



CM2303 - ONE SEMESTER INDIVIDUAL
PROJECT

40 CREDITS

FINAL REPORT

**A sprite-based video game engine
and editor for creating
component-based games**

Jack Edwards

Supervised by
Yukun LAI

Moderated by
Paul ROSIN

May 5, 2017

Contents

1	Introduction	4
2	Background	6
2.1	The Problem	6
2.2	Existing Solutions	6
2.2.1	Unity	7
2.2.2	GameMaker Studio	8
2.3	Technical Information	8
3	Specifications	10
3.1	Game Engine	10
3.1.1	Entity-component system	10
3.1.2	Scene Loading	11
3.1.3	Resource Management	11
3.1.4	Sprite & Font Rendering	11
3.1.5	Collision Detection and Reaction	11
3.2	Game Editor	11
3.2.1	Creating, Opening & Saving Projects	12
3.2.2	Entity View	12
3.2.3	Entity Properties View	12
3.2.4	Resources View	12
3.2.5	Scene Viewer	12
3.2.6	Running A Game	13
4	Design	14
4.1	Game Engine	14
4.1.1	Entity-component system	14
4.1.2	Scene Loading	18
4.1.3	Resource Management	18
4.1.4	Sprite & Font Rendering	19
4.1.5	Collision detection and reaction	20
4.2	Game Editor	20
4.2.1	Creating, Opening & Saving Projects	21
4.2.2	Entity View	22
4.2.3	Entity Properties View	22
4.2.4	Resources View	23
4.2.5	Scene Viewer	23
4.2.6	Running A Game	23
5	Implementation	24
5.1	Game Engine	24
5.1.1	Entity-component system	24
5.1.2	Scene Loading	25
5.1.3	Resource Management	26
5.1.4	Sprite & Font Rendering	26
5.1.5	Collision detection and reaction	27

5.2	Game Editor	27
5.2.1	Creating, Opening & Saving Projects	27
5.2.2	Entity View	28
5.2.3	Entity Properties View	28
5.2.4	Resources View	28
5.2.5	Scene Viewer	29
5.2.6	Running A Game	29
5.3	Unforeseen Problems	30
5.3.1	Compiling Project Resources	30
6	Results & Evaluation	31
7	Future Work	32
7.1	Build Settings	32
7.2	Sprite Animation	32
7.3	Sound Effects	32
7.4	Scene Viewer Improvements	32
7.5	Unit Testing	33
8	Conclusions	34
9	Reflection	35

List of Figures

2.1	The Unity game engine.	7
2.2	The GameMaker Studio game engine.	8
4.1	The process of saving a scene in the game editor and loading it in the game engine.	14
4.2	The entity-component system architecture used in the game engine. .	16
4.3	The main game loop for a scene.	17
4.4	The dictionary containing resources included in a project.	19
4.5	A mockup of the game editor user interface.	21
5.1	Code for the GetComponent method of the Entity class.	25
5.2	An example of using the OnCollisionEnter method.	27

1. Introduction

The aim of this project is to develop a 2D video game engine and game editor that allows users to create their own video games without requiring extensive prior game development experience. The primary beneficiaries of the project are developers with a small amount of experience in programming and game development that want to begin creating their own video games. The project was also designed to scale with the user's knowledge, so that it is not simply a learning tool for beginners and could be used by experienced developers to create potentially complex games.

The scope of the project originally encompassed three primary deliverables; a game engine, an editor and an example game. The game engine was to provide an entity-component system that would allow users to easily extend their games by creating modular components and attaching them to entities within the games. The engine would also feature several built-in components for accomplishing common game development tasks, such as drawing sprites to the screen or handling collisions between entities.

I decided to also include a game editor in the project to provide users with a simple way of creating and running games whilst minimising the amount of boilerplate code they would have to write. The editor allows users to add entities to scenes in their game and attach components to these entities - whether they are components included in the game engine or custom components written by the user. The editor also allows the user to edit the properties of a component using controls within the editor, allowing the user to change variables within their game without touching the code.

In order to demonstrate the capabilities of the engine and the editor, I decided to create an example game that showcases the different available features. The game is an implementation of the popular Breakout[9] type of game, in which the user controls a paddle and must move the paddle horizontally to deflect a moving ball and bounce it towards a number of bricks in the scene to destroy them. The game demonstrates several features available in the engine:

- A flexible entity-component system[5]
- Game loading from an XML[?] file
- Sprite and font rendering
- User input
- Collision detection & reaction between entities

The original scope of the project was to include functionality allowing users to add audio effects and animated sprites to their game, as well as a system for creating custom user interfaces. During the course of the project I decided that these features should be left out in order to focus on more important features, such as the entity-component system, data serialisation and editor tools. I believe that this was a small sacrifice to make to ensure that the core of the project was fully-functional and stable.

Due to the complexity of the project as a whole, I decided that the best approach would be to utilise the Scrum agile development methodology. This methodology

involves working in sprints to rapidly develop, test and adjust different functionality within the software. During this project I worked in one week sprints and focused on different, modular areas of the system during each sprint. This was a very flexible approach to designing and implementing the system because if I realised that certain functionality needed to be changed I was able to modify it without compromising any of parts of the system.

It is assumed that the end user will be running a Windows[\[10\]](#) machine with .NET Framework 4.5.2[\[12\]](#) or later installed. The end user should also have access to the Internet, because parts of the engine are downloaded and referenced once the user creates a new game project.

This report aims to cover background information regarding the project, the planning and design of the system, the implementation of the different software products, an evaluation of the final deliverable and future work that could be completed to improve the system.

2. Background

2.1 The Problem

Over the past decade, the amount of video games on the market made by independent developers has increased dramatically and continues to rise on a daily basis. This is partially due to the number of high-quality game development software solutions that are freely available. Tools like Unity[6] are flexible enough that they are relatively easy to learn whilst still being capable of producing high quality, large-scale games. This has allowed a tremendous amount of amateur game developers to learn how to create games and release them for the world to play.

Although Unity and other similar tools are great tools for creating games, many of them focus primarily on the development of 3D games. With the rising popularity of independent games, the popularity of 2D games has also increased. This is partially due to the fact that 3D games can often be more challenging to create as an independent developer due to the immense number of resources that are often required for the game, such as high detail 3D models and large, detailed game worlds. Although 2D games are becoming more and more popular, many of the most commonly used game engines still have limited functionality when it comes to developing 2D games. For example, Unity supports the development of 2D games but it is not strictly a 2D game engine, so it utilises a lot of workarounds for creating 2D games within a 3D space, which can be confusing to beginner game developers.

Game engines that specialise in 2D games do exist, although many of them are difficult to use and others are not flexible enough to be used when creating large, complex games. For example, GameMaker Studio[7] specialises in the development of 2D games, although games created using it must be programmed using their proprietary scripting language, which is not as powerful as other programming languages and can be much slower.

This project aims to solve this problem by providing an engine that specialises in the development of 2D video games. Focusing on 2D from the ground up will ensure that the engine and the editor simplifies a lot of aspects of 2D game development as much as possible. The engine will allow games to be created using an entity-component system, which will allow users to create games in a simple manner by writing custom components and attaching them to arbitrary entities. This will ensure that the engine is easy to learn and use.

2.2 Existing Solutions

A number of existing game engines provide a partial solution to the problem, although they each have their disadvantages. Most of these disadvantages are related to the fact that the engine was made to be generic so that it could be used for creating 2D and 3D games, or that the engine does not provide a flexible method of adding functionality to a game, making it difficult to create large or complex games.

2.2.1 Unity

Unity[6] is a cross-platform game engine developed by Unity Technologies and is used to develop 2D and 3D games for a wide range of platforms. Unity makes heavy use of the entity-component system architectural pattern by allowing users to add entities to a game and attach components to them. For example, a user could write a custom component that causes an entity to rotate over time, and then attach that component to any entity that they want to rotate. This makes it very easy to start learning how to make games in Unity and minimises code duplication.

Despite the advantages, Unity is firstly a 3D game engine and the functionality related to 2D development is mostly an afterthought. This can make it difficult to perform certain tasks that would be simple in an engine that specialises in 2D, such as drawing pixel-perfect sprites. The fact that Unity is firstly a 3D engine means that the editor features a lot of tools specific to 3D development that are not required for 2D development. This can make it somewhat difficult to learn 2D development in Unity because the amount of available tools is overwhelming. An engine that specialises in 2D games would be much simpler to use by only displaying tools that are useful for 2D development.

