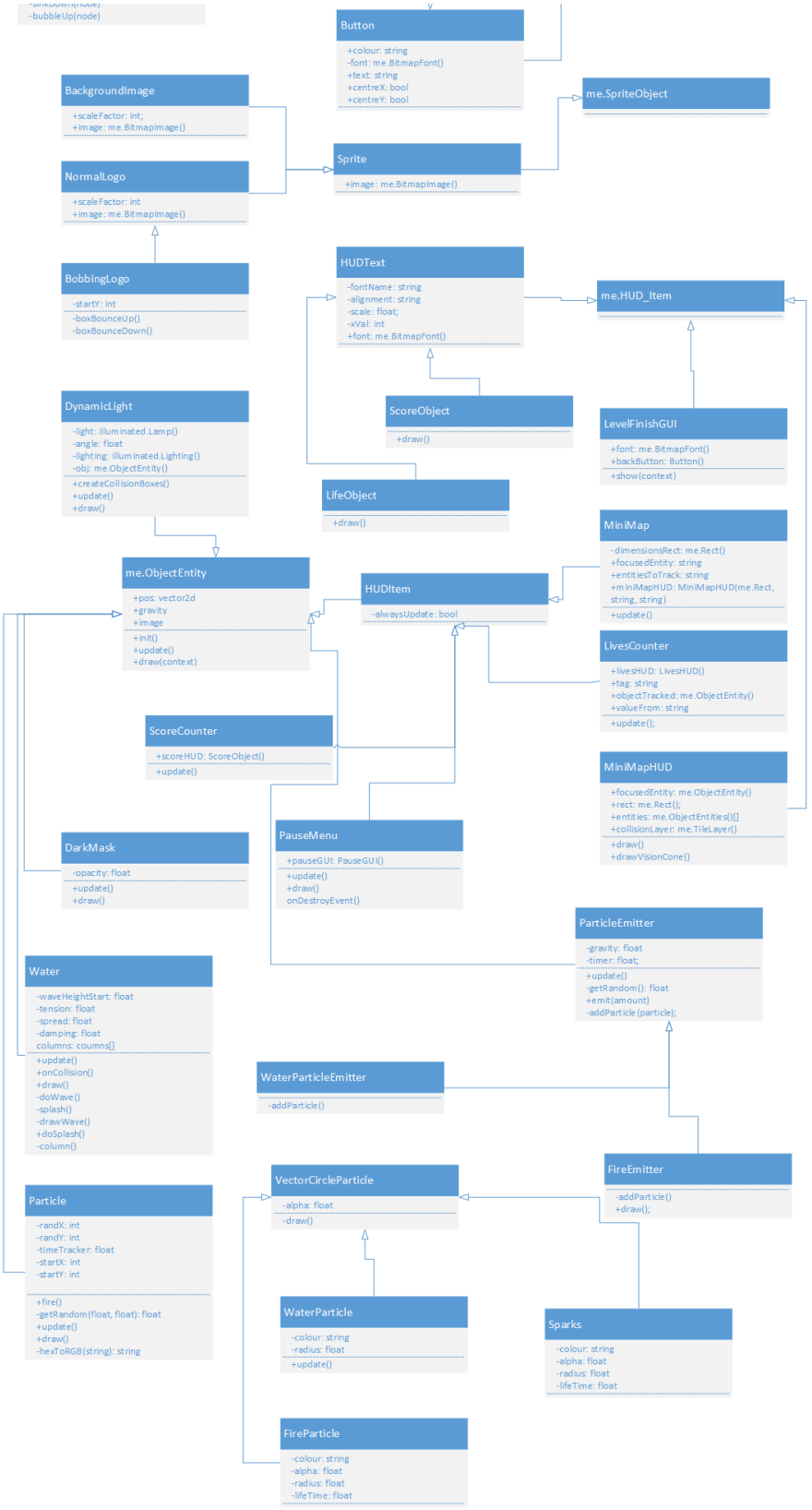


Figure 21 – UML Class diagram for the JavaScript game library framework

10.1.4 Sequence Diagrams

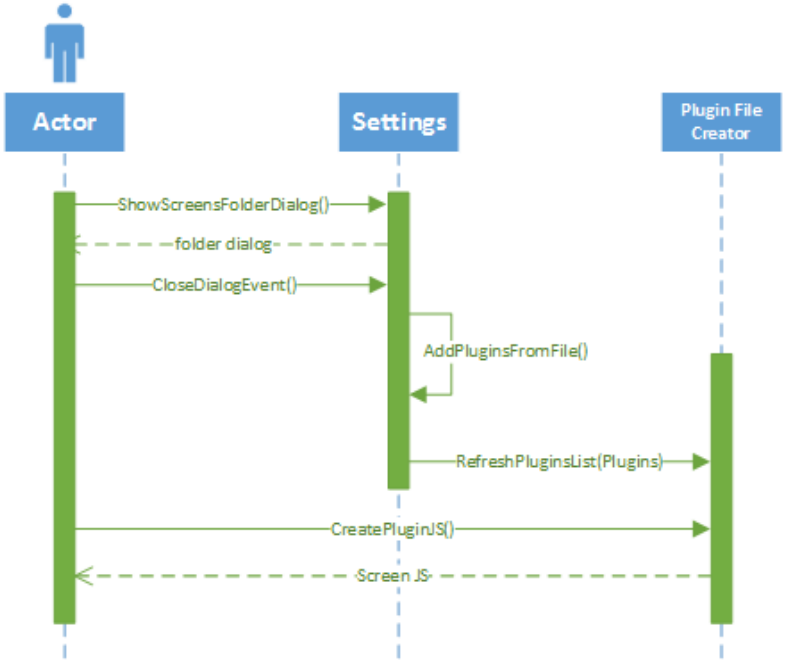


Figure 22 – Sequence diagram showing plugin.js creation

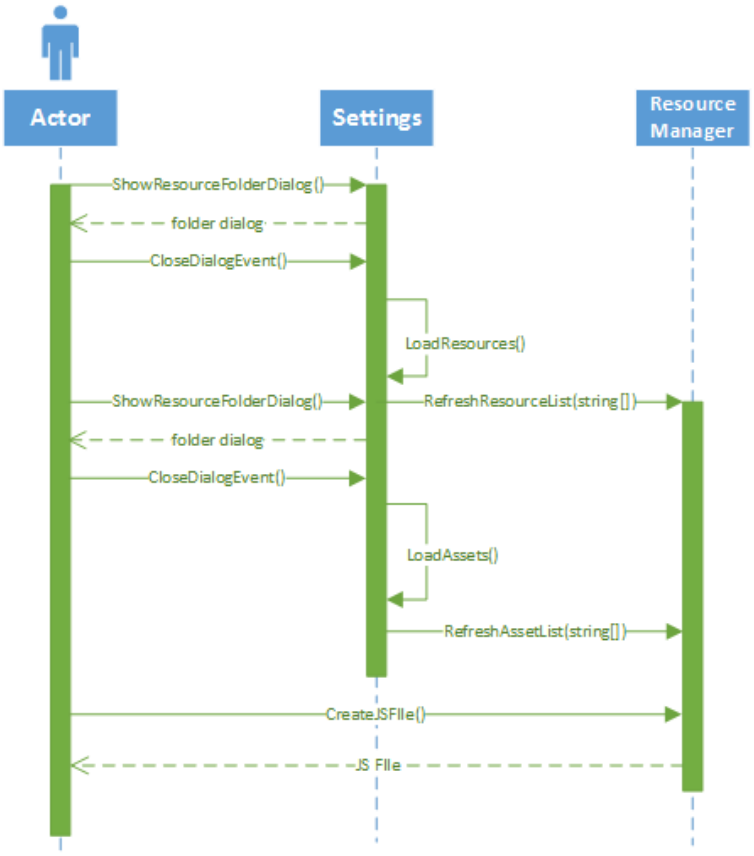


Figure 23 – Sequence diagram showing resource.js creation

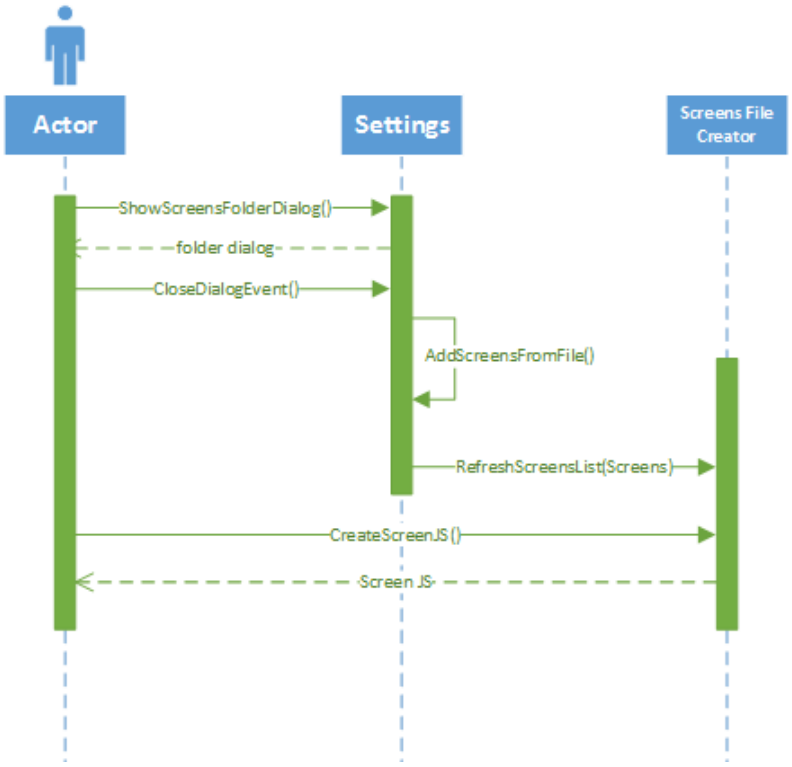


Figure 24 – Sequence diagram showing screen creation

10.1.5 Use case diagrams

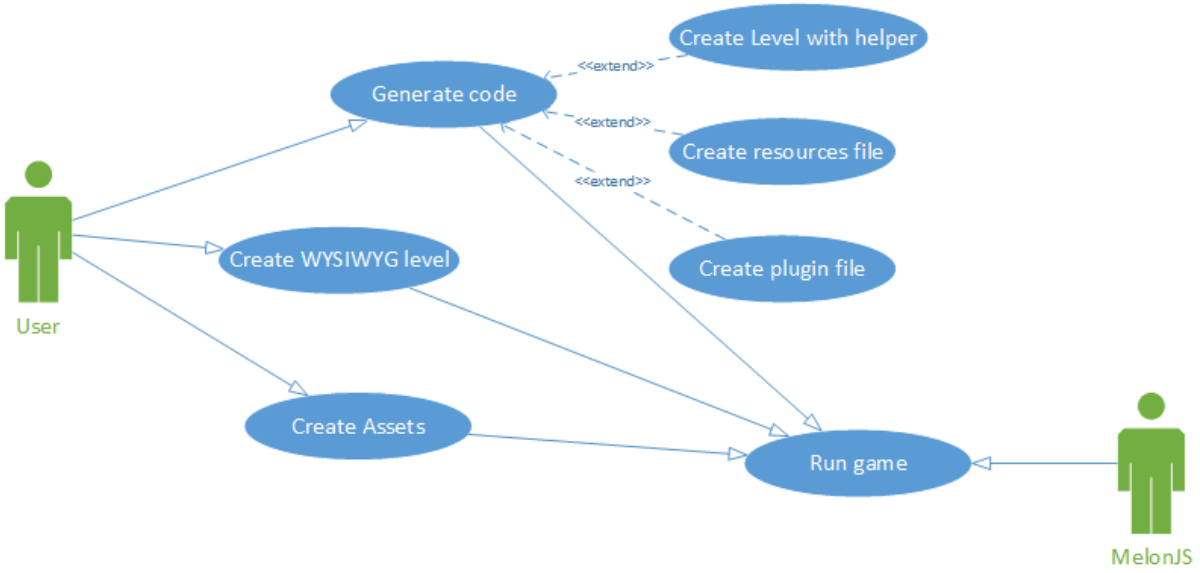


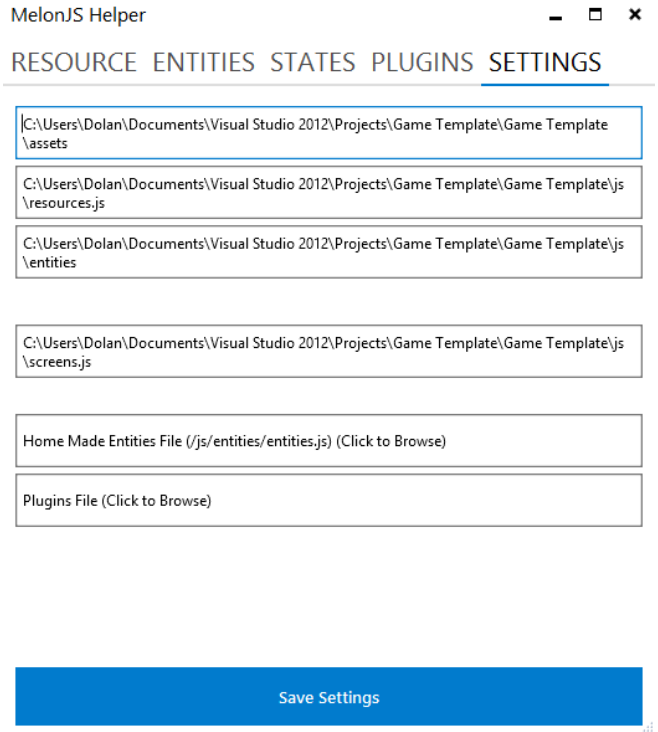
Figure 25 - Use case diagram for the User and MelonJS

10.2 Tutorial

Following this guide will aid you to making a simple platformer.

