Component Based Game Engine

Final Report

Submitted for the BSc in
Computer Science with Games Development

May 2018

by

Henry Vincent

Word Count: 8666

Table of Contents

Table of Figures

1 Introduction

A game engine sits as the foundation of almost any modern game, providing the systems and the overall framework by which the game will run, including physics, graphics rendering, input management and more. In effect, a game engine provides the tools for developers to make a game, and the framework for the game to execute on either specified or unspecified hardware. It would be incorrect to state that the engine is, in itself, a complete game; it is only part of a game – albeit an essential one (Thorn, 2011).

One modern architecture for a game engine is commonly known as the 'System-Entity' architecture. This architecture ditches the standard inheritance structure for one of aggregation. This architecture involves 'logic objects' called 'entities' which aggregate components (Gestwicki, 2012). Components simply hold data - for example, a component could be created in order to hold 3D position information of an entity. This is advantageous because it solves problems where inheritance causes most of the code for an entity to exist in the top level of the software structure. In a large and complex game, this can cause messy and disorganized code and therefore become very difficult to manage (Gestwicki, 2012). The system-entity architecture is sometimes referred to as 'Component-Based'.

The purpose of this project is to create a component-based game engine. Because of time constraints, it is unrealistic to attempt to create a complete game engine; however, some of the functionality can be completed using third party libraries, so that focus can be put on other a few specific managers. This engine will be used to make more simplistic games, as anything too complicated would likely need an engine more properly optimized for performance. Therefore, this engine has been developed for more 'light-weight' games, which exclusively use 2D graphics.

This report is to serve as a final breakdown of this project. In the 'Aims and Objectives' section, the purpose of this project will be further examined, as well as the criteria for which the engine's features have been determined. These objectives were specified at the start of the project's development so therefore shall not be changed here. The purpose of including them in this report is to establish a level of success from the finished project when compared to the original objectives.

The second section, 'Background', is included for the purposes of investigating the established technologies and practices in the area in computer science of which this project resides. An analysis of other game engines and game technologies will be presented, focusing on architecture and design patterns, with a focus on the different approaches from both mainstream and small or independent engines. Also, a further investigation into the concept of the system-entity architecture previously discussed in this introduction. Furthermore, 'middleware' is an important part of any game engine and will be examined in detail within this section.

Discussions into the actual development of this project will commence with section 4, 'Technical Developments'. Here, a breakdown of the design and implementation of the technologies of this game engine will be performed, with the purpose of explaining how this

engine functions. This will be achieved using UML diagrams and descriptions of the algorithms used in this project. Testing is also an important aspect to be discussed here.

Finally, the ‘Evaluation’ will be used to discuss the achievements and pitfalls of this project. This section is where the aims and objectives will be compared to the result of development, for the purposes of determining the successes and achievements, as well as the areas that still need improvement should the development of this project continue after the fact. Any mistakes during development will also be discussed in this section.

2 Aim and Objectives

The aim of this project is to produce a component-based game engine.

In order to meet this aim, the following objectives have been defined:

Objective 1 – Create a rendering manager

The rendering manager is the part of the engine, which handles drawing graphics onto the screen every frame. This manager will need to operate after all the other managers have done their work for the current frame, and cannot work concurrently. Therefore, the rendering manager should be the first manager developed for this project.

Objective 2 – Create a physics manager

The physics manager is the part of the engine, which handles calculating the physical simulation of entities in the game.

Objective 3 - Create an audio manager

The audio manager is the part of the engine responsible for generating and processing audio within the game. This system should react to events, which trigger audio and play an audio file or generate a tone depending on the wishes of the developer.

Objective 4 - Create an object manager

The object manager creates game entity objects and assigns the relevant components to them, based on scripting the developer has written.

Objective 5 - Create a scene manager

Sections of a game will be split up into scenes, which allows the developer to load in new objects and to change the state of the game. A menu is a kind of scene, as is a section of game play. The two are distinct and require different objects and behaviours. The scene manager is in charge of switching between scenes based on triggers written by the developer.

Objective 6 - Create an input manager

The input manager is responsible for handling all inputs - be it from a keyboard and mouse, joystick, or game controller. It should pass on the information to the other managers so they can react to the inputs.

Objective 7 - Create a game

So that the features of the engine can be demonstrated, it will be necessary to create a game using the engine, which clearly demonstrates all of the features of the engine working correctly.