Unity being a 3D-first game engine also introduces some unusual nuances when developing 2D games, such as the fact that all sprites in a 2D game are textured planes within 3D space. This means that if you wanted to, you could transform sprites in 3D space, which could produce some unwanted results. A game engine that primarily focuses on 2D games would not have the concept of 3D space whatsoever.

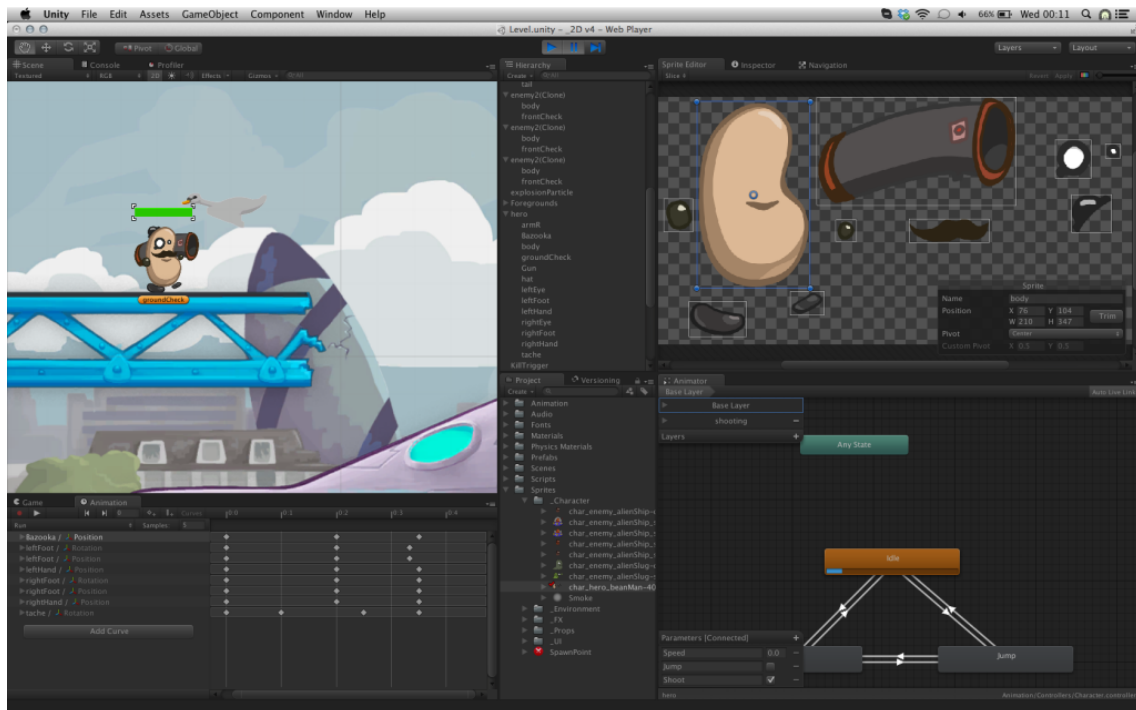


Figure 2.1: The Unity game engine.

2.2.2 GameMaker Studio

GameMaker Studio[7] is a 2D game creation system that allows users to create games. Although GameMaker does focus on the development of 2D games, it may not be suitable for developing large games. This is due to the fact that development is mostly performed using its proprietary drag-and-drop system that allows users to create games by dragging and dropping action sequences onto different objects, which determines the behaviour of the object. Although this system makes it very easy for beginners to learn how to make games, it is not very powerful and would not be suitable for a large game. GameMaker also allows you to use its proprietary programming language GameMaker Language[8] (GML) to create games, although it is a scripting language and is substantially slower than other languages.



Figure 2.2: The GameMaker Studio game engine.

2.3 Technical Information

Due to the fact that the engine is being developed for use on computers running the Windows operating system, I decided to use the C#[11] language with .NET Framework 4.5.2 to create the engine and the editor because the language supports many tools and frameworks to aid software development for Windows. Using .NET Framework also allowed me to develop the front end of the game editor using Microsoft's Windows Presentation Foundation[3] (WPF) technology, which allows software with a graphical user interface to be created with relative ease.

To develop the game engine, I decided to use the Monogame framework to handle the core mechanics of the engine, such as sprite rendering and user input detection. Utilising Monogame[2] allowed me to focus on the development of the user-facing portion of the engine and the interaction between the engine and the editor, without being concerned about implementing any low-level features, which would likely be

a time consuming task.

The aim of this project is to develop a software application that can be used to create 2D video games without any extensive prior development experience. In order to achieve this aim, this project will identify the current software solutions available for developing 2D games, determine their advantages and disadvantages, establish the areas in which improvements can be made and implement a software solution that includes these improvements.

3. Specifications

This project aims to provide a set of software products for the Windows platform that can be used to create 2D video games by utilising an entity-component architecture. In order to accomplish this, two software products must be developed - a game engine and a game editor. Although the two software products will work in unison, they are completely different types of applications and thus will be designed and implemented differently. Due to this, I have decided to discuss the specifications of the two applications separately.

3.1 Game Engine

The game engine software will be the core of a game created using the system and will contain the main entry point to the application. The game engine will manage the entity-component system and tasks such as the loading of scenes and rendering to the screen. To create a game, a user will simply provide the engine with scenes, entities and components and the engine will handle the low-level details behind the scenes. The game engine should also expose a library of helpful classes to be used when writing code for a game, including classes and static methods provided by MonoGame.

For the engine to be able to perform these tasks, it will require the features outlined in the following sections to be implemented.

3.1.1 Entity-component system

To achieve the level of flexibility required to make the engine simple to use, it must use an *entity-component system* (described in detail in section 4.1.1). Using an entity-component system will allow users to easily add functionality to their games by writing a custom component and adding it to an entity. The pattern favours composition over inheritance, which reduces the degree of complexity when making a game.

The entities in a game should be stored in a hierarchical format, allowing entities to be set as the parent or child of another entity. Using a tree structure will allow the user to organise the entities in their game. For example, they may create an entity named "Player" which has two children entities named "Sword" and "Shield". Every entity should also have a Transform component by default, which defines the position, scale and rotation of an entity. The position of an entity should be relevant to its parent so that when an entity is moved, all of its children are moved accordingly.

Each entity should have a name associated with it and functionality should be included to allow a user to search for an entity (or multiple entities) in a scene by name. An entity should also have an 'enabled' status that can be set by the user, allowing some entities to be disabled so that the engine will not update them.

3.1.2 Scene Loading

The engine must be able to load a scene from an XML file. A scene is mostly a container object that contains a hierarchy of entities. When a user creates a scene using the editor it will be outputted to an XML file, containing data for all of the entities in the scene and the components attached to each entity. The engine must be able to receive a scene XML file as an input and use the data to construct the entities and components within the game engine. This will allow users to create valid games using the editor and set default values for the properties of components in the game.

3.1.3 Resource Management

When a scene is loaded, the engine should load all of the required resources for the scene. For example, if a scene is loaded and a component in the scene references an image file called "PlayerTexture", the image must be found and loaded to be used within the game. The engine should also provide functionality to load resources at runtime so that the user is not restricted to adding resources using the editor.

3.1.4 Sprite & Font Rendering

The engine should be capable of rendering textured sprites to the screen as well as fonts. The position, scale and rotation of each sprite or text entity should be defined by the transform of the entity that the component is attached to. The user should also be able to specify a colour to render the text, or a colour to add to a sprite's texture as a tint.

3.1.5 Collision Detection and Reaction

The engine should be able to detect collisions between entities that have a collider component. If a collision is detected, the user should be informed of the collision and should be able to retrieve a reference to the two entities involved in the collision.

3.2 Game Editor

The game editor software should be an application with a graphical user interface that allows users to create games using actions within the editor, as opposed to coding everything manually. Giving users access to the editor to create games will greatly improve the overall usability of the system because it handles a lot of the complicated aspects of development for the user. The user should be able to create and open existing projects and scenes, add entities to scenes, add components to entities and edit the properties of components. The editor should then output an XML scene file to be read as an input by the engine.