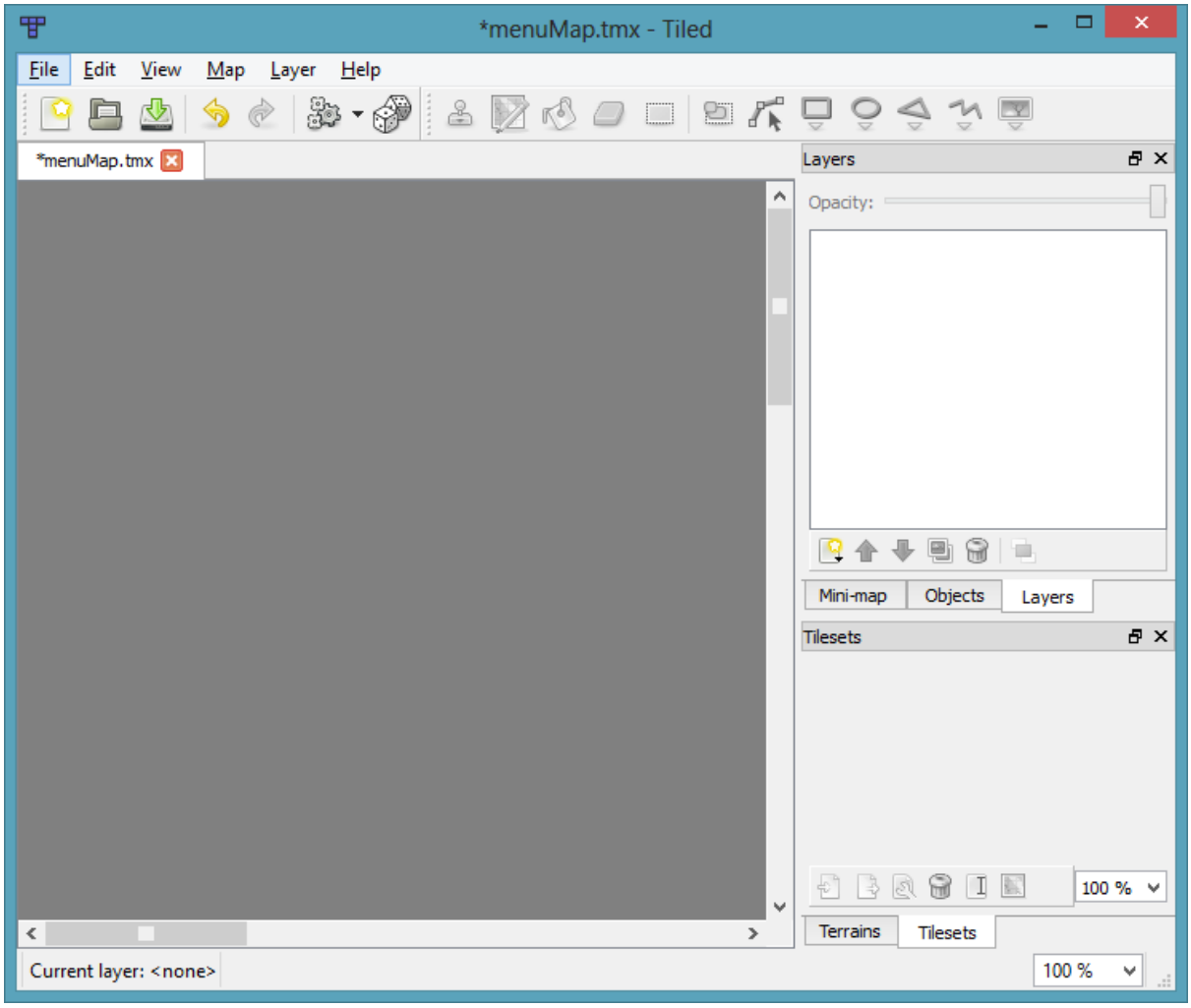
1. Make sure Visual Studio 2012 is installed and download the ZipApp template for the game. Unzip it in a desired location. Install the Tiled program located at the root directory.
2. First, we will set-up the MelonJS Helper tool for use in the future. Open the helper tool located in /MelonJS Helper/MelonJSHelper.exe, and click on the settings tab. Click on each of the textboxes to give the tool a reference to your project files. The

main ones is the assets folder, entities folder, the resource.JS file and the screens.JS. All of these files are within the template folder.



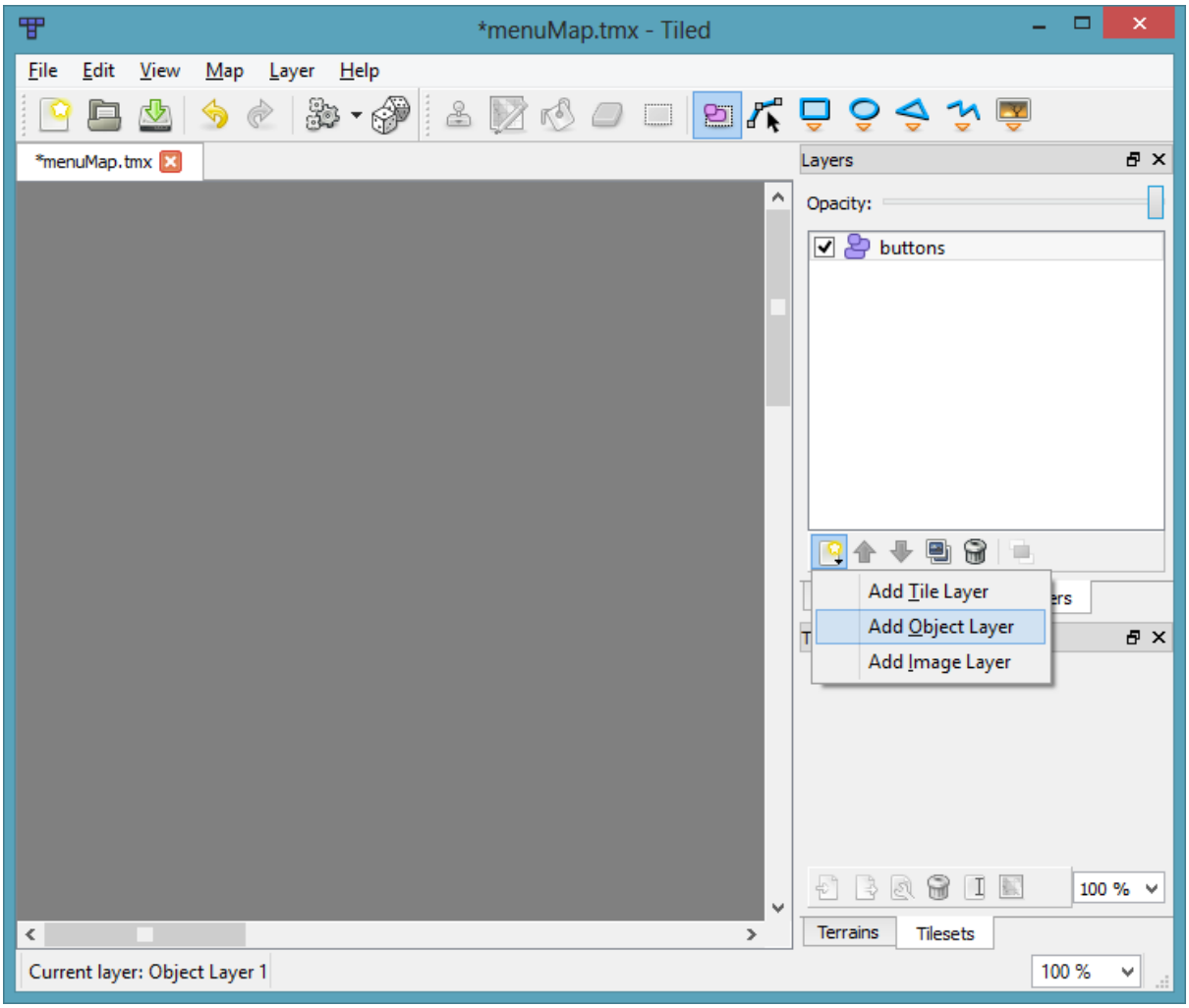
Afterwards, save the settings. It can now be closed, and the settings will be preserved.

3. Navigate to the /js/screens.js. Open it with Tiled. This is the default screen displayed launching the program, and will also be the Title Screen.

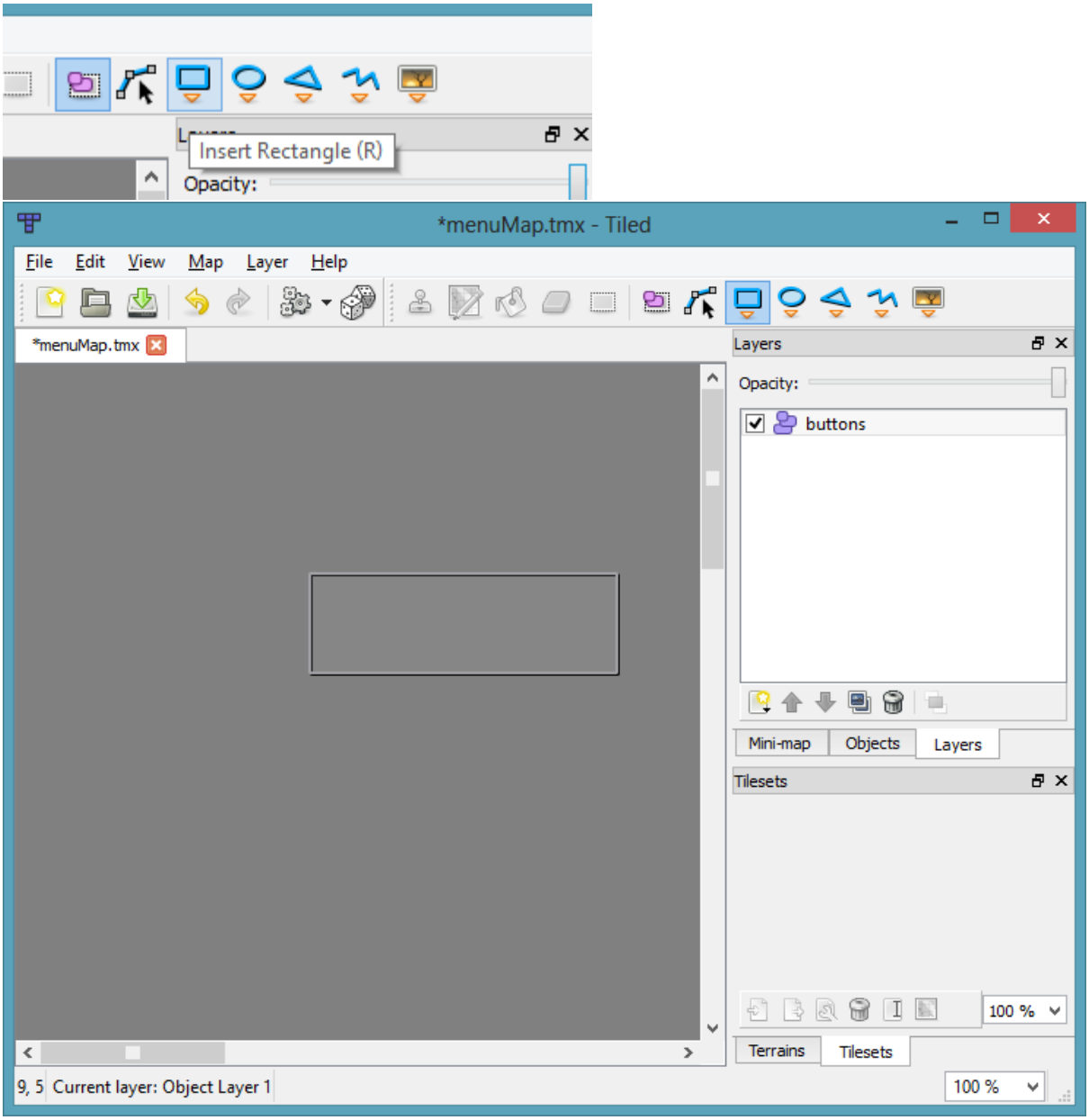


4. To create a Play button, we need to create a MelonJS object. To do this, click on the new layer button on the layers, and click ‘New Object Layer’. We will call it

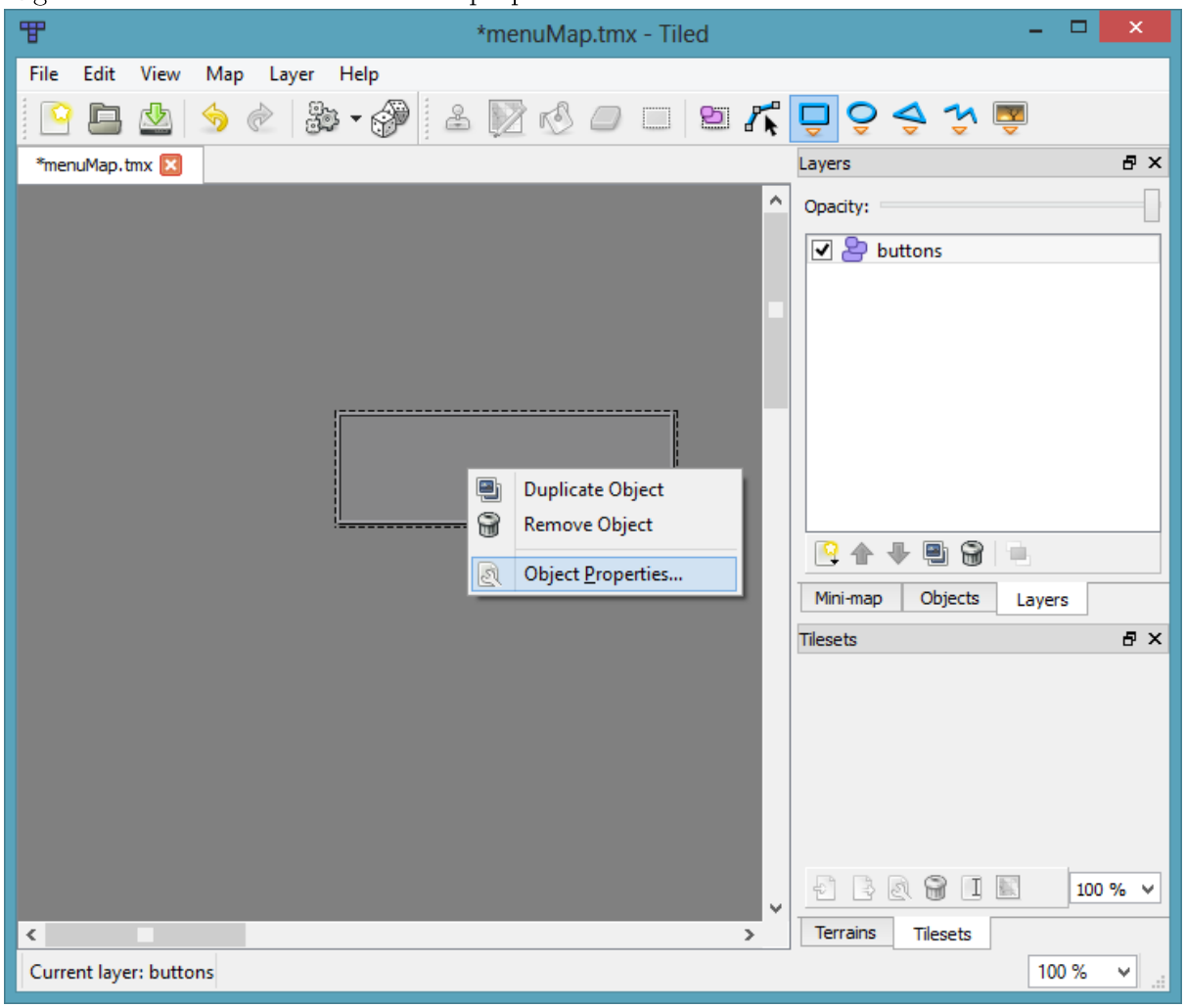
'buttons'.