3 Background

Problem Context

In standard object-oriented design, inheritance and subclasses are used to provide specific behaviours, data and logic for objects within a program. This is not a very practical approach for video games, because of the variety of different objects that are required – leading to an explosion of subclasses and as a result, redundant duplication of logic occurring even in objects which do not require that logic (Stoy, 2006). This approach also introduces the unfortunate consequence of limiting the behaviours that will be inherited to new objects within a game. If a new object is added to an already existing hierarchy which requires inheriting behaviours from objects in various places in the hierarchy, it is often to put them in an appropriate place. This forces the software designer to move most functionality to the top, making the structure very 'top heavy'. Methods need to be put in places of the hierarchy in illogical locations because of where they are in the class tree, rather than because of what they are (Rene, 2005). This can obviously lead to disorganized and messy code that is very difficult to work with.

One solution to the problem of class explosion is the 'Game Object Component System'. This system removes game logic from game objects and moves it into aggregated components, which are assigned to a simple game object (Stoy, 2006). This simple game object is not even necessary, as with this design, the 'game object' only needs to contain a handle for all the components to 'belong to' (Rene, 2005) (West, 2007). It can also be designed so that the game object is a container for objects to reside in, rather than just a simple ID container (West, 2007). An example of this can be seen in Figure 1.

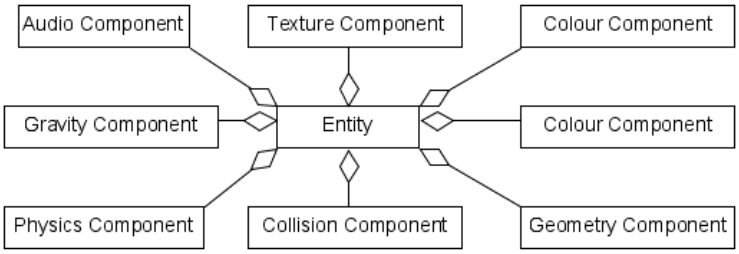


Figure 1 Example of entity component structure

In this architecture, because logic is no longer held within game objects, components are used by 'systems' which implement behaviour based on the data found in components (Gestwicki, 2012). For instance, a texture component simply holds the data required to render a texture. The Render system takes the texture information from the texture component and uses that to render the texture. This approach to game engine design provides various advantages.

One major advantage is in creating modularity. Because systems are independent and functionally distinct from one another, they can be added, removed or exchanged at the requirement of the game developer if more or different functionality is required (Thorn,

2011). This could also be done to keep the game engine up to date with the latest technologies. An aging render system can be easily replaced with a more modern one – even during the development of a game. This is particularly advantageous in terms of saving cost.

This leads into another big advantage of this system – recyclability. Because of the game-object component system's structure, alongside the structure of the system-entity architecture, the game engine can be reused for multiple projects very easily (Thorn, 2011). This saves money and time in development of complicated games. An engine can even be designed so that it can be used across multiple genres of games, for example, the Unity engine (Blackman, 2011).

Because the full scope of the required functionality of a game engine, it is rare that all the logic of the engine will be developed in house. Some game engine functionality, therefore, is achieved using what is known as 'middleware' (Hughes, 2011). Some examples of middleware are Wwise, Scaleform, Bink and Simplygon (Pethrus, 2016). Middleware is often used if a developer does not have the time or resources to code the same functionality to the level required for the game engine, especially considering how advanced some middleware can be. This not only saves time but also reduces risk of failure in developing the engine (Tulip, et al., 2006). This works with the modularity of the game engine – if a game is running on Xbox and uses the DirectX API, it might be necessary to switch the rendering system to OpenGL to get it working on Linux, which does not officially support DirectX (Perez & Royer, 2000).

One essential part of this architecture that has not yet been mentioned is the managers. A standard game engine will feature several managers, which do separate things. A Game Object or Entity Manager manages the creation and destruction of game objects, as well as storing game objects in some sort of database (Rene, 2005) (Gestwicki, 2012) (Andrews, 2009). Other managers include the Resource Manager, which loads and unloads assets such as textures and sound files; the Scene Manager, which loads and unloads the current scene; and the Error Manager, which logs errors and handles crashes gracefully to prevent damage to any data (Thorn, 2011).

Comparison of Technologies and Alternative Solutions

Game Engines

There are many currently available solutions for the problem of a component-based game engine. The following is a comparison of a few of them.

Unity

Unity is a cheaper game engine designed as an entry point for small developers and home users to get into game development. Becoming available on Windows in 2009, Unity is cross-platform, supporting PC, Xbox 360, PS3, Wii and more (Blackman, 2011). Because of its accessibility and ease of use, coupled with a strong level of online support, Unity is extraordinarily popular – with over 34% of games made with this engine (Unity Technologies, 2018).