To meet the requirements of this project, the editor should include the specific features outlined in the following sections.

3.2.1 Creating, Opening & Saving Projects

The editor should have the ability to create new projects, save existing projects and open existing projects. A project file would contain information specific to the project such as the name of the project and the resources it contains. The editor should also be able to save, create and open existing scenes. A scene is a container for entities and multiple scenes may be used to separate a game by levels or areas within the game. When a user has a scene open within the editor they should be able to add entities to the scene, add components to different entities within the scene and edit the properties of components. The user should then be able to save the scene to a file that can be used as an input to the game engine allowing it to run the scene.

3.2.2 Entity View

The editor should feature an 'entity view', in which the entities within the current scene are displayed in a tree view. This would allow users to view the current entities (and their children) in the current scene, as well as allowing users to select an entity. The selected entity would be referenced in other areas of the application when certain actions are taken. For example, when a user adds a component to an entity, it would be added to the entity that is currently selected in the entity view. The selected entity would also determine what is shown in the entity properties view (See section 3.2.3).

3.2.3 Entity Properties View

The editor should feature an 'entity properties view' that displays details of the currently selected entity, such as its name and whether or not it is enabled. This view should also display details about all of the components attached to the currently selected entity. It should display the component name as well as appropriate controls for each of the editable properties on the component. For example, if a component had a property with the data type **string**, a textbox would be displayed that the user could use to edit the value of the property. As another example, if a component had a property with the data type **Point**, two text boxes would be displayed to allow the user to enter the value for the X and Y properties. This allows users to update the properties of a component through the editor with ease by using the different controls that are displayed dependent on the data type of the property.

3.2.4 Resources View

The editor should feature a 'resources view' that displays the resources available in the current project. The resources view should allow the user to view the different resources in the current project and add existing resources to the project.

3.2.5 Scene Viewer

The editor should feature a scene viewer, which would allow users to see an example of what the game will look like when it is running, to aid tasks such as positioning and resizing entities. The scene viewer should display all sprites and fonts in the

game at their specified position, scale and rotation. The scene viewer should also update when an entity's transform is changed - for example, if the user moves an entity's transform to the right, it should move accordingly in the scene viewer to reflect the change.

3.2.6 Running A Game

The editor should have the ability to begin the game and load the scene that is open in the editor when the game is run. This allows users to quickly test their game without having to completely build the project and search for the compiled executable. The user should be able to click a button in the main menu of the editor to run the game, which will launch the game executable in a new window.

4. Design

The complex nature of this project deemed it necessary to put a lot of thought into the design process to ensure that the system was viable. To accomplish this, the game engine and game editor were designed separately, whilst also ensuring compatibility between the two applications. This helped to simplify the process because it meant that there were no complex dependencies between the two applications - the editor simply outputs a file that the engine can receive as an input, as seen in figure 4.1. The following sections describe the design process of the game engine and the game editor separately, focusing on the core functionality of each application.

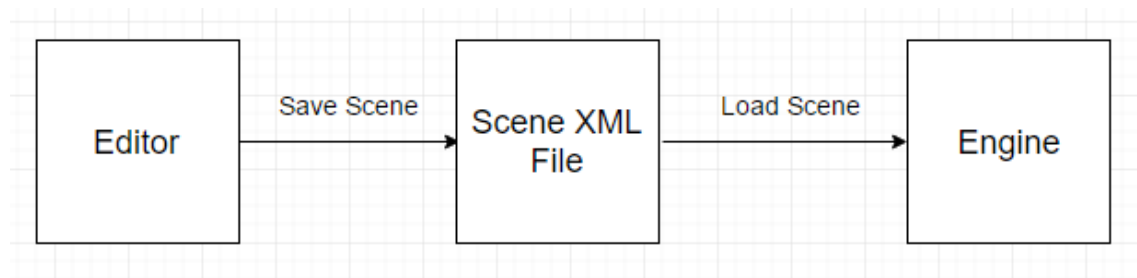


Figure 4.1: The process of saving a scene in the game editor and loading it in the game engine.

4.1 Game Engine

Due to the fact that the primary aim of this project is to provide a game creation system that boasts ease of use by utilising an entity-component system, the low level aspects of the engine (e.g. rendering and user input) are not required to be implemented from the ground up. To allow more time to be allocated to the implementation of the higher level functionality of the system, it was decided that the MonoGame framework should be used to manage the core of the engine. MonoGame will also provide a library of helper classes and functions that users can utilise when creating their game. The follow sections outline the design of each major component within the system.

4.1.1 Entity-component system

What is an entity-component system?

Entity-component system is an architectuerual pattern that utilises composition over inheritance and is often used in game development because of its flexibility. The pattern involves a game that comprises of many entities, which are each simply containers for components. Each entity does not have any specific behaviour in itself and its functionality and state is comprised of the components attached to the entity. This can allow for some very flexible behaviour within a game and makes it very easy to add functionality to an entity.

As an example, consider a game that features three different entities - humans, fish and frogs. Humans are able to walk on land, fish are able to swim in water

and frogs are able to do both. If inheritance were used in this scenario, entities would have to inherit from multiple different types of base entities depending on what functionality they require, causing the code to become overly complex very quickly. This would also lead to various problems regarding circular dependencies.

Using composition instead of inheritance simplifies this greatly. By using composition, we could simply add a 'Walk' component to the human entity, add a 'Swim' component to the fish entity and add both components to the frog entity. Due to the fact that an entity is simply a container, components can be added and removed from an entity at runtime to disable and enable functionality, making it exceptionally easy to implement features that could be increasingly complex if inheritance were used.

Entity-component system in the game engine

To ensure that the engine is easy to use, it was decided that an entity-component system was a necessary architectural pattern to implement. Although the pattern is widely recognised, there are various different ways that it can be implemented. For example, some implementations feature 'systems' that process different types of components, whereas other implementations allow components to process themselves.

It was decided that the system required a solid design early in the project timeline because the editor would also have to conform to the same design to ensure compatibility between the two applications.

Firstly, it was decided that a game should be separated into a collection of scenes, allowing users to create a logical separation between sections of their game. For example, a different scene could be used for each level in a game. A scene would contain a collection of entities, while each entity would contain a collection of components and a collection of children entities. This structure can be seen in figure 4.2. Entities within a scene are structured in a hierarchical tree, allowing entities to have children and a parent. This allows users to maintain a structure for the entities within their game.