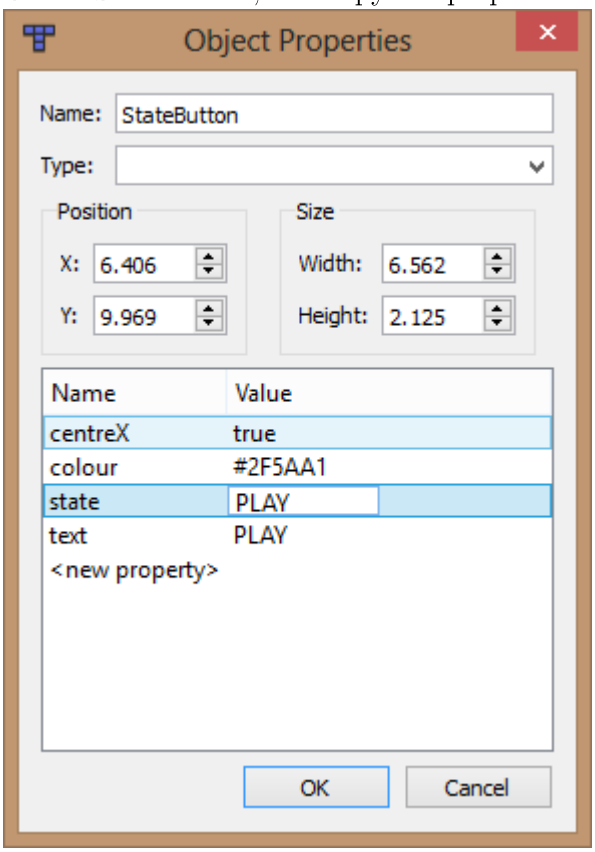
5. On the top bar, click on the rectangle button, and draw it somewhere on screen in the buttons layer.


Insert Rectangle (R)
Opacity:
*menuMap.tmx - Tiled
File
Edit
View
Map
Layer
Help
*menuMap.tmx
Layers
Opacity:
buttons
Mini-map
Objects
Layers
Tilesets
100 %
Terrains
Tilesets
9, 5
Current layer: Object Layer 1
100 %

6. Right click on the box and select its properties

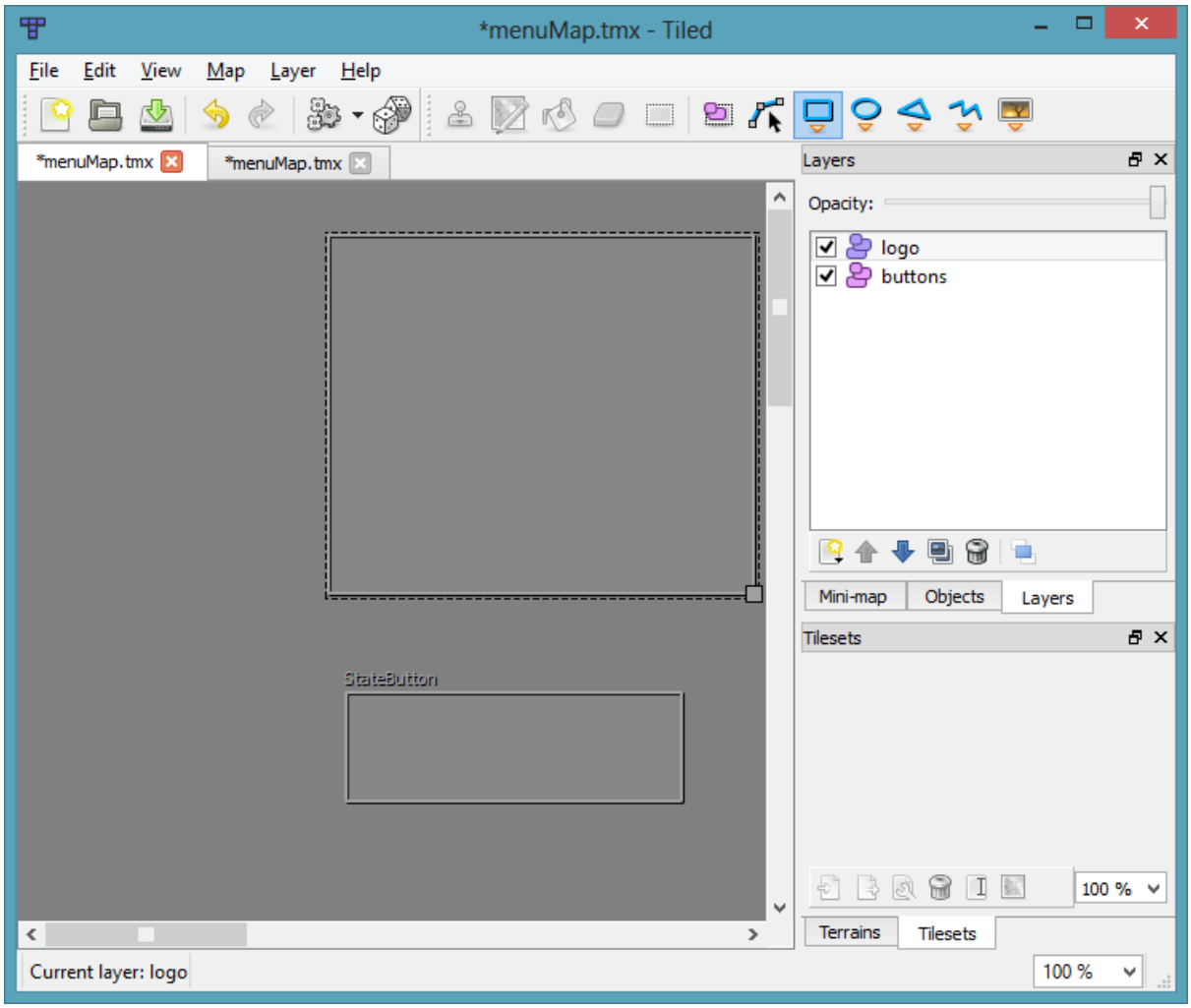


7. Call it StateButton, and copy the properties of the button shown below:

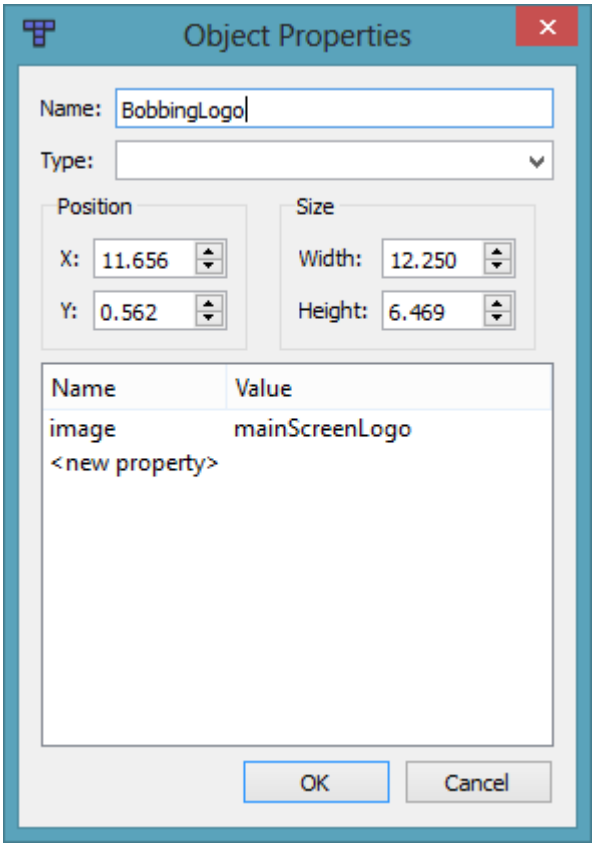


This will create a blue coloured 'StateButton' object which is centred horizontally on the screen which goes go the LEVELSELECT state which has the play text written on it.

8. To add a logo onto the screen, create another object layer like in step 3 – call it ‘logo’ and draw a box.

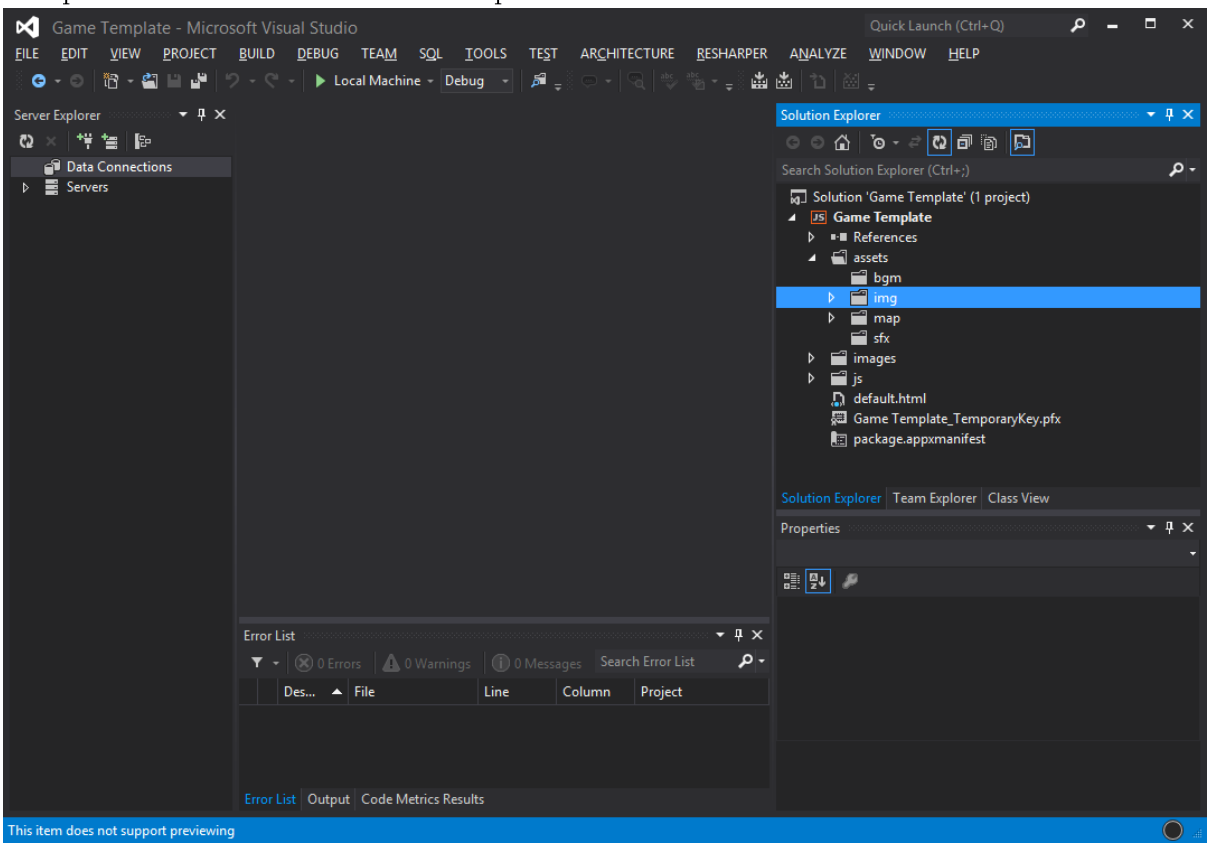


9. Access the new boxes properties again by right-clicking, then clicking ‘Object Properties’. Call it BobbingLogo and give it a property ‘image’ with value ‘mainScreenLogo’

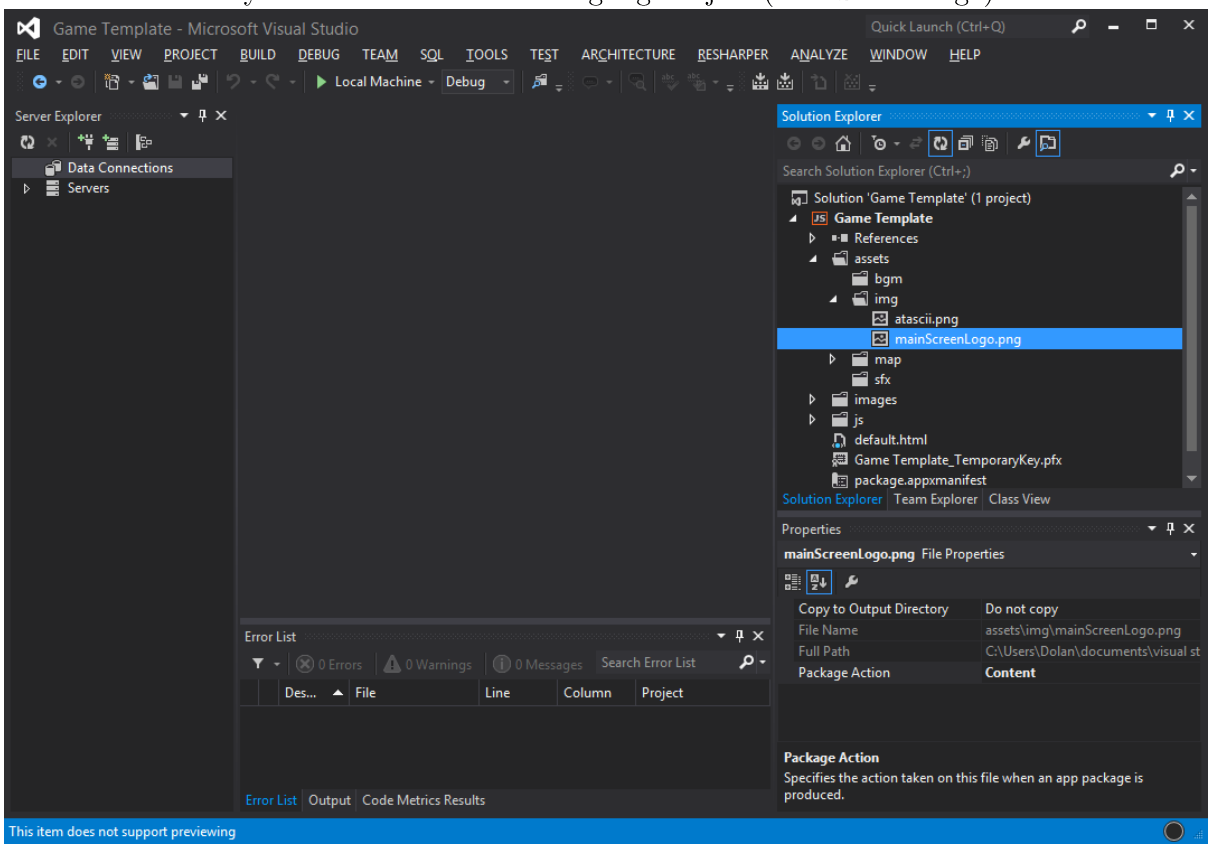


We have not made 'mainScreenLogo', so at this point, the game would not run. We need to include this image into the project files.