Unity is a component based game engine which uses the standard system-entity architecture discussed previously (Unity Technologies, 2018) (Xie, 2012). One of its biggest selling points is how it provides an editor for developing games, which means that minimal coding is required to make simple games. This is an attraction for young or inexperienced developers who are more interested in making games than learning to code. It supports 2D and 3D rendering physics, middleware such as Box2D and NVIDIA PhysX, and custom scripting (Unity Technologies, 2018).

A Unity game is built using a collection of scenes, which contain environments and menus of the game. A scene is essentially a separated game level (Unity Technologies, 2017). Scenes can be switched between to form the overall structure of a game. So essentially, a game is comprised of a collection of scenes. This is a standard feature of this kind of architecture (Thorn, 2011).

Another feature of Unity is prefabs. Prefabs are saved game objects which have had components and properties already pre-assigned to them. Prefabs can be instantiated, and the components and properties of instances can be overridden on an individual level (Unity Technologies, 2018). The advantages of this approach are obvious, if many of the same object are required in a game, multiple copies can be created easily, and can be varied at the game designer's whim. This speeds development immeasurably when compared to doing the same just with code which has to be manually copied and pasted can therefore can lead to messy code. In Unity, if the game designer wishes to alter all pre-established instances of a prefab they can simply alter the base prefab (Unity Technologies, 2018).

While Unity is popular with 'low-budget' developers, it does have its downsides, which lead to some looking to other solutions. For one thing, Unity is known for having poorer-than-average graphical capability, as well as an under-featured physics system. People also have complained about the source code not being available, making modifying the engine practically impossible (Wilcox, 2014) (Yeeply, 2014).

Unreal

In June of 1998, Unreal launched on PC. At the time, it was considered one of the best-looking games on the platform. The engine for this game was released to paying customers in July of 2001 (Busby, et al., 2005). The current version of the Unreal Engine is Unreal4 and is, like Unity, component based (Epic Games, n.d.). It also is designed for multiple platforms, like PC, PlayStation 4, Xbox One and more (Epic Games, n.d.).

Unreal engine 4 is freely available for anyone to use, although payment must be made in order for a game, developed in the Unreal engine, to be sold publicly (Epic Games, n.d.). It has the advantages of being considered one of the most graphically superior game engines on the market (Wilcox, 2014), and providing the source code, which allows for modifying of the engine (Epic Games, n.d.). It also features 'blueprints' which allow users to build basic logic without needing to code (Epic Games, n.d.).

While the Unreal Engine does use the game-object component structure, it also features some amount of hierarchy as well. Unreal refers to game objects as 'Actors' which both hold components and can be inherited (Epic Games, n.d.). This is similar to Unity's prefabs and have all the same advantages despite being a different implementation.

Unreal, like Unity, has the same Scene structure, albeit Unreal refers to scenes as ‘Levels’ (Epic Games, n.d.). It also features an editor, similar to Unity’s, in which the entire game can be created without coding, if it is simple enough.

It has been noted, however, that Unreal is an advanced game engine, and is therefore quite difficult for inexperienced developers to get to grips with it, especially as it uses C++ (Wilcox, 2014) (Ronai, 2017). It is also more expensive than Unity.

MonoGame

MonoGame is a game engine based on Microsoft’s XNA toolkit. It supports Android, iOS, Mac, Linux and Windows as well as consoles such as the PlayStation 4 and Xbox One (MonoGame, n.d.).

While MonoGame is not specifically a component based game engine, it is notable as a game engine that is specifically code-oriented, with only a few GUI tools to assist in development. This obviously carries the many disadvantages of not using a component-based approach. The engine provides no standalone editor and is used through Visual Studio, MonoDevelop or Xamarin Studio (MonoGame, n.d.). This makes MonoDevelop a good option for game developers who wish to focus on writing code, and makes MonoGame a strong educational tool for young or otherwise inexperienced developers.

It is quite difficult to find many well featured, free, open-source, code-oriented, component based game engines, and there is clearly a gap in the market.

Libraries

OpenGL

OpenGL is a 3D graphics and modelling API standard controlled by the Khronos Group, which provides a software interface to a computer’s graphics hardware. It is intended for a variety of purposes like computer-aided design, CGI for films and video games. Published in 1992, it has been implemented by hardware manufacturers like AMD, NVIDIA, Intel and Apple. It has become very popular since its conception for a variety of reasons. For instance, while hardware manufacturers need to licence their implementation, software developers do not, OpenGL is cross-platform and features what is known as the ‘extension mechanism’, which allows hardware manufacturers to add extra features to their hardware and expose them to software entirely through OpenGL (Wright, et al., 2011). OpenGL also features a powerful shading language called ‘GLSL’ (Tatarchuk & Licea-Kane, 2005). Finally, OpenGL includes several other libraries such as a utility library called GLU. GLU provides functions for projection matrices, describing complex objects, processing of surface rendering and other more complicated tasks (Hearn, et al., 2011).