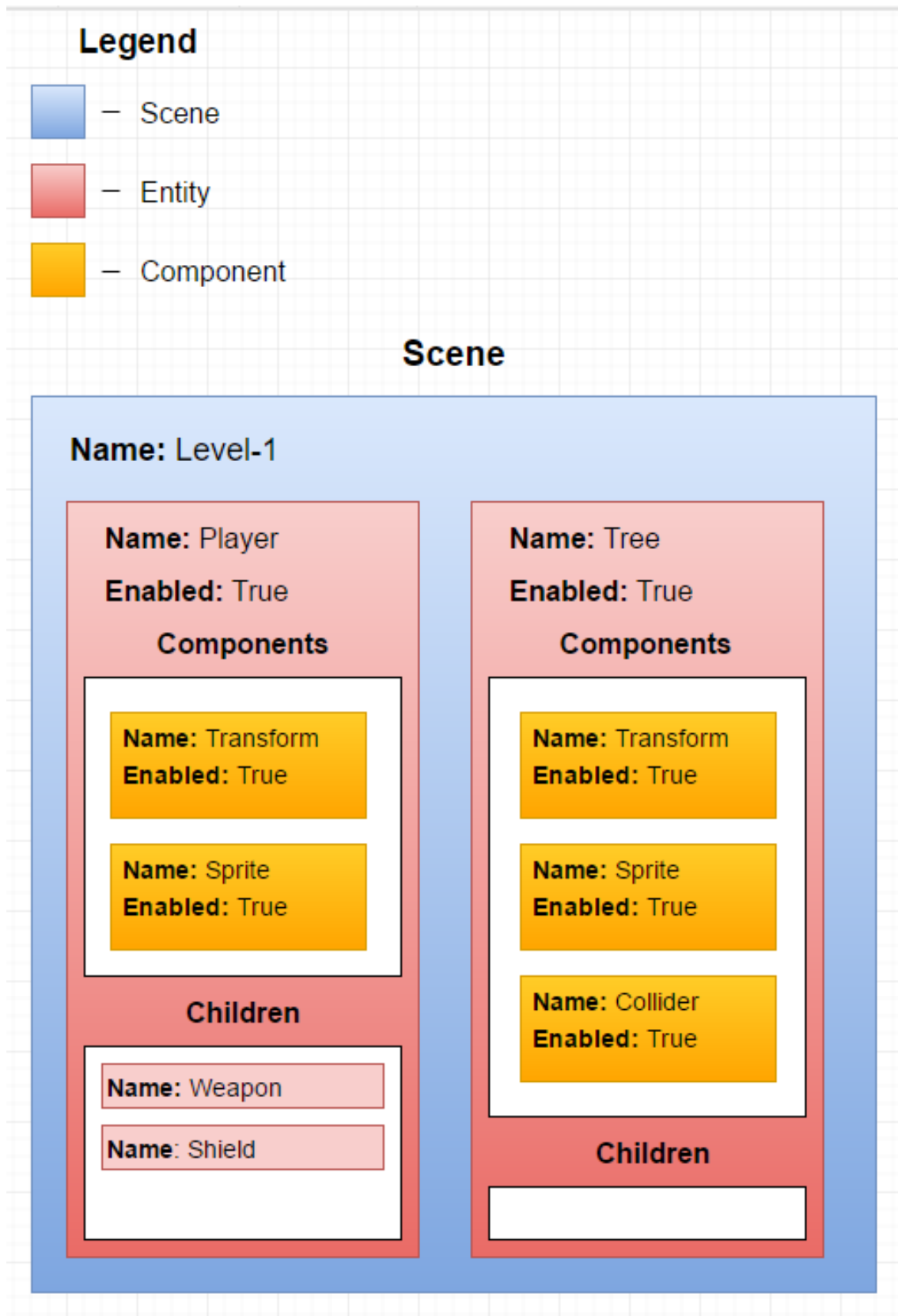


Figure 4.2: The entity-component system architecture used in the game engine.

Each component attached to an entity will be an instance of a component, whether it is a component included in the game engine or a component that the user has created themselves. Each component will therefore be able to contain numerous properties and methods. The most effective way to allow users to add their own functionality within a component would be to provide a base 'Component' class that all components would have to inherit from. This class would contain a few abstract methods - the most important being the 'Update' method, which the user could override to add custom functionality. On every frame in the game, the engine will execute the 'Update' method of every enabled component within the current scene. By doing this, the only task a user must perform is to add code to the method that will be executed on every frame - for example, this could be code to move an entity by a certain amount if a specific key is pressed. The engine would also run a number of other methods on every frame, such as the 'Draw' method (described in detail in section 4.1.4). There will also be certain methods that the engine will execute under the correct conditions. For example, the 'OnCollision' method (described in detail in section 4.1.5).

The main game loop that is iterated over as the game runs can be seen in figure 4.3.

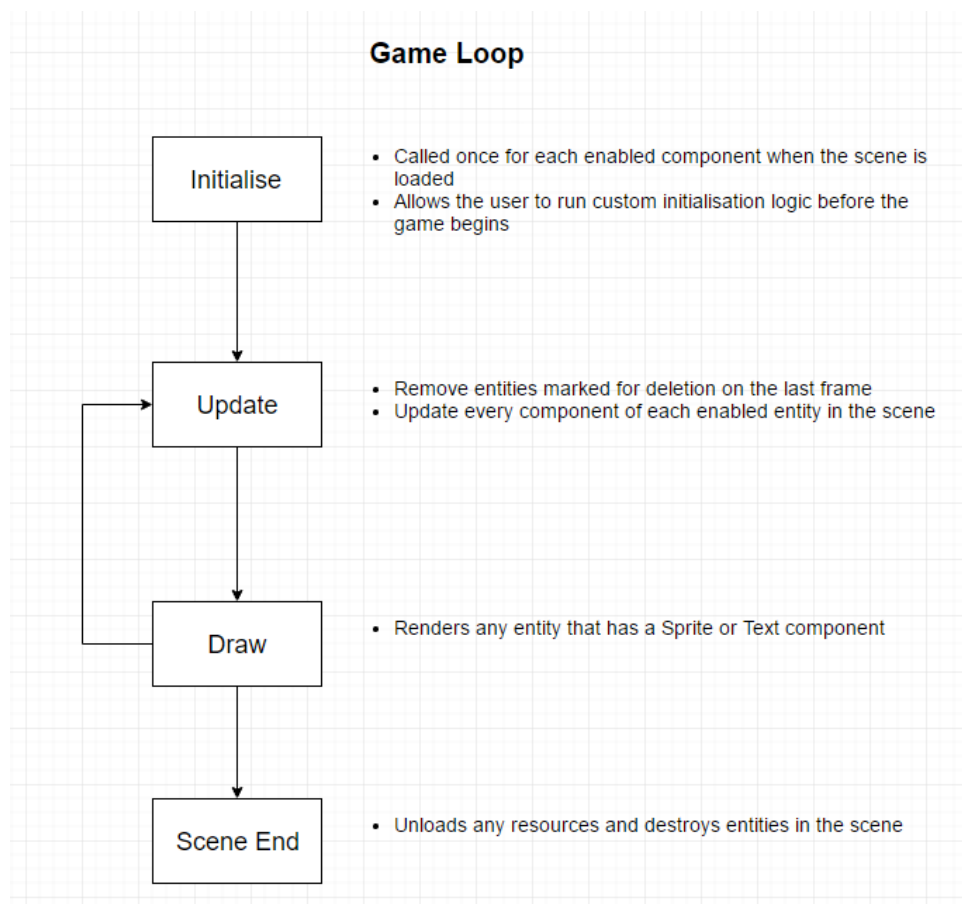


Figure 4.3: The main game loop for a scene.

4.1.2 Scene Loading

When a user runs their game, the game engine must take a file as an input that contains all of the data required for the game to run. To be able to output these files using the editor and use them as an input for the engine, a common file format was required. I first thought about creating a custom binary format that the editor would serialise objects into that the engine could then deserialise. After some initial research, this method proved to be a large amount of work for a feature that is less important in comparison to other features and would not provide any useful advantages in my scenario. Due to this, I decided to store the data for each scene within an XML file. Each file would represent a scene and would contain nodes describing the entities in the scene, their components and the properties of each component. The engine will then be able to read a scene file as an input, parse the XML data and create the corresponding objects within the game engine. The engine should also be able to accept a 'build settings' XML file that describe which scenes should be included in the build of a game if a user wants to use multiple scenes within their game.

4.1.3 Resource Management

The loading of resources in the game engine is a unique task because it involves external data. Up until now, data in a game has been restricted to the values of a components properties. This will not be sufficient for the system because utilising external resources is an extremely common task in game development. For example, a user may want to use an image saved on their computer as a texture for an entity with a Sprite component.

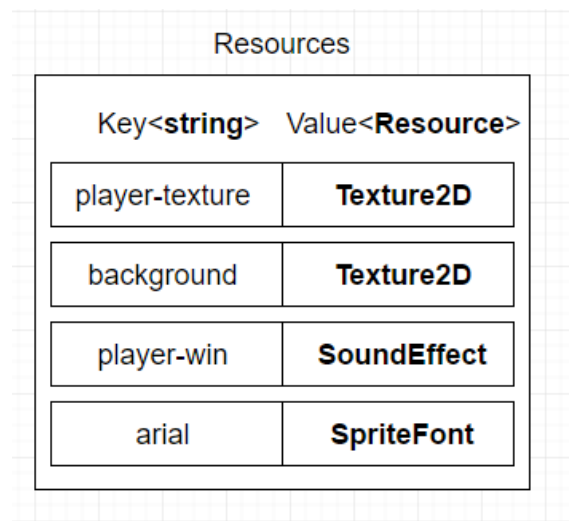
The initial idea I had for loading resources within the engine was to simply include a file path to the resource that the user wants to use within the scene XML file. For example, in the XML file a sprite component could have a 'path' attribute that pointed towards the location of the texture on the users local machine. When the engine is required to use the resource, it would then load it using the file path included in the scene XML file.