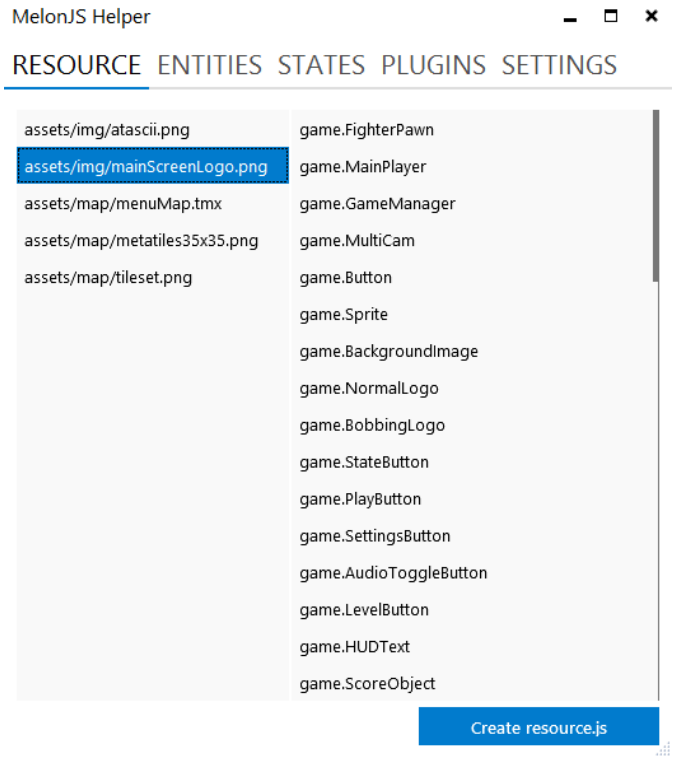
10. Open the Visual Studio Project (.JSPROJ) file in the root folder you downloaded the Template in. Visual Studio will now open:



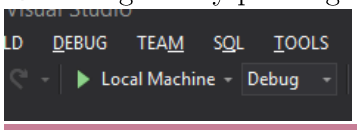
11. In the img folder (/assets/img/), drag and drop a picture as a logo for the game and called it whatever you called it in the BobbingLogo object (mainScreenLogo).

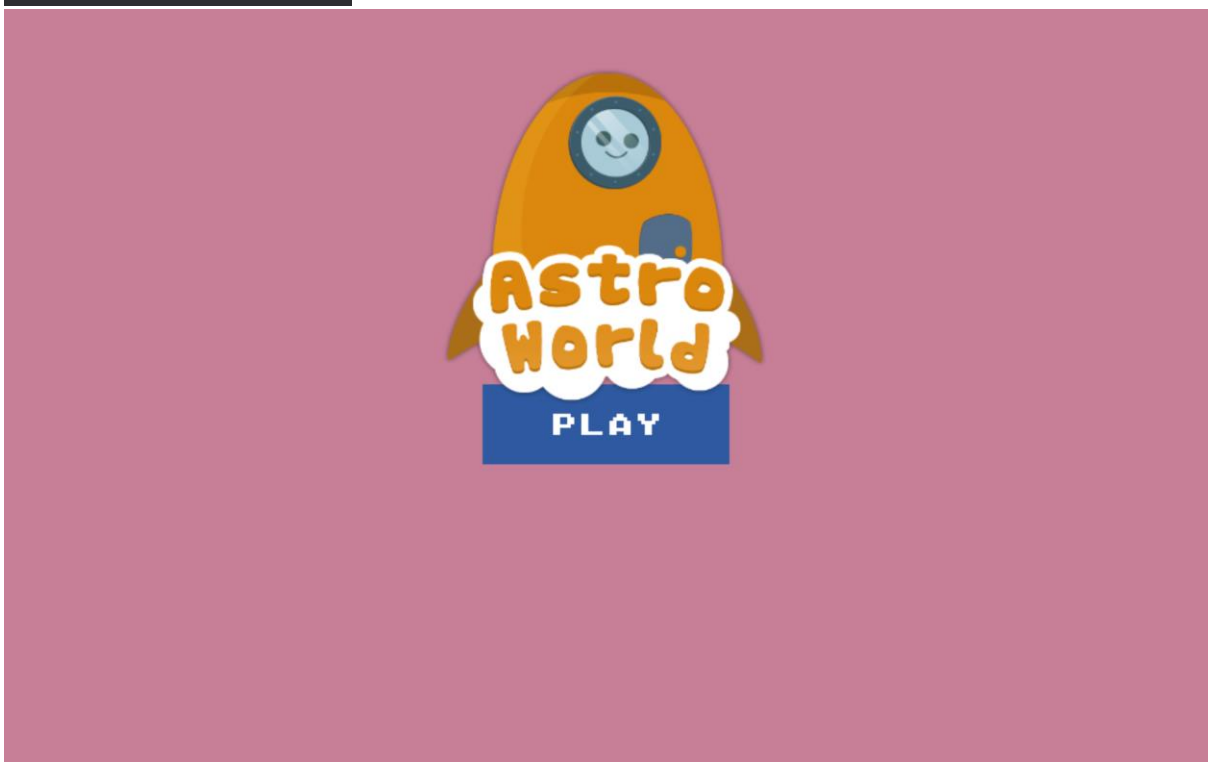


12. Open up the MelonJS Helper, and go to the resource tab. Make sure assets/img/mainScreenLogo.png is there, if not, restart MelonJS Helper. Now click 'Create resource.js'. The resource.js file will be updated to the references to your newly added logo.



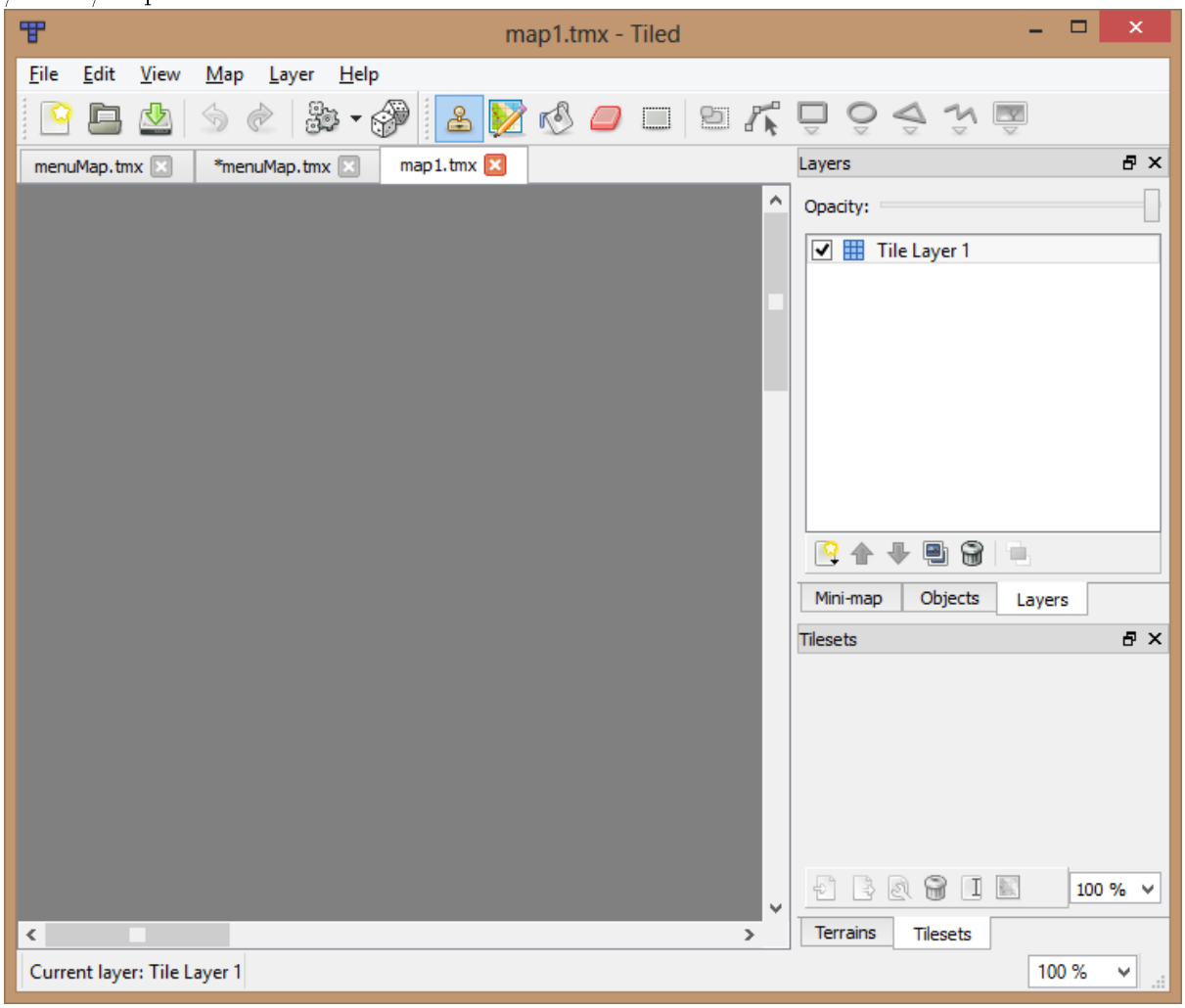
13. Run the game by pressing ‘Local Machine’ on Visual studio



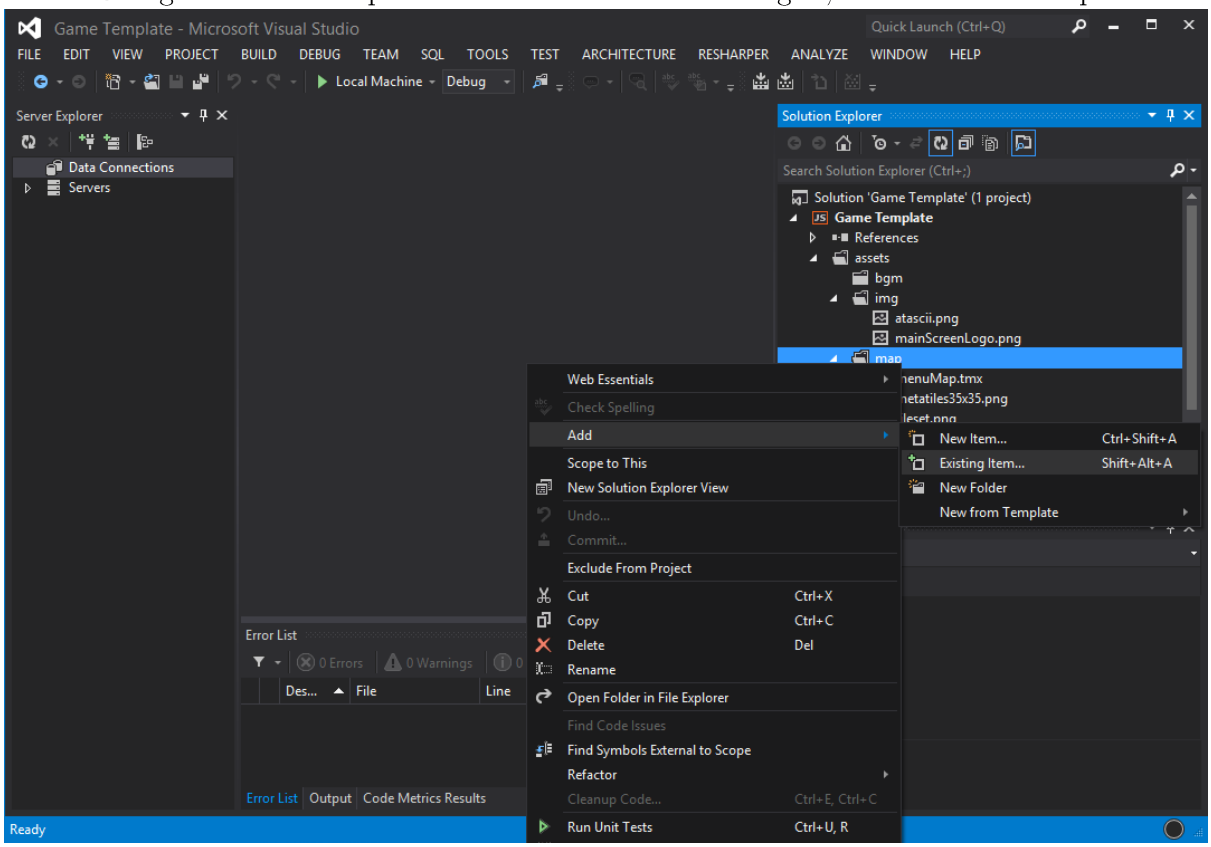


14. Currently when clicking ‘PLAY’ it will fail because the state we created earlier doesn’t exist. Go into Tiled, and create a new map and save it as ‘map1.TMX’ in the

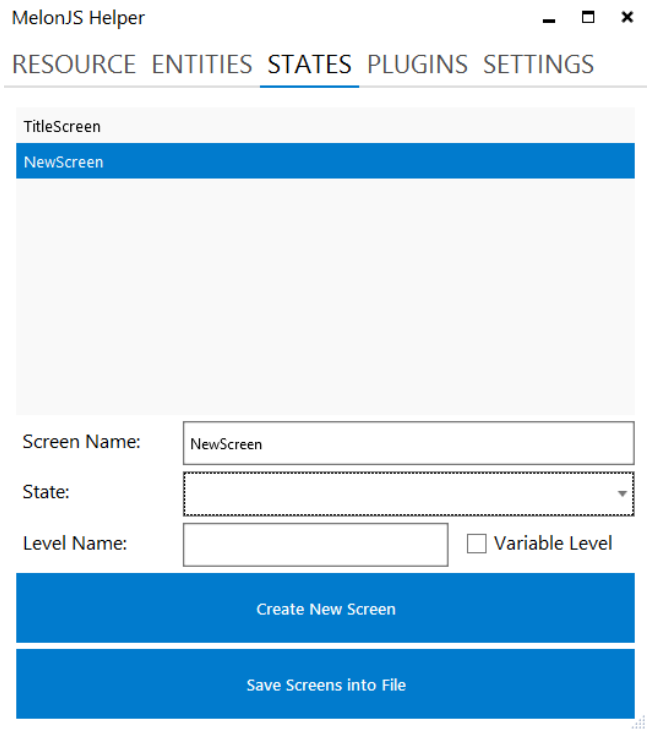
/assets/map folder.