One popular implementation on OpenGL is OpenTK. Open Tool Kit is an open source set of libraries including OpenGL, built around the .Net framework. It is provided free for anyone to use (OpenTK, n.d.).

OpenAL

Open Audio Library is a cross platform 3D audio API which can simulate moving sound sources in a 3D environment as well as multiple other audio effects for video games (OpenAL, n.d.). The OpenAL specification was first released in 2000 and is supported by hardware manufacturers, Creative Labs and NVIDIA. It was designed to be cross-platform and easy to use alongside OpenGL (OpenAL, 2005).

OpenAL also a library included in OpenTK, although the redistributable DLL for OpenAL must be downloaded separately (Xamarin, n.d.).

4 Technical Development

System Design

The entity component design is not exclusive to this kind of software. It is actually based on a common design pattern known as the decorator pattern. The decorator pattern's strengths are in its flexibility, as base objects are expanded at runtime via property objects, but has the weaknesses of adding many small classes to a design, making it hard to understand easily (Freeman, et al., 2004). Using this standard design pattern, a basic structure was developed for this game engine. Specifically, for the relationship between game objects and components.

When a game is being developed, being able to test quickly is a very important feature of a game engine, as it allows small changes to be made quickly. If it were that the whole engine needed to be compiled every time a small change were made, this feature would not have been seen in this engine. For this reason, the engine was designed to be a class library structure that fits alongside a C# application. Whenever a small change is made to a scene within the game, only the scene and base game class would need to be compiled, speeding up development time.

As the engine was to be a class library, it was needed that an overall structure be defined. The overarching structure was decided as such: objects, components, managers, and systems. A basic structure of this design can be found at Appendix A: Overall basic engine structure.

Overall structure

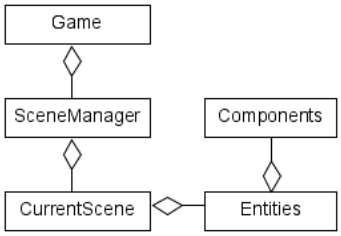


Figure 2 Structure of scenes

Objects include the base Entity class, which holds a group of components, as well as the base Scene class, which represents a game 'level'. The scene holds all the game logic for that 'level' of the game, including the list of handles to entities for that scene. This can be seen in Figure 2. The scene manager here acts as a container for the current scene and links up methods in the scene to the game.

Components simply are all the various properties as objects that can be applied to an entity. Included are transform, audio, collision, and more.

Managers are the objects that actually hold on to the objects in the game. For instance, all entities are held within a list in the entity manager. The entity manager provides accessor and mutator classes for manipulation of entities, but entities themselves never leave the entity manager unless the game designer has some need to – however this would be going against the intention of the design. The other managers are the resource manager and the scene manager.

Finally, systems can be thought of as where the game logic is executed with regard to the components. Systems read the entities and the relevant components in order to perform operations on them. All systems have access to all entities and all components within them, making the full scope of what a system can do to an entity very vast. Systems are also designed to be removable and replaceable, as modularity is a fundamental part of a game engine design. The game designer is free to create their own system if they will it.

All of this being separated in a class library away from the game makes the actual game code simpler and easier to manage, however it does create the problem of not being able to modify the various systems and managers if all the game developer has is the DLL file for the engine. The Unreal engine approach, as discussed in the background section of this report, to this problem is to provide the source code of the engine so that it can be modified and then recompiled - which is also the approach this engine takes. However, because this engine is specifically code first in terms of actual game development, another opportunity is presented. By putting the engine execution code in the game code rather than the engine library, the game developer can modify the game more easily than a process that requires the recompiling of the engine.

It has been previously established in the background section of this report that modularity is an important part of any game engine and this is achieved in this engine by establishing the systems and managers in use at runtime. A base ‘Game’ class exists in the game code (but not the engine class library), which sets up the systems, managers and OpenGL graphics options, and then establishes the game loop. The structure of this can be seen in Figure 3. A game developer could change what systems and managers are in use, and could even write their own using the base classes available in the engine class library. This approach provides options to the developer, modularity, recyclability, and expandability.