I later decided that I had to take a different approach because there were several problems with this method. Firstly, including absolute file paths in the XML file would cause the game to break if it could not find the resource file. For example, if the user had previously created a game and wanted to move the project to another computer, all of the absolute file paths would still prefer to the file system of the first computer and the engine would not be able to find the resources. Secondly, the engine loading the resource whenever it is required would be terribly inefficient. As an example, when rendering a sprite, the engine would attempt to load the texture every time the sprite should be rendered. This would cause the engine to read from a file on the local machine potentially hundreds of times every frame, which would dramatically reduce performance.

The approach I decided to take was to include any resources in a folder relative to the project directory. This would allow the game to refer to a resource using a relative file path, which would ensure that the game would function even if the project was moved to a different location. Specifically, project resources will be located in a directory named 'Resources' which is within the project directory. Users would be able to create more directories within the Resources directory to organise

their resources, but any resource that is used would have to be placed within the Resources directory or any of its subdirectories.

To avoid a negative performance hit, I decided that the game engine should load all of the resources used within the game into memory during the initialisation of the game. This would mean that any resource that may be required would be available at all times while the game is running. To do this, I decided that a key-value pair store such as a dictionary would be the most appropriate data structure to contain the resources. Each item in the dictionary would contain the relative file path to a resource (the dictionary key) and the actual resource data (the dictionary value). A depiction of this can be seen in figure 4.4. Using a dictionary would allow the user to easily obtain a reference to a resource by supplying the relative file path.



Key<string>	Value<Resource>
player-texture	Texture2D
background	Texture2D
player-win	SoundEffect
arial	SpriteFont

Figure 4.4: The dictionary containing resources included in a project.

4.1.4 Sprite & Font Rendering

Designing the sprite and text rendering system within the engine proved to be more complicated than previously anticipated, mostly due to the entity-component architecture of the system. A simple solution would be to allow users to override a 'Draw' method on a component to implement their own rendering logic, but I thought that this put too much responsibility on the user and would reduce the simplicity of the system.

In order to ensure that rendering was handled quietly in the background, I instead opted to make the engine find references to each component that is 'drawable' (i.e. a Sprite or Text component). Although this prevents the user from implementing any custom drawable components of their own, it simplifies rendering greatly. Each Sprite or Text component will have a 'Draw' method that renders the entity using the settings of the component. The user will be able to simply adjust the values of properties on the Sprite component to determine how the sprite is rendered (e.g. the rotation of the sprite).

4.1.5 Collision detection and reaction

Collision within the game is another feature that was difficult to design in the context of an entity-component architecture, due to the fact that not every entity will necessarily need to trigger collision events. The first idea I had was to simply cause every entity in the game to trigger a collision event whenever they collide with another entity. Although this would simplify collision, I decided that it would be much too inefficient in a game that potentially has hundreds or thousands of entities.

I decided that the best approach would be to create a 'Collider' component and only trigger collision events if an entity contains the component. This would allow users to decide which exactly components should raise events when a collision occurs, allowing them to optimise their game to a finer level. On each frame, the engine would retrieve every entity that has a Collider component and check if a collision has occurred between two entities. If a collision has occurred, the engine will call the 'OnCollision' method of each component. This allows the user to react to a collision by simply overriding the 'OnCollision' method in the desired component. Note that the component overriding the method would have to belong to an entity that also has a Collider component, otherwise the method would never be called.

4.2 Game Editor

I decided to use the WPF framework to create the game editor due to its reliability, available tools and its compatibility with the C# language. WPF also supports binding elements of the user interface to variables in code, automatically updating the information displayed in the UI when it is changed in code. This functionality will be heavily utilised in the application due to the amount of data that is constantly changing and the importance of the information displayed being accurate and up-to-date. I also decided that a strong architectural pattern should be used when developing the editor due to the scope of the project - if it is not built with a strong framework, it could easily become extremely difficult to manage as the application becomes more complex. The pattern that I decided to use is Model-View-ViewModel (MVVM). This pattern is used to separate the presentation of an application from its business logic by severing any dependencies between the two. This pattern is described in more detail in section 5.2 of this report.

It should also be noted that due to the fact that the aim of this project is to implement the underlying system allowing games to be created, the user interface design of the editor was kept as a low-priority task. I decided to stick to certain conventions when designing the user interface to ensure that it is easy to use, but the interface was kept as minimalistic as possible, only containing controls that allow the user to perform the required actions. Due to the fact that the user interface design was kept as basic as possible, it will not be discussed in too much detail. Despite this, I created a basic mockup of the editor user interface (as seen in figure 4.5) to ensure that it would be easy to use and that the information required by the user could be displayed to them in an acceptable manner.

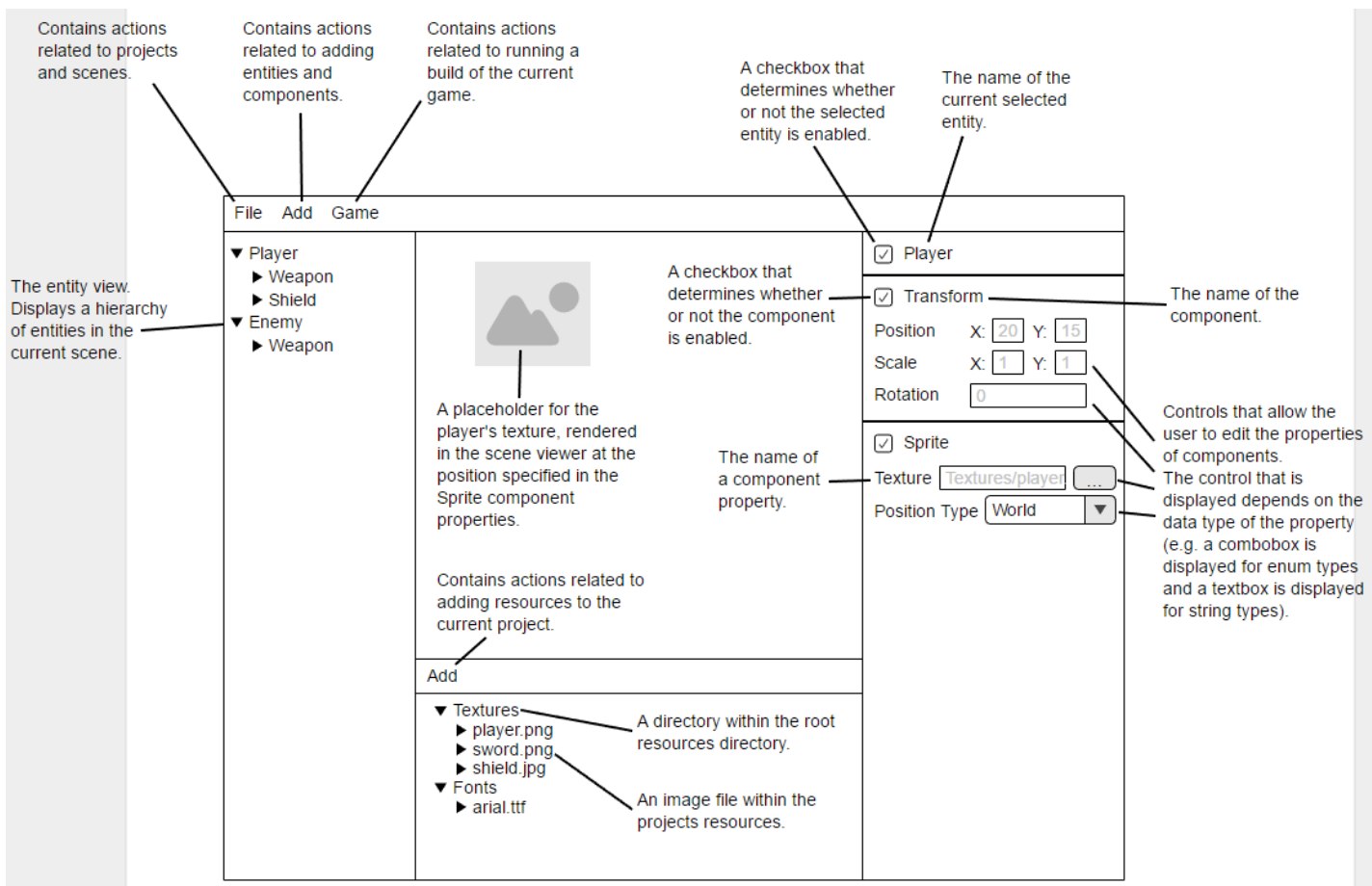


Figure 4.5: A mockup of the game editor user interface.