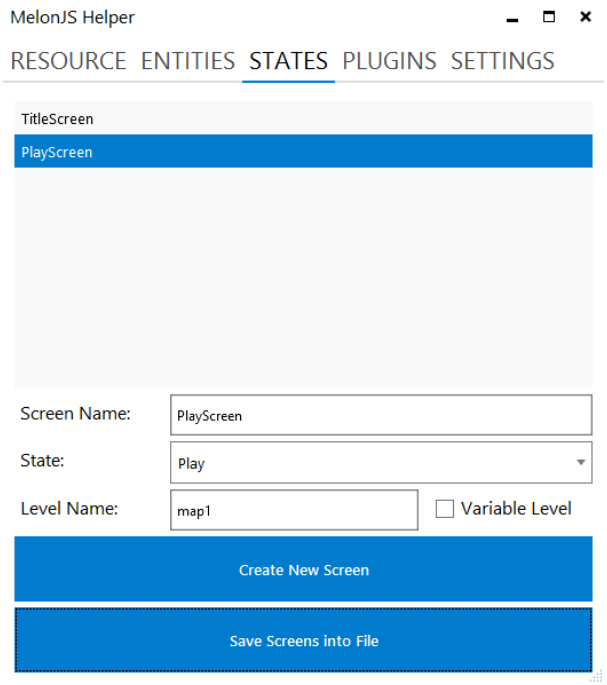
15. Drag and drop this file into the corresponding Visual Studio 2012 /assets/maps/ folder. Or right-click the map folder and select 'Add Existing...', then select the map.



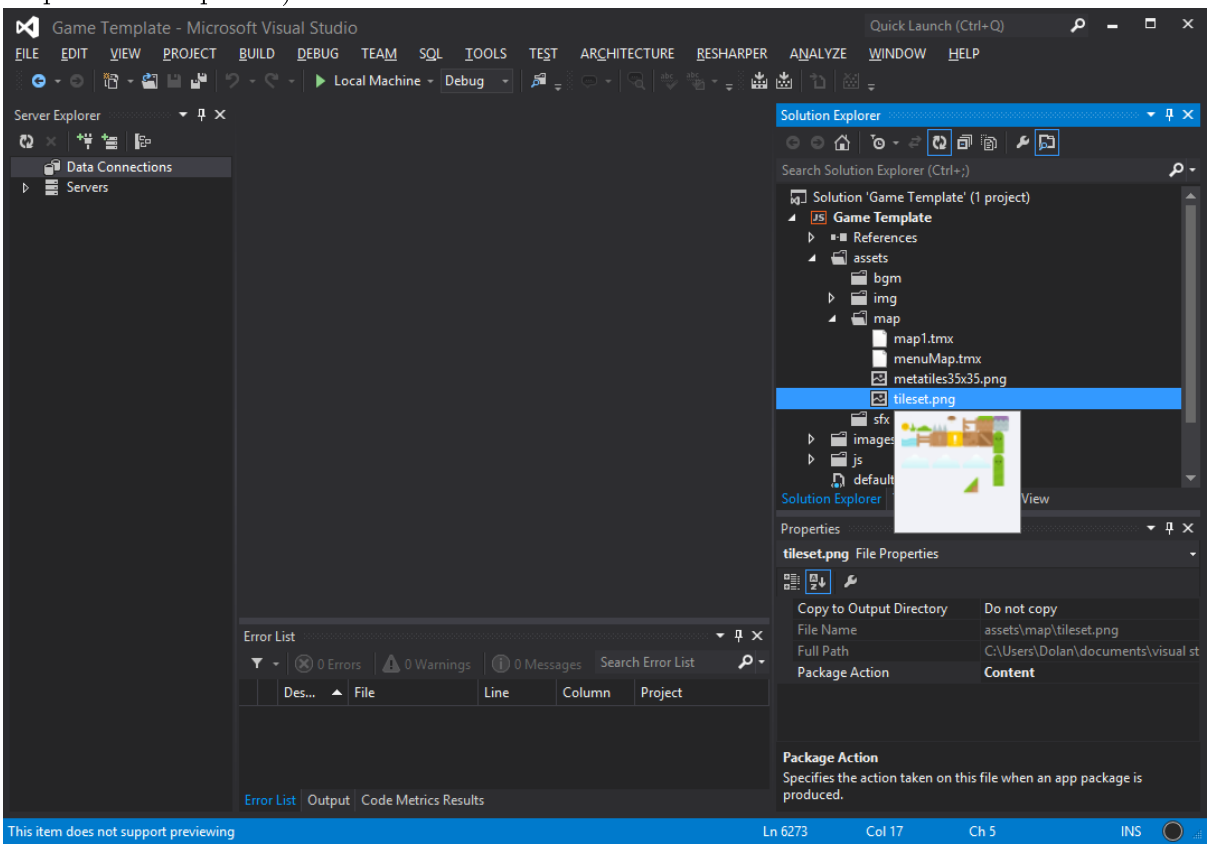
16. Open MelonJS Helper and click on the 'States tab. The title screen will be already included. To create a new screen, click on the 'Create New Screen' button. This will create a new screen



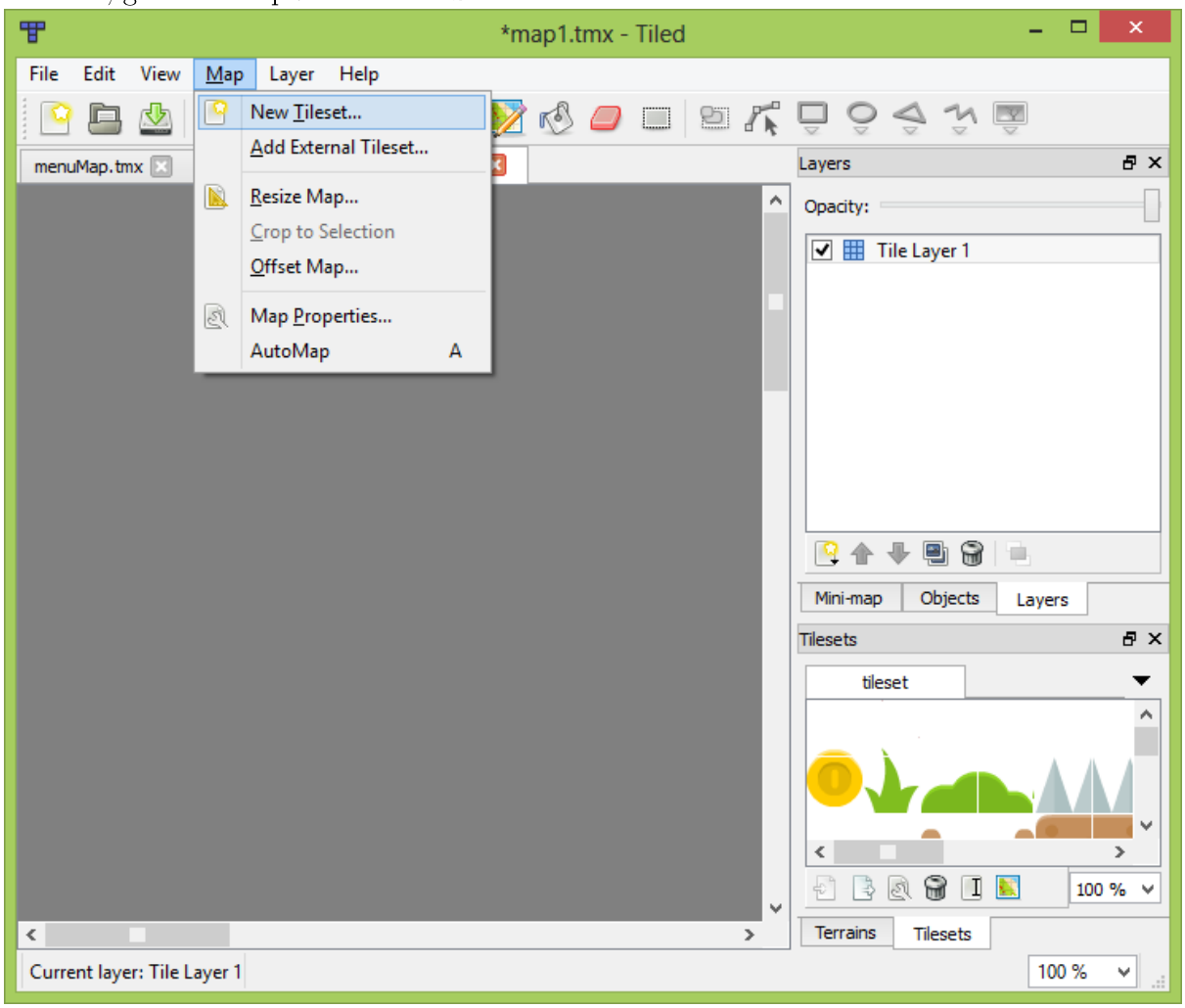
17. Name the screen whatever you want, select the 'Play' state and make the level equal to map1. Then click 'Save Screens into File'

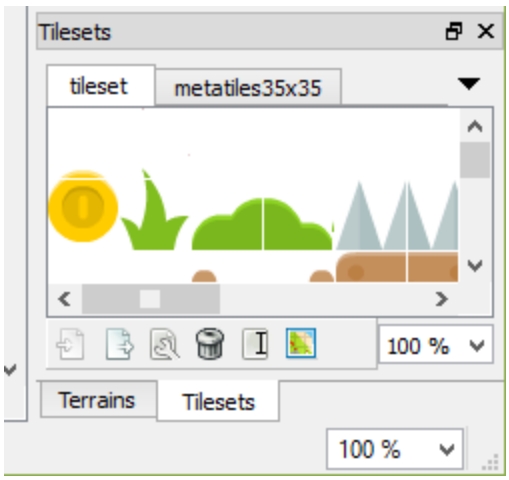


18. Create resource.JS again to include the map1.tmx made earlier.
19. We can now start designing our level. To do so, we need some sprites, so add a tileset and a collision layer into the assets/map/ folder in Visual Studio (same folder map1.tmx was placed):

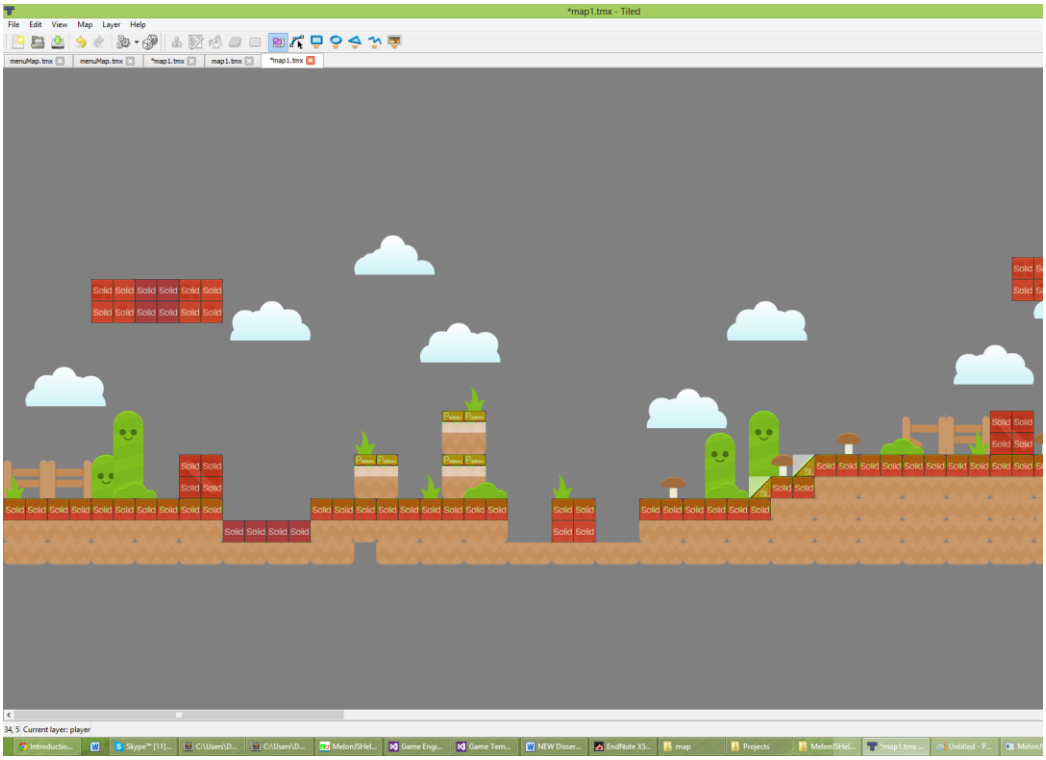


20. In Tiled, go into Map > New Tile Set to add a tile set.



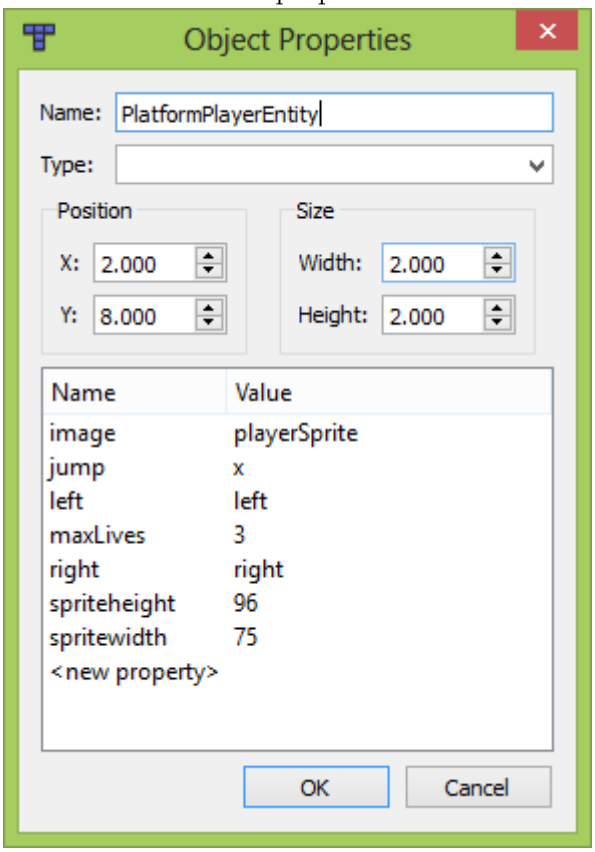


21. Start painting the new level, see http://gamedev.tutsplus.com/tutorials/level-design/introduction-to-tiled-map-editor/ for an in depth guide on how to paint levels in Tiled.
22. Afterwards, paint the collision layer in the tile layer called 'collision'. Use the collision tile-set provided for neatness. It is important to call it collision for MelonJS to recognise it as a collision layer.