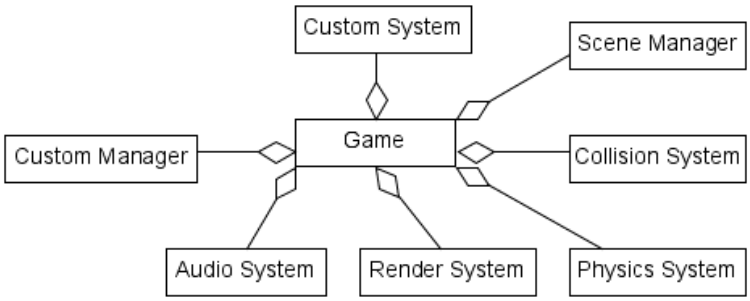


Figure 3 Managers and systems are aggregated to the base game class

Class Breakdown

Entity

The entity acts as a container for a group of components, which describe an object within a game. The entity object can be thought of as a small, instantiated, component manager of sorts. It contains a list of all the components as well as methods for accessing and mutating them.

The entity object's main other function is to hold on to an ID handle which will be used by systems and the entity manager to identify it, as all entities are instances of the same class. All ID handles in this engine are generated as GUID objects. This is done for simplicity and convenience purposes; as GUID objects are guaranteed to be unique and also, having to create custom code to generate a unique ID would be unnecessary, apart from in an attempt save on memory usage, but could possibly limit how many entities can exist, reducing flexibility. Entity handles are generated within the entity when they are constructed, and do not necessarily rely on the entity manager; therefore, an entity could be constructed within a game scene and not given to the entity manager at all. It is hard to see why this would be done however, as systems would not be able to access an entity not stored in the entity manager. Entitles can be added to the entity manager after they have had components attached to them, however.

Component

All components inherit from the same component class. The component class generates a handle, similar to how the entity class does the same, and sets the component type. The component type is an enumerated value, which is used to identify what kind of component (texture, collision, audio etc.) it is. Component type is available globally throughout the engine namespace. An entity cannot have more than one of any type of component - this is to prevent conflicts; if there are two transforms, which should be used when determining the position on the screen to draw the entity? This approach does hold the disadvantage of not allowing the entity to have more than one texture or audio file, but this is a necessary sacrifice for the sake of consistency, however it should be noted that a more complex game engine would allow some component types to be added to one entity more than once, depending on what they are.

Components are not expected to hold any particular logic; they are for holding data that describes the properties of a game object only. However, it was found to be necessary to build some logic into the transform component for the sake of making the component more useful. Building functions, which can add to the position, rotation, or scale of the main transform matrix, were found to be advantageous when testing the functionality of the engine, because otherwise, an unnecessary amount of code was required in the scene to do the same thing, and this lead to code that was difficult to read.

Scene

The scene class is very simple in implementation. New scenes are inherited from the scene class and the IScene interface. They are then placed into the scene manager, which links the methods in the scene to the base game class. When a new scene is loaded, previous

entities, and by extension, components, are destroyed and removed from memory by the entity manager. Components that have held assets are also destroyed, but the resource manager does not delete those assets. This is because it is practically impossible for the resource manager to know when all the components that need the asset is question have all been destroyed, so unloading could cause a runtime error. In any case, the function of this engine is to be a lightweight engine, so these kinds of optimisations are not the priority – it is unlikely that a shortage of memory will be a problem for the kinds of games that this engine is intended for.

System

The system class is where the logic pertaining to the components is performed. A system looks through all the entities within the entity manager and reads their components, pulling out the components with the appropriate types and using that data to perform related actions.

The render system takes a texture and the transform and draws that texture in a polygonal box that has been transformed by the transform information of the transform component. When initialised, the render system loads a set of vertices with texture mapping into a buffer using OpenGL via OpenTK. This buffer is used to draw all the textures. This does mean that the transform needs to be scaled appropriately in order for the texture to be drawn in the correct proportions, this is not hard to do however.

The audio system uses the OpenAL library to set up a listener and a source; it then reads an audio component from the entity and uses that information to assign an audio buffer, which has been generated previously in the resource manager, to the system's source and to play the sound. An audio component also contains a Boolean value, which describes if the sound should start playing in the current update or not.

The collision system looks for entities that have a collision component. A collision component simply holds a Boolean value to keep track of whether this entity is collided with anything or not. For each entity that has a collision component, the collision system compares the transform of that entity to all the transforms of all the other entities that have collision components. The system determines whether two entities are overlapping, and if they are, sets both of their collision components to 'collided'. The collision takes into account not only position, but also scale and rotation when it is making calculations. The collision 'hit-box' is defined as being the same size as the render system's default box size, augmented by the transform of the entity.

The physics system is closely related to the collision system. Indeed, the only reason they are separated is the potential usefulness that being able to know that an entity has collided this update could provide. In addition, the game designer might not want the engine to provide collision response for all objects that have collision components; they may wish to detect collision only, and write their own code for how to respond to it.

The function of the physics system is that it looks for a rigid body component, as well as a collision component. A rigid body component holds information such as the previous position of this entity, as well as a vector that describes gravity's effect on this entity. If an entity has collided with another, the physics system will move that entity back to its previous transform.