The following sections outline the design of each core feature of the game editor.

4.2.1 Creating, Opening & Saving Projects

To allow the user to create, save and open existing projects I had to decide on a file format to use to represent the project and scenes within the project, as previously discussed in section 4.1.2. I decided that XML would be the most convenient format to use.

By using the MVVM design pattern to create the editor, I can associate a view model with the game editors window. This ViewModel is a class that will contain properties and methods that the window can access - for example, clicking a menu item could run a method in the view model associated with the window. This will allow me to easily implement the actions that take place as a result of user input. The view model associated with the window will contain data regarding the currently open project and scene, and this data will be serialised to XML when it is saved and deserialised when it is loaded by the editor.

Like most Windows applications, the game editor will have a main menu at the top of the window. This menu will contain several menu items that the user can click to perform different actions in relation to projects and scenes, such as:

- **File -> New -> New Project** - Creates a new project.

- **File -> New -> New Scene** - Creates a new scene.
- **File -> Open -> Open Project** - Opens a file dialog that allows the user to specify the XML file of an existing project and loads the specified project.
- **File -> Open -> Open Scene** - Opens a file dialog that allows the user to specify the XML file of an existing scene and loads the specified scene.
- **File -> Save Project** - Saves the current state of the currently open project if there are any unsaved changes.
- **File -> Save Scene** - Saves the current state of the currently open scene if there are any unsaved changes. If the scene has not been saved before, a file dialog is opened allowing the user to specify the name and the destination directory of the scene.
- **File -> Save Scene As** - Opens a file dialog allowing the user to specify the name and the destination directory of the scene.
- **File -> Close Project** - Closes the currently open project and the currently open scene.
- **File -> Close Scene** - Closes the currently open scene.

If the user attempts any of these actions they will be prompted to save any unsaved changes in the currently open scene or project, if necessary. For example, this will prevent a user from losing work in an existing project if they selected the 'New Project' menu item.

4.2.2 Entity View

I decided that the entity view should display a hierarchy of the entities in the current scene, using each entity's name as an identifier. Displaying only the names of the entities in this view prevents the view from becoming too cluttered. Instead, the entity that is currently selected in the view will determine the information that is displayed in other views, such as the entity properties view (see section 4.2.3).

4.2.3 Entity Properties View

The entity properties view will display extra details about the selected entity, such as its name and whether or not it is currently enabled. It will also display a list of the components attached to the selected entity, along with details about each component and controls that allow the user to edit the properties of the component.

In a previous design of the entity properties view, I used a text box control to edit any type of property. This was problematic because each value would have to be parsed - for example, if the user wanted to set an enum value they would have to know all of the possible values in advance and would have to make sure they entered the enum name exactly right. If not, the engine would not be able to parse the value. To maximise usability, I decided that the control that is displayed should be dependent on the data type of the underlying property. By doing this I will be able to display custom controls that are appropriate for the data type. For example,

if the property is a texture, a textbox and a button will be displayed. Clicking the button will open a Windows file dialog that allows the user to browse their files and select the texture they want instead of typing the relative file path manually.

4.2.4 Resources View

The resources view will display the resources associated with the current project, allowing the user to check what resources are available to them at any time. The subdirectories and files within the root directory are displayed in a tree view, allowing users to navigate through subdirectories and view all of the files in the project. Due to the fact that all of the resources used in a given project are located in the same 'Resources' directory relative to the project directory, this view can be populated with directory and file names by recursively searching the root directory and adding an entry to the tree view for each resource.

The resources view will have a menu with actions that allow the user to add existing files to their project. Selecting the menu option will open a Windows file dialog, allowing the user to select an existing resource on their computer. This resource will then be copied to the projects 'Resources' directory to ensure that it is always available.

4.2.5 Scene Viewer

To allow the user to see a visual representation of their game whilst they are building it, the scene viewer will render the sprites and text of any entities within the current scene. To do this, I will make use of the binding functionality available in WPF. Each component in the scene will be bound to a corresponding representation of the component in a view model. This view model will contain the current values of all of the components properties which the user interface elements will be bound to. This ensures that when a user modifies a component property it is correctly updated in the view model.

Using the WPF binding functionality also has advantages when it comes to the scene viewer. Due to the fact that the different properties on the view model will always correspond to the latest value of the property, the entities within the scene viewer can simply be bound to the same values. This means that each entity being rendered in the scene viewer will have access to the position, scale and rotation of the entitys transform component, so that the scene viewer will be updated whenever a value is changed. For example, if the user changes the rotation of a sprite using the controls in the entity properties view, the visual representation of the entity will be rotated accordingly.

4.2.6 Running A Game

As a way for the user to test the game, the user should be able to build the game and run it through the editor. To do this, the main menu of the editor will contain a 'Run' menu option that simply launches the game executable. The user will then be able to interact with the game (e.g. for testing purposes) and close it once they are finished to return to the editor.

5. Implementation

During the implementation of the project I was able to adhere to the original design, although in some areas I encountered a number of unforeseen problems. In the following sections of the report I will discuss the implementation details of the core features of the game engine and game editor. I will also discuss the various problems that I encountered and how I managed to overcome them.

5.1 Game Engine

5.1.1 Entity-component system

The implementation of the entity-component system in the engine relies on several core classes:

- **Scene**
- **Entity**
- **Component**

The **Scene** class is used as the root of a game and contains a list of entities. Users have the ability to switch between scenes by using the **SceneManager** class, which handles loading and switching between different scenes in a game.

The **Entity** class is a nothing more than a container for components with a name and an 'enabled' status, due to the fact that all of the functionality of an entity should be provided by its components. This means that every entity in the game is an instance of **Entity**, and the class should not ever be inherited from. I decided to use a **Dictionary<Type, Component>** to store the components for a number of reasons. Firstly, it means that an entity can only ever have one component of each type - there is no reason for an entity to have more than one of the same component, so this is desired behaviour. As a rule of thumb, if a user ever felt like they wanted to add a second component of the same type to an entity, they should generally just create a second entity that has the same component. Secondly, it meant that I could make use of generic methods when providing an interface for users to use when adding and removing components. For example, consider the implementation of the **Entity.GetComponent<T>** shown in figure 5.1. By using a **Type** as the key in the dictionary, the user will be able to specify the type of the component that they want to find and the method will return the component if it exists. If the method instead allowed a user to find a component by specifying its name, the name the user provides would have to match the component exactly and if they misspelled it they could face errors that would be difficult to debug. By using a generic method, if the user misspelled the type name they would receive a compiler error which is much more useful.

```

public T GetComponent<T>() where T : Component
{
    bool componentExists = Components.TryGetValue(typeof(T), out Component component);
    if (componentExists)
        return (T)component;

    return null;
}

```

Figure 5.1: Code for the GetComponent method of the Entity class.

The **Component** class is an abstract base class that all components must inherit from to be processed by the engine. I decided that this would be the best approach because it allowed me to define values that every component in a game should have, such as a name and an **Update** method.

5.1.2 Scene Loading

The scene loading functionality in the engine is comprised of a number of steps. Firstly, the user provides a path to the scene XML file. The file is then deserialised into a **XmlScene** object containing all of its entities, components and component properties. The deserialisation is performed using the serialisation functions included in the .NET framework. Secondly, the scene must be mapped from an **XmlScene** object to a **Scene** object. An **XmlScene** is a data model for the scene that contains all of the scene data in a textual format - this must be done because the data comes from an XML file, which is simply a text file. Alongside the **XmlScene** class, there are also the **XmlEntity**, **XmlComponent** and **XmlProperty** classes. This step was more complicated as I originally anticipated due to the fact that the component properties can be a number of different data types. Due to the fact that an XML file can only contain plain text, a property is represented in XML by specifying the name of the property, a string interpretation of the value and the fully qualified name of its data type. For example, consider the following XML representation of an enum property:

```

<Property Type="XmlEnumProperty" Name="Effects" Value="None" TypeFull-
Name="Microsoft.Xna.Framework.Graphics.SpriteEffects" />

```

Each of the XML attributes are simply strings, but the engine needs to be able to store the value as its actual type (in this case, a 'SpriteEffects' enum). To achieve this, the mapping classes in the engine take advantage of reflection to retrieve a type based on its name. This allowed me to convert the XML representation of a property to an object with the corresponding type.