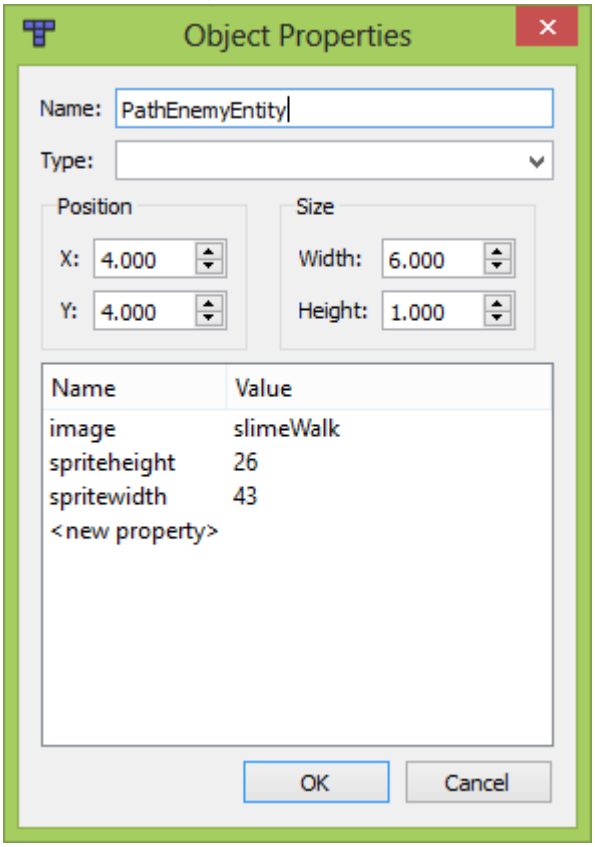
It should look something like this above now.

23. To add the main player, create a new object layer and call it ‘player’. Create a box and fill it with these properties:



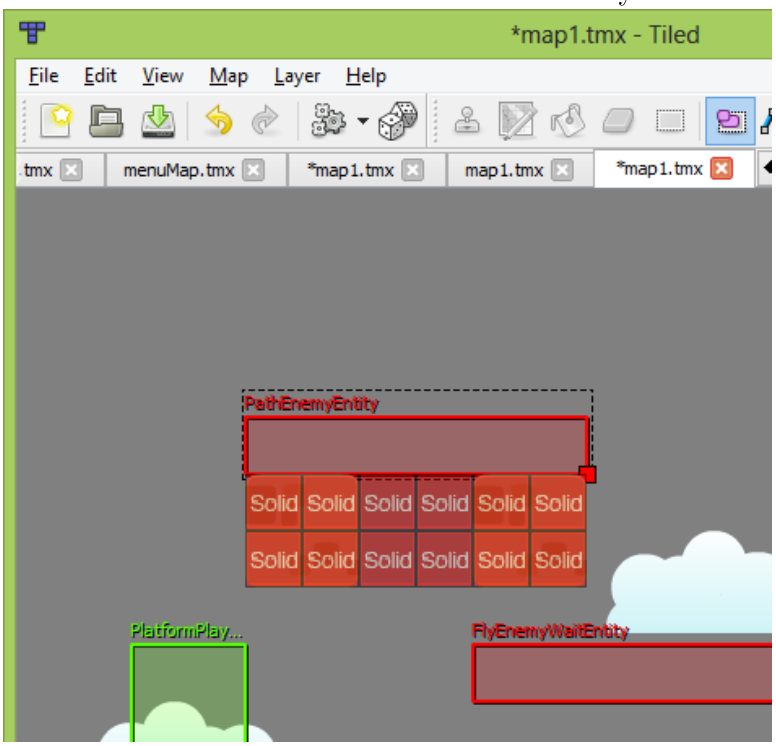
PlatformPlayerEntity is the pre-made platform player in the system. ‘maxLives’ gives the player a set number of lives. ‘left’, ‘right’, ‘jump’ are the controls for the character, the value is the key. For animated characters, the size of the individual frame of the animation must be specified. For me, it’s 96 x 75.

24. We haven't placed our sprite anywhere yet. Recall we called the image of the player 'playerSprite'. Create one of those names, and add it into Visual Studio and re-create the resource.js using the MelonJS Helper.
25. To add enemies, create a new box in on a new layer in Tiled and use these properties

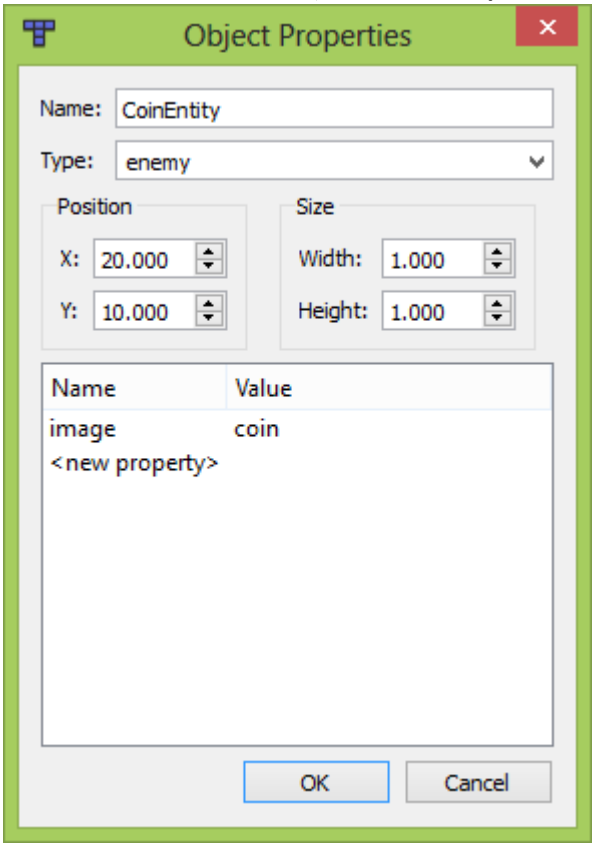


Again, the enemies are animated, so the spriteheight and width must be specified. The sprite name is also included. Be sure to add this image in the assets folder in Visual Studio and update with MelonJS Helper tool.
There are many enemy types in this template
- PathEnemyEntity
- PathWaitEnemyEntity
- FlyEnemyEntity
- FlyEnemyWaitEntity

The box it stretches indicates how far the enemy will travel.

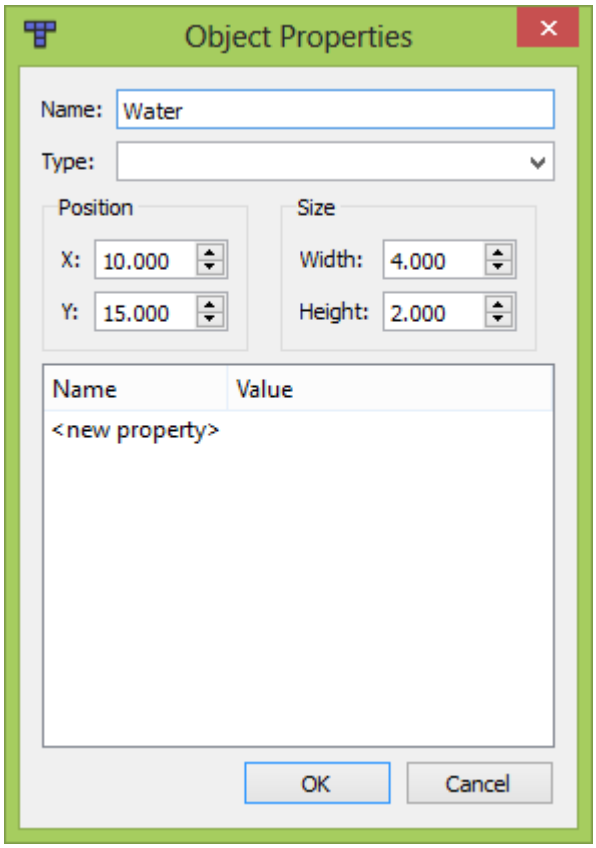


26. To add collectable coins, create an object as before and copy these settings.

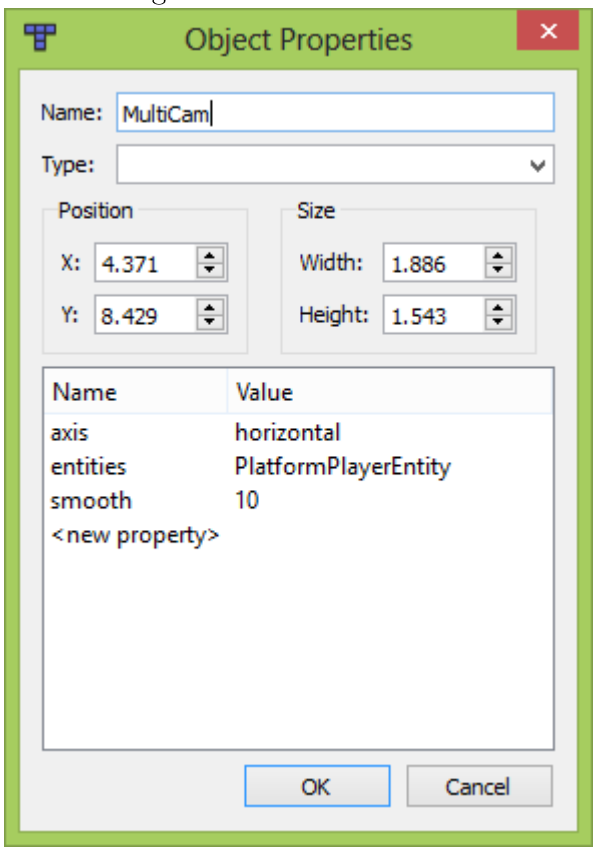


Remember to include the coin image in the assets/img folder and update with MelonJS Helper tool.

27. To create water, create an object as before and name it ‘Water’, it has no settings at this moment in time.

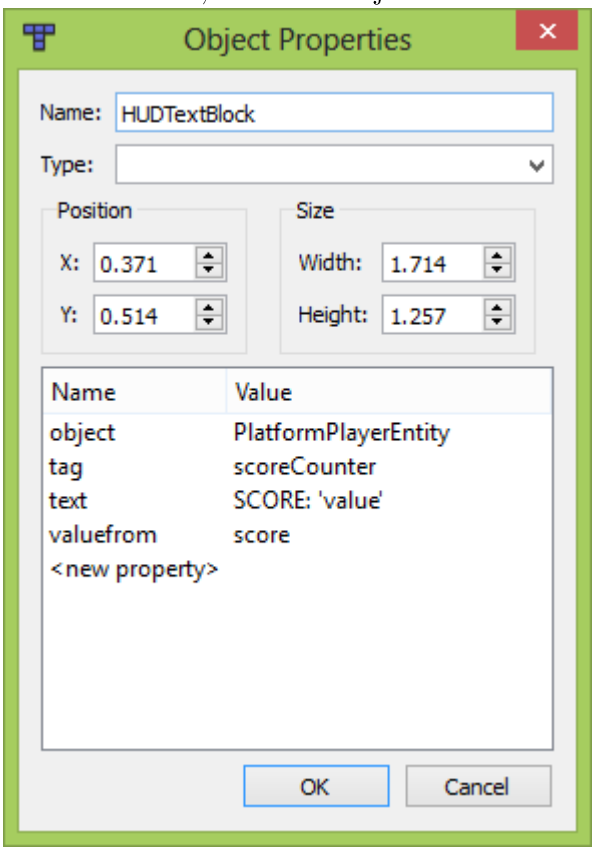


28. To create the camera, create an object as before and name it ‘ MultiCam’. Copy these settings:



The axis is what axis it will follow along. It could be vertical, horizontal, or both. The entities are the entities it will follow. At the moment it will only follow one entity, the player. The smoothness is how smooth it will track the player. A high value will create a very smooth slow camera, whereas a small one will be less smooth and faster. A value of 0 will lock the camera into the player.