This does not immediately make the collision component read as ‘not collided’ so a scene can also detect and respond to collisions in its own way. The physics system then uses the gravity vector in order to move the entity slightly in the direction specified by it. Because of this, it should be noted that gravity is defined on a per-entity basis rather than a global one. This is in aid of improved flexibility, as it cannot be predicted that the game designer will want all entities to be affected by gravity in the same way. The downside of this is that if all entities are given the same gravity, it is hard to change mid-way though a scene as it would require looping through all the entities and changing the gravity manually. This is considered an unusual problem, however, with potential work-arounds such as implementing a custom gravity system which does use global gravity.

Resource Manager

There are three managers in this game engine, the entity manger, the scene manager, and the resource manager. All but the latter have been described in acceptable detail in passing so far in this report. The resource manager has not so far had such adequate mention. The resource manager is responsible for loading textures and wav audio files. Textures are loaded into an OpenGL buffer, and audio is loaded into an OpenAL buffer. When both are loaded, the relevant APIs provide a handle in order to address this buffer. The handles are stored in relevant dictionaries alongside the names of the files that were loaded. This means that if a request to load an already loaded asset is made, the resource manage can simply look up and return the handle it has stored.

This is important because it can significantly saves on memory usage. If many entities wish to use the same texture component, it would be inefficient and redundant to load the same texture information multiple times when all the entities could just use the same buffer. This is especially important when it is considered that the standard computer model, the Von Neumann architecture, is mainly bottlenecked by too much data being moved around the system (Eigenmann & Lilja, 1998).

Implementation

As has been previously inferred, the implementation of this engine uses OpenTK - mainly because of its utilities and implementation of OpenGL - and OpenAL, for its simplicity in generating sound within a 3D environment. This means that the dependencies of a game produced with this engine are both OpenTK and OpenAL, which are both freely available and therefore provide no hindrance to the possibility of using this engine for most people. OpenAL does require downloading of the OpenAL redistributable DLL package, which is available through Microsoft’s Nuget package distribution system (Nuget, 2013).

Starting a game project off from scratch with this engine would be difficult, as it requires a base class to be created that has all the systems and managers set up correctly. It is not implicitly obvious how to do this without instruction. For this reason, it has been decided that the engine should be packaged with a template project that has the base class already created and a scene, along with dependencies. This should be enough to get developers going with their game in this engine. If only the class library was presented to a developer, the might struggle to know what to do with it, and how to use it. Obviously, documentation would be required with the distribution of this engine. An example of a quick start guide can be found in Appendix C: Quick start user guide

Testing

Testing in a game engine is a relatively simple process, but also one that is difficult to automate. One of the best ways to test an engine like this is to create a game of test scenarios and observe results. For instance, if the collision system is being tested, a scene with two objects could be created in which one-object moves to collide with the other. If the response is not what was expected, the test failed. This can be a slow process but does allow for convenient debugging. In this way a whole ‘test-game’ can be developed which can be conveniently used to check that every feature of the engine is working as expected. This would of course, be quite time consuming but potentially worthwhile if any major engine alterations are performed, so that no setup is required to test features quickly.

If errors are found in the engine code, standard debugging tools are available though most IDEs, and can be used to identify issues. During the development of this engine, debugging was used very often in cases when things did not work as expected. Because this engine is comprised of a class library, it is necessary that the test game should be included as part of the project solution, so debugging can occur with ease. A full testing document for this can be found at Appendix B: Engine testing document. A fully featured test game should have scene scenarios for every one of these tests.

Testing is not only limited to finding bugs however. During the development of the engine, several methods and functions were added to some classes because it was found to be more convenient and produced more readable code. For instance, components and entities are accessed through lists but identified through a handle. Whenever either needs to be accessed, they need to be found in their lists. The code for this was found to be very repetitive when setting up tests for functions in the engine, so a function for both was made which returns the index of the entity or component based on a given handle. Also, the transform component has functions which allow the position, rotation and scale values to be added to without having to copy the transform matrix out of the component and into the scene. This improved optimisation and made code more readable.

Performance of the engine was never an issue, owing to the fact that it is very simple, so optimisation was not so much considered during development in lieu of implementing more features. However, this is not to say that no consideration was given to optimising the code – there were several improvements made for the sake of improving the performance. From calling the garbage collector at times when entities were being unloaded, to setting up OpenGL buffers and matrices in order to correctly utilise the graphics hardware of the computer.