After the serialisation and mapping processes have finished the engine will have representations of each scene, entity and component in the game. Each component within the game can then be updated and rendered on each frame.

5.1.3 Resource Management

To maximise usability, I decided to use a static class **Resources** to store the resources of the current scene. Although I had originally planned to store all of the resources within a project in a dictionary, this proved to be problematic due to the fact that several different types of resources need to be available to users. Instead, the class has several static key-value pair stores for different types of resources (such as **Dictionary<string, Texture2D>**). These dictionaries use a relative path to the resource as a key which will guarantee that the key for each resource is always unique. The value of the dictionary is the actual resource, such as a **Texture2D** or a **SpriteFont**. Using this approach, the user is able to access resources at runtime by simply querying the static **Resources** class and providing a relative file path to the resource that they want to retrieve.

During the initialisation of a scene, each resource is loaded into the correct dictionaries using the following process:

1. The 'Resources' folder of the project is recursively searched, and any file supported by the engine (e.g. JPG or PNG) is loaded into memory and stored in the appropriate dictionary.
2. The scene XML file is searched to find every instance where a resource is requested (for example, when a Sprite component references a Texture) and the corresponding component is provided with the resource in the dictionary with a matching key.

This allows users to specify resources using the editor as well as specifying them during runtime.

5.1.4 Sprite & Font Rendering

Sprite and font rendering is handled in the engine using the following process:

1. On every frame, the engine retrieves a collection of all of the Sprite and Text components in the current scene.
2. The components are then separated by their **PositionType**, which determines whether the sprites should be rendered relative to the world position or the screen position. Using world position means that entities will move when the camera moves, whereas using screen position renders entities at a fixed position that does not change when the camera moves.
3. Each entity then ensures that their Sprite or Text component is not null, calls the **Render** method on the entity and then calls the method on each of the entities children.
4. The entity is then rendered using the settings provided by the Sprite or Text component, which can be modified by the user to adjust how an entity is rendered.

5.1.5 Collision detection and reaction

Collision detection in the engine was rendered similarly to the original design. On every frame, the engine iterates over every entity that has a **Collider** component and checks if it is colliding with any other entity with a **Collider** component. The collision is detected by using simple AABB collision detection, which checks if two rectangular colliders are intersecting.

If a collision is detected, a number of different methods may be called on every component attached to the entity with the collider:

- **OnCollisionEnter** - Called when the collision begins (i.e. if the entities were not colliding on the previous frame).
- **OnCollisionExit** - Called when the collision ends (i.e. if the entities were colliding on the previous frame but are not currently colliding).
- **OnCollisionStay** - Called every frame whilst the collision is still occurring.

Users can override each of these methods in any of their custom components. If a user overrides one of these methods, they will be called whenever a collision occurs on the entity the component is associated with. Each of the collision methods are also passed a 'CollisionData' which contains a reference to the entity that was collided with, the direction of the collision and the a **Rectangle** object which represents the intersection of the collision. This allows the user to implement custom collision logic. For example, the snippet of code in figure 5.2 will reduce an entity's health by 10 if it is collided with.

```
public override void OnCollisionEnter(CollisionData collData)
{
    var health = collData.Entity.GetComponent<Health>();
    health.Reduce(10);
}
```

Figure 5.2: An example of using the OnCollisionEnter method.

5.2 Game Editor

5.2.1 Creating, Opening & Saving Projects

I implemented the ability to create, save and open projects and scenes by taking advantage of the binding capabilities in WPF. The entire editor window is bound to a view model, **MainWindowViewModel**. This view model contains a **ProjectViewModel**, which in turn has a property 'CurrentScene', which is a **SceneViewModel**. Each view model in the system inherits from a base class **ViewModel** which defines functionality that allows the user interface to be automatically updated whenever the value of a property on a view model changes. This meant that to create, open and save projects and scenes all I had to do was modify the value of its corresponding view model. For example, to close a scene I simply set the value of the **CurrentScene** view model to null. Opening a project or scene is as simple as choosing the menu option, using the file dialog to choose the appropriate XML file and the editor will then deserialize the file and map it to a set of view models.

5.2.2 Entity View

The entity view was implemented by binding the `TreeView` control that displays the entities to the list of entities on the `CurrentScene` view model. This meant that whenever an entity was added or removed from the scene, the tree view would update accordingly.

To allow a user to add entities to the scene I added a context menu that appears when a user right-clicks on the root of the tree view or an entity within the tree view. The user can choose the 'Add Entity' menu option, enter a name for their entity and it will then be added to the scene. If the user clicked the root of the scene the entity will be added to the root, but if the user clicked an entity, the new entity would be added as a child of the entity that was clicked.

I also added a second menu option 'Add Component' that only appears when an entity is right-clicked. Selecting this option would open a custom window that allows the user to select a component to be added to the selected entity. Implementing this window was more complex than originally anticipated because the available components depend on the actual code files that the user has added to their game (because the editor would need to allow the user to add components they have created themselves). I achieved this by using reflection to access the assembly of the users project and find every defined type that derives from the `Component` type. These components are then mapped to their corresponding view models representations so that they can be displayed to the user. The user is then able to select a component, which adds the component to the selected entity.

5.2.3 Entity Properties View

The entity properties view turned out to be one of the most difficult parts of the system to implement, due to its requirement of displaying a different, custom control depending on the data type of each component property.

I achieved this by using numerous different resources that were available to me. Firstly, I took advantage of WPFs 'data templates'. Data templates allow you to specify markup in XAML (the markup language used to design user interfaces in WPF) that can be conditionally displayed depending on the type of a bound property. By using data templates I was able to design a custom control for all of the data types that the system supports and then display a different type based on the data type of the corresponding component property.

Although this feature took a lot of time to implement, I believe that it is one of the largest contributors to the simplicity of the system. Allowing users to modify properties with different data types by using a control that is designed specifically for that data type speeds up the development process greatly.

5.2.4 Resources View

I implemented the resources view in the same way as described in the design section - fortunately, this feature was fairly straightforward to implement and there were no unforeseen problems. The only aspect that was changed was the user interface. Instead of displaying a single tree view that would display folders and files, I decided to split the resources view into two parts - a tree view and a list view. The tree view would display each folder within the 'Resources' folder and the list view would

display the files within the folder that is currently selected in the tree view. This made it easier to specifically find files, and also allowed me to add extra information about each file (such as their file extension).

When a project is loaded, the editor recursively searches the root 'Resources' folder to find all of its subdirectories. Each subdirectory is then searched for files and the files are stored in a view model that the list view is bound to. This ensures that whenever the user selects a different folder in the tree view, the information in the list view is also updated to reflect the new selection.

5.2.5 Scene Viewer

The scene viewer was another feature of the editor was more complex than anticipated. I decided to use MonoGame in order to render sprites and text within the scene viewer, but MonoGame instances are not natively supported in WPF applications so I had to find a way to make it work.

After some online research I found a freely C# library, 'WpfInterop'[15], that allows MonoGame to run within WPF applications. Due to a lack of high quality documentation and no other suitable alternatives, getting a MonoGame instance running within the editor took longer to implement than expected. After getting the library to work correctly within my application I encountered another unforeseen problem related to compiling project resources into XNB files, which is described in more detail in section 5.3.1.

The scene viewer works similarly to how the game engine works, but instead of using data from an XML file to run the game, the scene viewer binds to the view models for each entity in the scene. This allowed me to create a dynamic scene viewer that updates as a components properties are changed. For example, if a scene contains an entity with a Sprite component, the sprite will be drawn in the scene viewer at the position specified in the entitys Transform component. If the user updates the position using the appropriate control in the editor, the sprite will immediately update its position accordingly.

This feature also greatly increases the usability of the system because it allows users to easily see a preview of what their game will look like, allowing them to perform tasks involving the positioning, scaling and rotating of sprites and text with ease.

5.2.6 Running A Game

To allow the users to test the game I added a 'Run' menu option to the main menu. When this is clicked, the editor builds the game content, saves the current scene and launches the build executable as a separate process. Launching the game as a separate process was advantageous because it allowed me to reroute any errors thrown in the executable to an alternative stream, such as a file. This helped massively when debugging the editor.