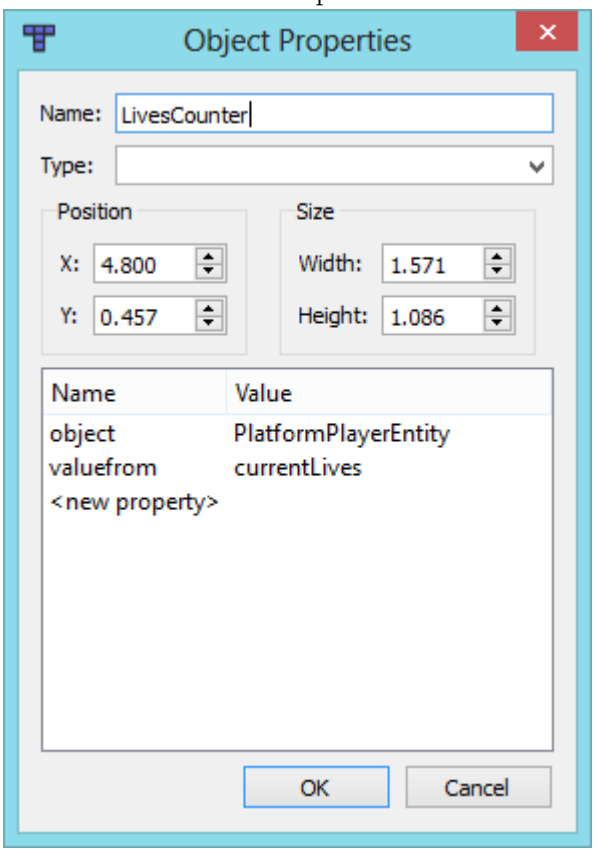
29. To create score, create an object called HUDTextBlock, and copy these settings:



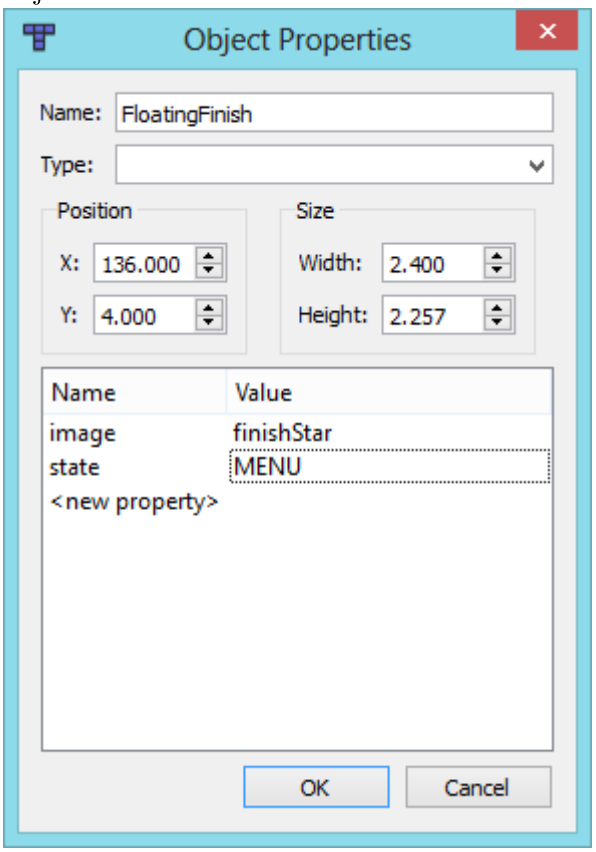
The object is the player, the tag is what the object will be referred to as if need be in code. The text is what it displays on screen. 'value' is a placeholder text which becomes replaced by the value it listens to. 'valuefrom' is the variable it listens to in the object. So this will listen to PlatformPlayerEntity.score and display 'SCORE: 0' on screen initially.

30. Similarly, a lives counter can be created by creating a 'LivesCounter' object. Although we could use 'HUDTextBlock', lives counter object is dedicated to showing

lives and thus much simpler to create.

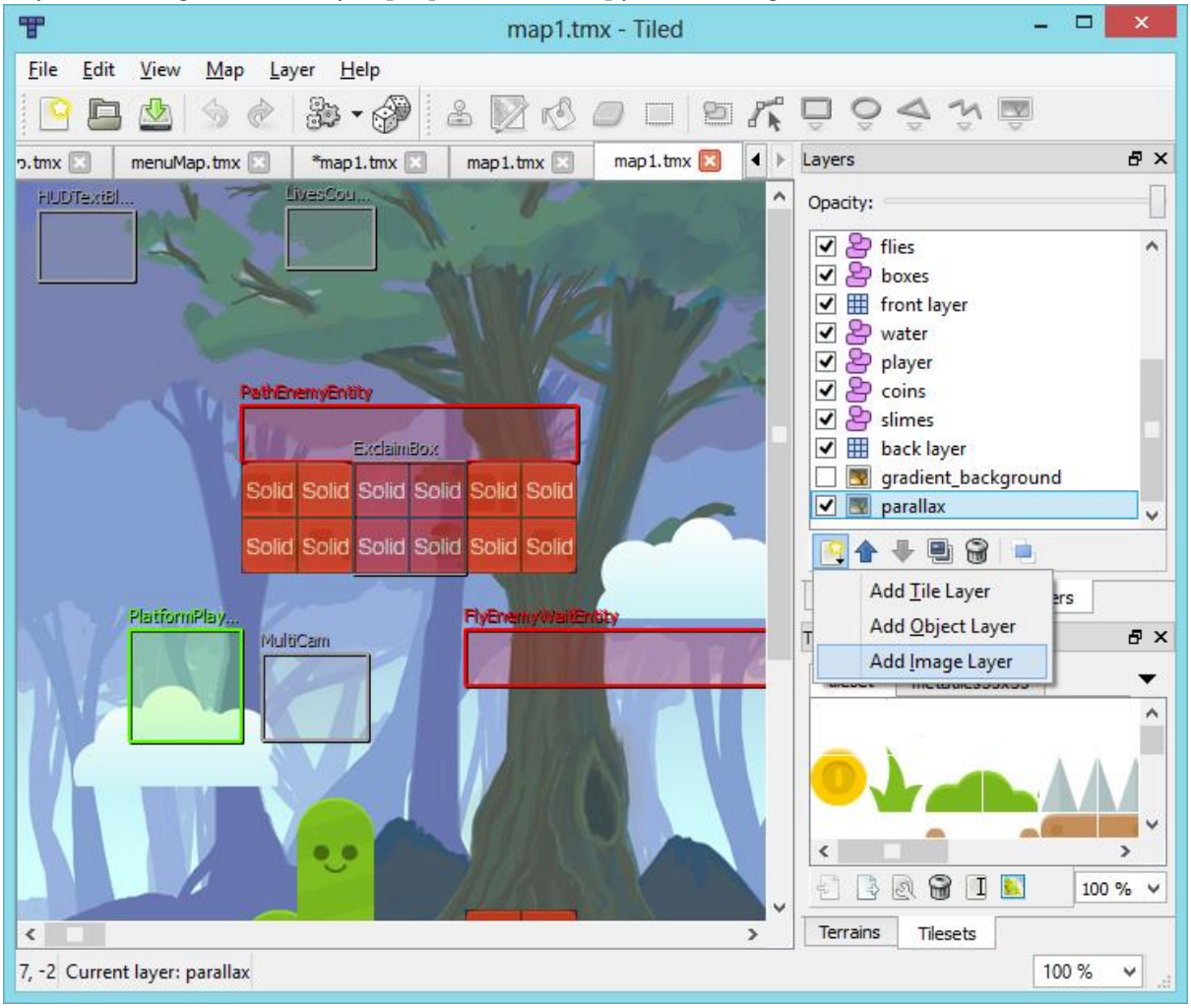


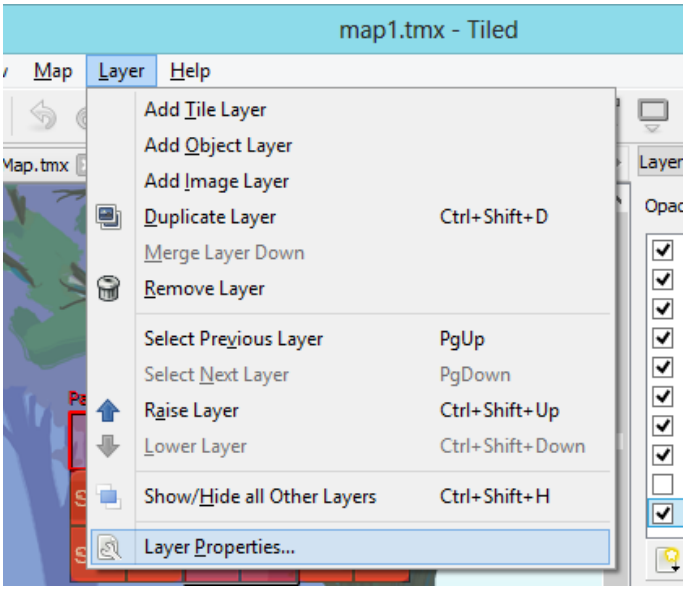
31. To create a finish line, either creates a 'FinishLine' object or a 'FloatingFinish' object.

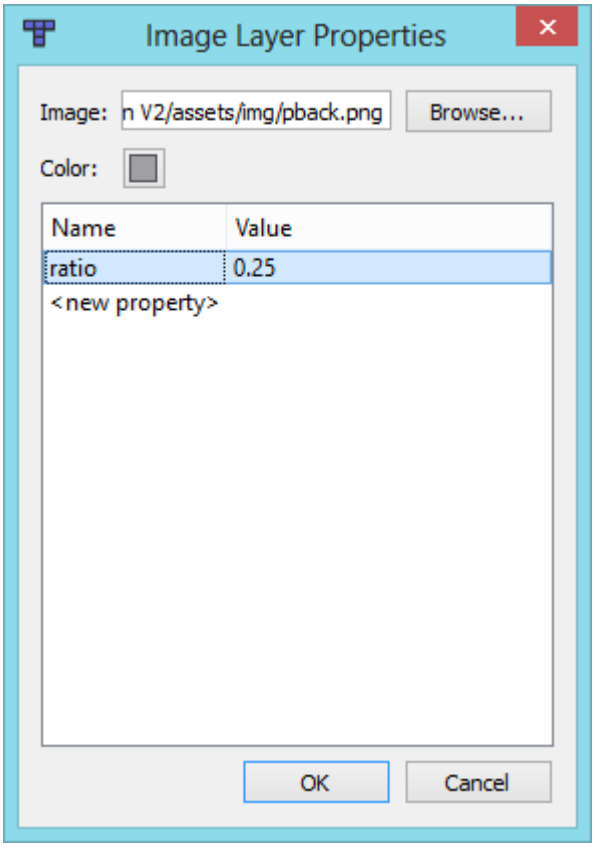


The image is what the finish line would look like, and the state is where the finish line would take you after it contacts the player.

32. To add parallax backgrounds, click on the new layer button and click 'New Image Layer'. Then go into its layer properties and copy the settings.







The image is the image you want as the background, and the ratio is how much the picture travels for every camera movement. This creates an illusion of depth.

33. To run the game, press Local Machine in Visual Studio.

10.3 Code Listing

10.3.1 Tests

```
        [Test]
        public void CodeBlockTestIfAddsCodeBlock()
        {
            StatementLine il = new StatementLine("test", "123");
            CodeBlock cb = new CodeBlock(il);
            //CodeBlock cb2 = new CodeBlock(new StatementLine("newStatement", "123"));
            List<string> lines = new List<string>();
            string lineToTest = "hello()";
            lines.Add(lineToTest);
            cb.AddCodeBlock(lines);
            //string line = "{" + lineToTest + ";}";
            string lineOut = cb.CodeBlocks[0].ToString();
            Assert.AreEqual(lineToTest + ";", lineOut);
        }

        [Test]
        public void CheckIfAddMethodFromStringWorks()
        {
            MelonJSObject mjsh = new MelonJSObject("game.CoolItem",
"me.ObjectEntity");
            string signature = "init: function() {";
            List<string> methodLines = new List<string>();
            methodLines.Add("this.parent()");
            methodLines.Add("},");
            mjsh.AddMethodFromString(signature, methodLines);
            Method m = mjsh.Methods[0];
            if (m.Name == "init" && m.Lines[0].ToString().Equals("this.parent();"))
            {
                Assert.Pass("Works!");
            }
            else
            {
                Assert.Fail("Doesn't work");
            }
        }

        [Test]
        public void CHeckIsRecognisedLine()
        {
            //MethodLine ml = new MethodLine("doStuff", "thing1", "thing2");
            string[] methodStrings =
            {
                "me.levelDirector.loadLevel()", "me.game.disableHUD()",
"me.state.set()", "this.parent()",
                "this.updateColRect()", "this.anchorPoint.set()", "this.bindKey()"
            };

            for (int i = 0; i < methodStrings.Length; i++)
            {
                bool valid = MethodLine.IsRecognisedMethodLine(methodStrings[i]);
                if (!valid)
                {
                    Assert.Fail("This method does not work: " + methodStrings[i]);
                }
            }
            Assert.Pass("Success! All methods work.");
        }
```

```
        [Test]
        public void CheckMethodGeneration()
        {
            Method m = new Method("testFunction: function(blah1, blah2, blah3)");
            string name = m.Name;
            string[] arguments = m.Arguments;

            if (name == "testFunction" && arguments[0] == "blah1" && arguments[1] ==
"blah2" && arguments[2] == "blah3")
            {
                Assert.Pass("Works!");
            }
            else
            {
                Assert.Fail("Failed.");
            }
        }

        [Test]
        public void CheckInvalidSyntaxReturnsNull()
        {
            NewMethod method = new NewMethod("init: function () {");

            List<string> lines = new List<string>
            {
                "this.patchme.DefaultLoadingScreen, \"onResetEvent\", function () ",
                ""
            };

            method.AddCodeBlock(lines);

            var output = method.CodeBlocks;

            foreach (var current in output)
            {
                Assert.Null(current);
            }
        }

        [Test]
        public void CheckMethodGeneration()
        {
            Method m = new Method("testFunction: function(blah1, blah2, blah3)");
            string name = m.Name;
            string[] arguments = m.Arguments;

            if (name == "testFunction" && arguments[0] == "blah1" && arguments[1] ==
"blah2" && arguments[2] == "blah3")
            {
                Assert.Pass("Works!");
            }
            else
            {
                Assert.Fail("Failed.");
            }
        }

        [Test]
        public void CheckGetSetStateString()
        {
            Screen s = new Screen("game.Bobby", "me.ScreenObject");
            s.State = new State("GAMEOVER");
            string stateString = s.GetSetStateString();
```

```
            Assert.AreEqual("me.state.set(me.state.GAMEOVER, new game.game.Bobby());",
stateString);
        }

        [Test]
        public void CheckIfRecognisedLine()
        {
            string[] lineStrings =
            {
                "this.gravity = 0", "this.vel.x = 10", "this.vel.y = 0", "this.visible
= true",
                "this.collidable = false", "this.type = \"mainPlayer\""
            };
            for (int i = 0; i < lineStrings.Length; i++)
            {
                bool valid = StatementLine.IsRecognisedStatementLine(lineStrings[i]);
                if (!valid)
                {
                    Assert.Fail("Line failed: " + lineStrings[i]);
                }
            }
            Assert.Pass("Success!");
        }


    public class XMLParserTest
    {
        private FileDirectories fd;
        private XDocument doc;

        [SetUp]
        public void SetUp()
        {
            fd = new FileDirectories
            {
                ResourceJSDirectory = @"c:\TestJSDirectory",
                AssetDirectory = @"c:\TestAssetDirectory",
                EntityFilesDirectory = @"c:\TestEntityFilesDirectory",
                ScreenFileDirectory = @"c:\TestScreenFileDirectory",
                HomeMadeEntitiesDirectory = @"c:\TestHomeMadeEntitiesDirectory",
                PluginsFileDirectory = @"c:\TestPluginsFileDirectory"

            };

            doc = new XDocument(
                new XDeclaration("1.0", "utf-8", "yes"),
                new XElement("Settings",
                    new XElement("ResourceJSDirectory", fd.ResourceJSDirectory),
                    new XElement("AssetDirectory", fd.AssetDirectory),
                    new XElement("EntitiesDirectory", fd.EntityFilesDirectory),
                    new XElement("ScreensDirectory", fd.ScreenFileDirectory),
                    new XElement("HomeMadeEntitiesDirectory",
fd.HomeMadeEntitiesDirectory),
                    new XElement("PluginsDirectory", fd.PluginsFileDirectory)));

        }

        [Test]
        public void Test_Saving_Settings()
        {
            XMLReaderWriter.SaveSettings(fd);
            XDocument readDoc = XDocument.Load("settings.xml");
```

```
            IEnumerable<XElement> settings = from row in
readDoc.Descendants("Settings") select row;

            FileDirectories outputFileDirectories = new FileDirectories();
            outputFileDirectories.ResourceJSDirectory = GetNodeValue(settings,
"ResourceJSDirectory");
            outputFileDirectories.AssetDirectory = GetNodeValue(settings,
"AssetDirectory");
            outputFileDirectories.EntityFilesDirectory = GetNodeValue(settings,
"EntitiesDirectory");
            outputFileDirectories.ScreenFileDirectory = GetNodeValue(settings,
"ScreensDirectory");
            outputFileDirectories.HomeMadeEntitiesDirectory = GetNodeValue(settings,
"HomeMadeEntitiesDirectory");
            outputFileDirectories.PluginsFileDirectory =
GetNodeValue(settings,"PluginsDirectory");

            Assert.AreEqual(fd, outputFileDirectories);
        }

        [Test]
        public void Test_Loading_Settings()
        {
            doc.Save("setings.xml");
            var readFile = XMLReaderWriter.ReadSettings();
            Assert.AreEqual(fd, readFile);
        }

        public string GetNodeValue(IEnumerable<XElement> element, string nodeName)
        {
            return element.Elements().First(x => x.Name.LocalName == nodeName).Value;
        }

    }
```