Overall, testing proved to be an important part of the development of this project, as improvements were made due to testing – in some ways, the testing of this project shaped the functionality of the finished product. This is cohesive to the requirements of a useful game engine, because one of the purposes of a game engine is to provide a framework which allows for quick development and testing of a game, so providing tools which allow testing to be done quickly is important.

Process and Methodologies

During the development of any big project, unexpected changes to design and implementation occur quite often. These can occur if technical problems are difficult to overcome, if the specification was not thorough enough leading to unexpected changes in design, or even unfortunate circumstances relating to illness or injury. It is therefore unreasonable to assume that an initial time plan is likely to provide an accurate prediction of how a project will be developed and when everything will be completed by. This is not to say that an initial time plan is unnecessary, it is important in order to prove a project can be completed in the time allotted, but it is uncommon for a project to remain exactly on the same time plan by the end. For this reason, this project was developed under the agile methodology. Week by week, an analysis of what had been completed was performed, and using that, it was decided what needed to be done in the next week. This means the if any problems occur where sections of the software are taking more time than expected to complete, the overall plan could be quickly adapted. The project was completed but not on the same schedule as was set out in the initial time plan.

Agile development also implies changes to the initial project plan. Several small changes have been made to the plan through the implementation process due to decisions made related to more fully understanding the problem, or simplifying areas that took too long to develop. Initially, for instance, components were going to be held within their relevant system, rather than in the entity, as it ended up being. This was a change made because it was decided that components belonging to systems caused too much dependency in what is supposed to be a modular framework. If for instance, the collision system was to be removed, the physics system would break because it would rely on the collision system existing. By keeping components contained within entities, the engine is made much more flexible and modular, which is an important principle of game engine design, as has been discussed in the background section of this report.

Much of the success of this project can be put down to using an agile development process. By dynamically reducing and expanding sections of the scope of this project, a more functional and balanced engine was achieved, because some of the original design aspects ended up being either too complicated or too under developed for this engine to work.

5 Evaluation

This project is now completed in that it has been delivered in time for the project's deadline, but – as with any project – work can still be done to improve it. If it were that the project would be continued to be developed passed this deadline, there are many things that could be improved upon and many features that could be added to make this engine a more usable and functional game engine. This was always going to be the case from the start; as stated in the introduction, because of the scope of a fully featured game engine, it would be impossible to implement every feature of a typical game engine in the time allocated to this project. This project was only ever intended to make a partial, simple game engine which is basically functional as a game engine.

The initial report included aims and objectives for this project, all but one of these have been met. The input manager ended up not being implemented because, while the function was

clear, the necessity was not. Initially the idea was to give support to game controllers like the Xbox One controller. However, OpenTK provides functions for the use of controllers in games, so it was unclear as to how an input manager would improve the ability to map controllers to inputs in the game. The idea that keyboard controls could be used alongside controllers is one that draws from features of other game engines, however this would require creating interfaces for being able to setup controllers in games, and this proved too complicated for the scope of this project, especially as other more important features of the engine had not been completed thoroughly enough by the time the question of the input manager had properly arisen.

The first aim was to create a rendering manager. It is referred to as a manager here because when the aim was written, it was not fully understood by the author what the difference between a system and a manager was. A rendering system was created, and it was indeed the first system to be developed, as is suggested in the initial objective's description. This rendering system uses Open GL via OpenTK to draw textures to the screen via polygonal 2D boxes. It's a simple solution for a 2D graphics systems, which works well to tie in with the collision system, which requires hit boxes. By making the render system draw everything as boxes and rely on the entity being scaled and rotated to fit the texture's proportions, the collision system incidentally gets a correctly sized and rotated hit-box around the entity. This saves on code and optimises the amount of memory being used by the engine as a whole. The render system also uses OpenGL buffers to load the vertices and texture mapping information, further optimising the pipeline of the engine.

The render system was initially designed to be in 2D. This was keeping in mind the time constraints of developing this engine. If further work is to be done, the obvious thing to do would be to add support for 3D graphics. This would be a large task to complete, but would make the engine more useful for more kinds of games. Further features would have to be added, including the addition of loading 3D model files, and shader support. This engine did initially have shader support, but it was removed because it was considered that the shader was not able to be easily edited without modifying the render system, and everything the shaders were doing could have been achieved without them, using inbuilt OpenGL functions. Even more features could be added as well, including proper support for sprites rather than boxes which hold textures, which again would be useful for 3D rendering.

If the engine were to support 3D, other systems would also have to be modified, like the collision system, which would have to also be able to detect collisions in the z axis, rather than just the x and y, as is the current case. This would also affect the physics system in a similar way.