5.3 Unforeseen Problems

5.3.1 Compiling Project Resources

During the design stage of the project I had the understanding that MonoGame was capable of loading resource files (such as PNG files) directly. Unfortunately, this was wrong - MonoGame loads resources in the XNB format, which is a custom binary format used to compress resources and obfuscate their content. This meant that I was unable to load any type of resource into the game engine using the method described in the design section.

To overcome this, I had to find out how I could compile all of the resource files in the game to XNB files which could be understood by the engine. I discovered that MonoGame provides a separate tool, 'MonoGame Pipeline'[13], that is used to compile resources manually. To use this tool I would have to expect each user to learn how to use the tool and compile their resources whenever they add a new resource to the game. This would not be sufficient. Instead, I decided to make use of the 'MonoGame Content Builder'[14], a command-line tool provided by MonoGame that allows resources to be compiled to XNB files by providing the tool with a text file consisting of commands that are understood by the tool. Due to a lack of useful documentation, figuring out how the tool works was a time-consuming process.

I decided that to properly utilise this tool I would have create two helper tools:

1. A tool that can generate a valid file that describes what resources should be compiled.
2. A tool that runs the external MonoGame Content Builder tool and provides it with the generated file as input.

I discovered that the MonoGame Pipeline GUI tool actually uses that MonoGame Content Builder tool under the hood, so I started by compiling resources using the GUI tool and to see what code was generated. I analysed this code to determine how the input file to the command line tool should be formatted and wrote a tool that parses the contents of the projects resources directory (and subdirectories) and generates an input file based on the files available in the project resources.

After this, I wrote a tool that launches the MonoGame Content Builder as a separate process and provides it with the previously generated file as input. I was able to overcome the problem by generating a file and running the tool every time the game runs, which ensures that up-to-date XNB files are always available to be used.

6. Results & Evaluation

Overall, I believe that the project was a success. Although I was not able to meet reach every goal that I had originally set at the beginning of the project (this is described in detail in section 7), the majority of the functionality works correctly and the system achieves its goal of providing the ability to create 2D video games with ease. A user is able to follow the whole process of creating a project, adding entities, creating custom components, adding components to entities and modifying the properties of components using the editor and testing their game through the editor.

I originally planned to include an example game with the project to demonstrate the capabilities of the system, but unfortunately I was unable to finish this due to the implementation taking longer than anticipated. To test the system, I created multiple projects using the editor and attempted to use each function in the editor (e.g. adding entities, setting component property values etc.). For each project, I attached the game executable process to Visual Studio so that I could detect any errors that were thrown. By the end of the project, I was able to test every aspect of the current state of the project without encountering any errors.

I believe that the current state of the project proves that a game engine specifically designed to create 2D games using an entity-component system would be an extremely helpful tool to game developers, although in this projects infancy it is not powerful enough to be considered a serious game development tool and is more like a prototype. I believe that with the chance to extend the system and add missing features that are necessary for serious game development, this tool could fill a gap in the game development software market.

7. Future Work

Although the project achieves its main goals, there are a number of features that I had hoped I could implement in the system but was unable to due to time constraints. The following sections describe the different features that I would implement if given the opportunity in the future.

7.1 Build Settings

In its current state, the game engine is only capable of running a single scene at any time. Although this is acceptable for small games and for testing, large games with many scenes would require the ability to fine-tune the build settings of the game.

Build settings would allow the user to specify what scenes they want to be included in a build of the game, what platform to build the executable among many other settings.

7.2 Sprite Animation

Although I had originally planned to include a component that handles sprite animation in the engine, time constraints meant that I was unable to provide this. If I had the chance, I would include a component that allows users to create a sprite animation by selecting multiple textures that would be used for each frame of the animation. The component would also allow the user to specify other settings related to the animation, such as the length of the animation and whether or not it should loop on completion.

7.3 Sound Effects

I would also implement a component that handles sound effects and background music within a game. Users would be able to add this component to an entity, specify an audio resource and specify settings such as the volume of the sound, whether or not the sound loops on completion and the tempo of the sound. I would also add an 'Audio Listener' component. By doing this, the user would be able to specifically define entities that are able to hear audio, allowing functionality to be added to handle things such as the sound from an audio source becoming quieter as the player moves further away from it.

7.4 Scene Viewer Improvements

One of the main features that I was unable to implement in the given timeframe is scene viewer handles. Scene viewer handles would be graphical tools displayed on the editor that the user could interact with to manipulate entities. For example, a position handle would allow the user to click and drag the handle to move an entity within a scene. This would be much more user friendly than editing position coordinates manually and would provide a much better user experience when creating games using the editor.

7.5 Unit Testing

While it is not specifically a feature of the system, if I was to work on the system in the future I would make extensive use of unit testing during the development stage. Writing and running unit tests would allow me to test individual modules of the system programmatically, speeding up the testing stage and providing accurate feedback.

8. Conclusions

In conclusion, I believe that the project was a success because it achieves the original aim of the project, but its functionality is not extensive enough to be considered a serious game development tool. The project proves that utilising an entity-component system can improve the simplicity of a game development platform and allows for fast progress during the development stage.. The project also shows that using a game engine designed specifically for creating 2D games can simplify the process because it hides a lot of unnecessary functionality that is only necessary when developing 3D games.

9. Reflection

Prior to my work on this project, I had not been involved in any long-term software development projects with such a large scope. I also had very little experience with the non-technical aspects of software development, such as identifying requirements, designing a system and evaluating what has been done.

This project has helped me to realise how much work is really involved in software development. Being inexperienced in the area, it was easy to overestimate the amount of work that could be accomplished within the timeframe of the project, which made it difficult to accomplish all of the goals of the project on time. The project has given me insight into what real development is truly like and I feel like the experience I've gained will be invaluable when working in software development in the future.

I also realised how much longer a project can take due to unforeseen problems that are mostly out of your control - for example, the reliance on 3rd party software packages that are difficult to use, lack documentation or are not supported on certain platforms. When working on projects in the future I will have to ensure that any 3rd party tools or frameworks that may be required are identified early and I will set aside time to ensure they can be used effectively before using them in a project.

Although I encountered many more issues than I had originally thought I would encounter, I am happy with the work that I have done and the final product that I have produced. I believe that the experience working on this project will help me many times in the future and has helped to prepare me to work on large software development projects in the future.

Bibliography

- [1] Wikipedia.org *XNA* https://en.wikipedia.org/wiki/Microsoft_XNA
- [2] Monogame.net *MonoGame* <http://www.monogame.net/>.
- [3] Wikipedia.org *Windows Presentation Foundation*
https://en.wikipedia.org/wiki/Windows_Presentation_Foundation
- [4] Microsoft.com *What is XAML?* <https://msdn.microsoft.com/en-us/library/cc295302.aspx>
- [5] Gameprogrammingpatterns.com *Component - Decoupling Patterns*
<http://gameprogrammingpatterns.com/component.html>
- [6] Unity3d.com *Unity - Game Engine* <https://unity3d.com/>
- [7] Yoyogames.com *GameMaker* <http://www.yoyogames.com/gamemaker>
- [8] Yoyogames.com *GML Overview* https://docs.yoyogames.com/source/dadiospice/002_reference/002_reference.html
- [9] Wikipedia.org *Breakout (video game)* [https://en.wikipedia.org/wiki/Breakout_\(video_game\)](https://en.wikipedia.org/wiki/Breakout_(video_game))
- [10] Microsoft.com *Windows* <https://www.microsoft.com/en-gb/windows>
- [11] Wikipedia.org *C#* [https://en.wikipedia.org/wiki/C_Sharp_\(programming_language\)](https://en.wikipedia.org/wiki/C_Sharp_(programming_language))
- [12] Wikipedia.org *.NET Framework* https://en.wikipedia.org/wiki/.NET_Framework
- [13] Monogame.net *Pipeline* <http://www.monogame.net/documentation/?page=Pipeline>
- [14] Monogame.net *MGCB* <http://www.monogame.net/documentation/?page=MGCB>
- [15] Github.com *MonoGame.Framework.WpfInterop*
<https://github.com/MarcStan/MonoGame.Framework.WpfInterop>