10.3.2 JavaScript unit tests

```
describe("Helper -  Rotate Point", function () {
        it("Rotate point 90 degrees", function () {
                var origin = new me.Vector2d(2,3);
                var point = new me.Vector2d(10,2);
                expect(Helper.rotatePoint(origin, point, 3.14)).toEqual(3);
        });
});


describe("Helper - Find Angle", function () {
    it("gives angle in radians", function () {
        var point1 = new me.Vector2d(2,3);
                var point2 = new me.Vector2d(2,5);
```

```
        expect(Helper.findAngle(point1, point2)).toEqual(1.5707963267948966);
    });
});

describe("Helper - get middle point", function () {
    it("should return a middle Vector2d", function () {
        var points = [];
        points.push(new me.Vector2d(2,3));
                points.push(new me.Vector2d(2,5));

        expect(Helper.getMiddlePoint(points)).toEqual(new me.Vector2d(2,4));
    });
});

describe("Helper - find the furthest point", function () {
    it("should return furthest points from set of points", function () {
        var points = [];
        points.push(new me.Vector2d(2,3));
                points.push(new me.Vector2d(3,5));
                //points.push(new me.Vector2d(2,5));
                var centrePoint = new me.Vector2d(2,2);

        expect(Helper.findFurthestPoints(points, centrePoint)).toEqual();
    });
});

describe("LightDraw Effects - CandleFills", function () {
    it("should ", function () {
        var points = [];
        points.push(new me.Vector2d(2,3));
                points.push(new me.Vector2d(3,5));
                //points.push(new me.Vector2d(2,5));
```

```
                var centrePoint = new me.Vector2d(2,2);

        expect(LightDrawEffects.CandleFill()).toEqual();
    });
});

describe("Light - Reduce Points", function () {
    it("should reduce points properly", function () {
        var points = [];
        points.push(new illuminated.Vec2(2,3));
                points.push(new illuminated.Vec2(3,5));

        expect(Light.ReducePoints(points)).toEqual(4);
    });
});

describe("Light - Add Point", function () {
    it("should add point to point list", function () {
        var points = [];
        points.push(new illuminated.Vec2(2,3));
                points.push(new illuminated.Vec2(3,5));

        expect(Light.ReducePoints(points)).toEqual(4);
    });
});

describe("Visible - Check Intersection", function () {
    it("should intersect", function () {
        var p1 = new me.Vector2d(2,3);
        var p2 = new me.Vector2d(2,6);
        var p3 = new me.Vector2d(4,5);
        var p4 = new me.Vector2d(1,2);
```

```
        expect(Visible.CheckIntersection(p1,p2,p3,p4)).toEqual(true);
    });

    it("should not intersect parallel lines", function () {
        var p1 = new me.Vector2d(2,3);
        var p2 = new me.Vector2d(2,6);
        var p3 = new me.Vector2d(4,3);
        var p4 = new me.Vector2d(4,6);

        expect(Visible.CheckIntersection(p1,p2,p3,p4)).toEqual(false);
    });
});

describe("AStar - Path finding search", function () {
    it("should find path to A-B and bring a list", function () {
        var points = [];
        points.push(new illuminated.Vec2(2,3));
                points.push(new illuminated.Vec2(3,5));

        expect(AStar.search(points)).toEqual(4);
    });
});
```

10.4 ZipApp Mock-up UI

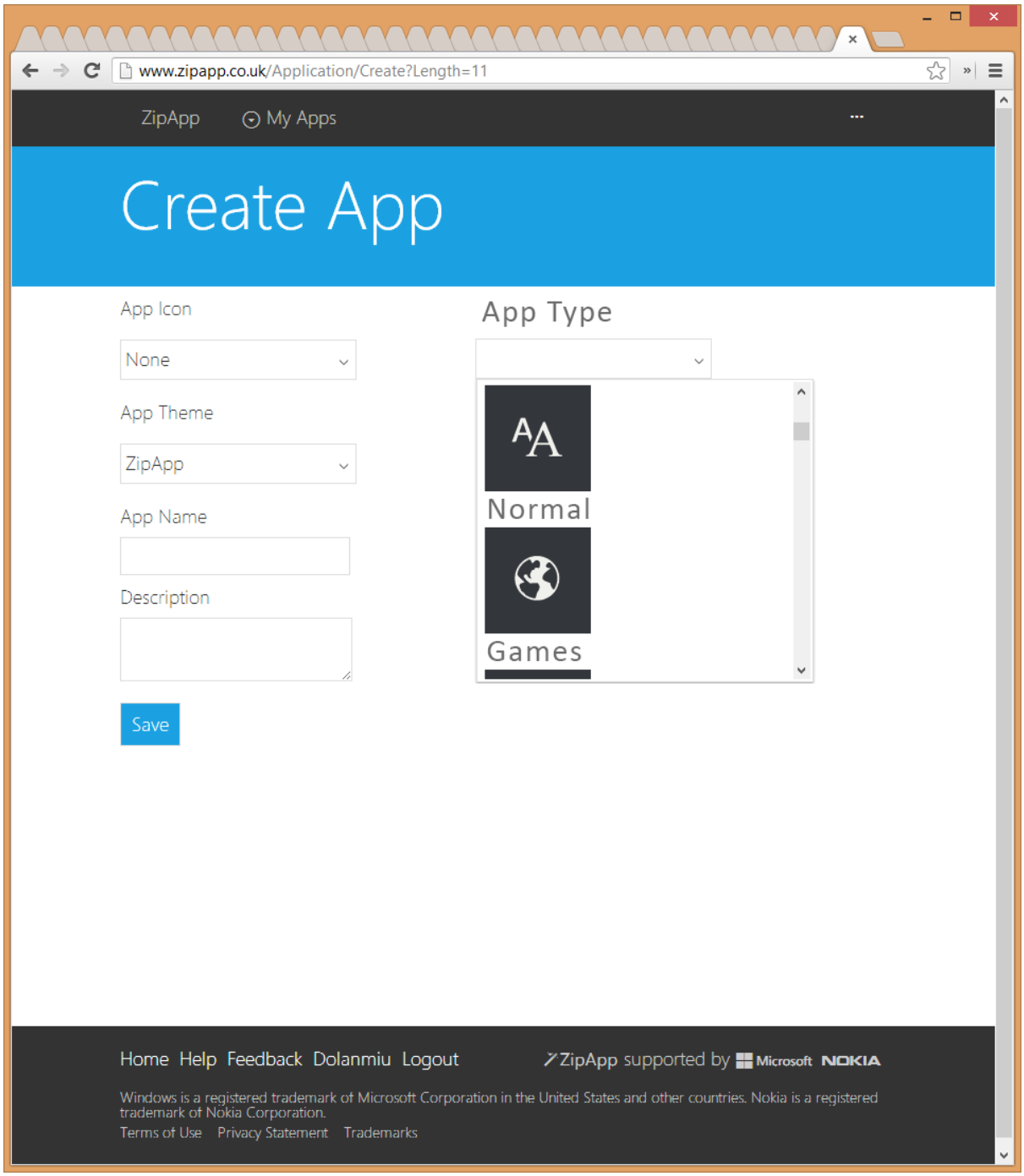


Figure 26 - ZipApp mock-up screen of what it would look like if the system was integrated into their website

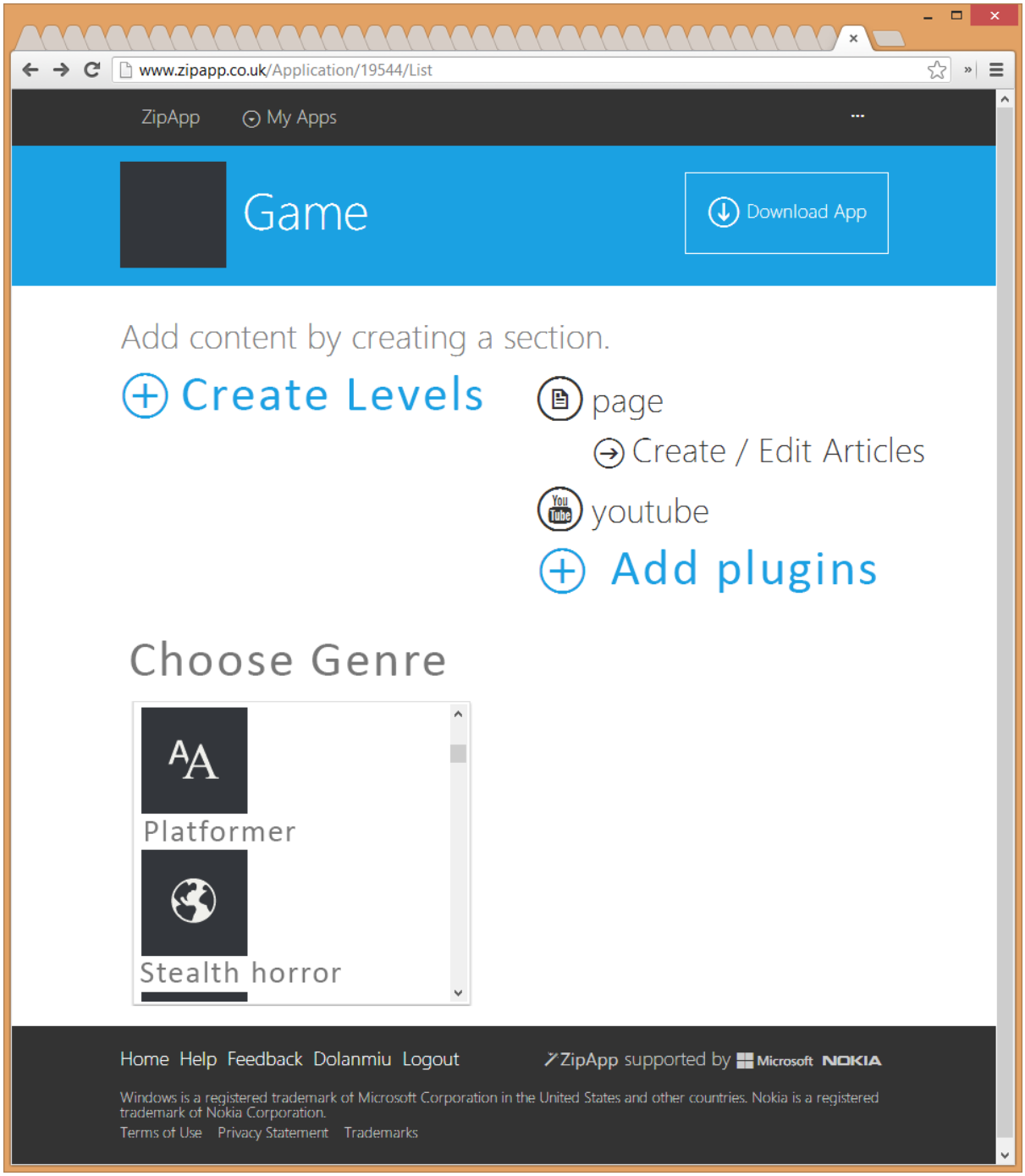


Figure 27 - The web page to customise the game. The genre can be selected and plugins can be added