The physics system was the second objective. The physics system, in the original design, also handled collisions. This was separated out into two systems because it was considered that it would be useful to the game designer if the engine could detect collisions separately to responding to them, so that collisions could be responded to in a custom way in the scene class. Collisions are detected between objects while taking into account the rotation, scale and positions of the objects. This information is then used by the physics system to prevent objects from overlapping. The physics system also has a gravity function on a per-object basis which acts as a kind of velocity function. An object that has gravity will move in that direction and will respond to collisions as appropriate.

Other than making the physics and collision systems support 3D physics, both systems could be expanded to feature more physically based simulation. Objects could be given properties like mass and density, and therefore would respond to gravity more realistically. For this to mean anything, objects would have to respond to collision more realistically as well. Instead of simply stopping, objects should bounce off of other objects, simulating reality. This is a feature that should be capable of being disabled, but would make a more useful and featured game engine when it comes to more realistic games.

The third objective was to create an audio manager. An audio system for this engine does exist, and can load and play audio when it has been triggered by a game scene. Audio is loaded into one OpenAL source and is played to one OpenAL listener. The way this system operates is quite limiting however. Only one sound may be played at any one time, because there was a neglect to expand this system out to be more properly functional. The first thing that should be done to improve this system is to allow for more sounds to be played at once. Also, the game becoming 3D means that 3D special sound would also be an invaluable feature to implement. This is a feature built into OpenAL and would not be very difficult to implement. Other OpenAL features should also be included, such as dynamic audio effects. Audio is an important part of any game, but this engine, as it currently stands, it is lacking in this area.

The next objective was to create an object manager. An entity manager does exist in the engine, and has the ability to hold on to and manage all the entities use in a game. The entity manager allows systems and scenes to manipulate all the entities as they need to, using a handle structure to improve efficiency, as entities do not need to be copied to scenes or systems when they are manipulated. The entity manager is a relatively simple manager because it is not required to do very much, however this once could still be improved. Mainly, the structure in which entities are held could be bettered. Currently, entities are held in a basic list structure, but this has the disadvantage of being difficult to manage. If the entities could be held in a more sophisticated database structure, efficiency could be improved, as well as a potential for more functionality in regards to handling multiple entities.

Fifth on the list of initial objectives was a scene manager. The scene manager in this engine is the simplest section of all. It has the ability to link methods in a scene to methods in the base game class. An expansion of the functionality of the scene manager would include the ability to save the state of a scene, so that, for example, in an adventure game, a character could go inside a building, which would load another scene, and then could leave the building to find that the previous scene has been restored to how they left it. This could vastly improve the potential for more complicated games made with this engine.

The final objective was to create a demo game for this engine. This has been done and is effectively shows off the features of this engine. If the features of this engine were to change and be expanded upon, this demo game would have to be adapted to reflect this.

There are other areas of the engine that also need expanding and adjusting. One major issue is the lack of any kind of error manager. The error manager is responsible for handling errors in that it logs them and prevents crashes, or at least handles them in a graceful manor in order to prevent damage to the data of the game. Without an error manager, this engine just crashes and gives no explanation why, other than with debugging tools.

The next important addition would be the inclusion of the functionality to save games. Saving games is important to game developers, because almost all games are complicated enough to need to be saved, as most players wouldn't complete a whole game in just one sitting. A save game manager would write all variables and objects to a file, probably after encrypting it in some way so that players cannot cheat very easily. This functionality is possible with this game engine, but it must be written by the game developer. Having an inbuilt system for this would save the developer time.

One final major immediate expansion would be a scripting manager. This would allow script objects to be attached to entities. A script object is simply a C# code file with logic written about that entity. This allows for much more custom functionality that could be shared between multiple entities as components. This would especially help in reducing code, scene by scene. Logic stored as a component would mean less duplication of code if the same logic is required in multiple scenes, and across multiple entities. It would also reduce the amount of code written into a scene overall, as most logic would be attached to entities as components, and only logic that is in regard to functionality between objects. This is probably the most potentially great expansion of functionality within this engine, as it would allow for far more complicated game ideas to be implemented more simply.

Overall, this project meets the main objective of creating a component based game engine. Six of the seven objectives have been met, as one objective was scrapped during development because of reasons already discussed in this evaluation. However, if development continues after the fact, the areas that could be improved upon are numerous. Many expansions can be made, and there are a few known issues, especially with collision detection, that need to be fixed, but are too complicated to be fixed in time for the deadline of this project.

6 Conclusion

The system-entity and entity-component models have become standards for game engines for many years now because they provide flexibility, modularity and expandability to the very complicated problem of creating tools for game development. Creating a modern game is an enormous task, and using a game engine has become an essential part of the process for the majority of game developers, as they provide a powerful framework for developers to more freely implement and develop game concepts into finished games.

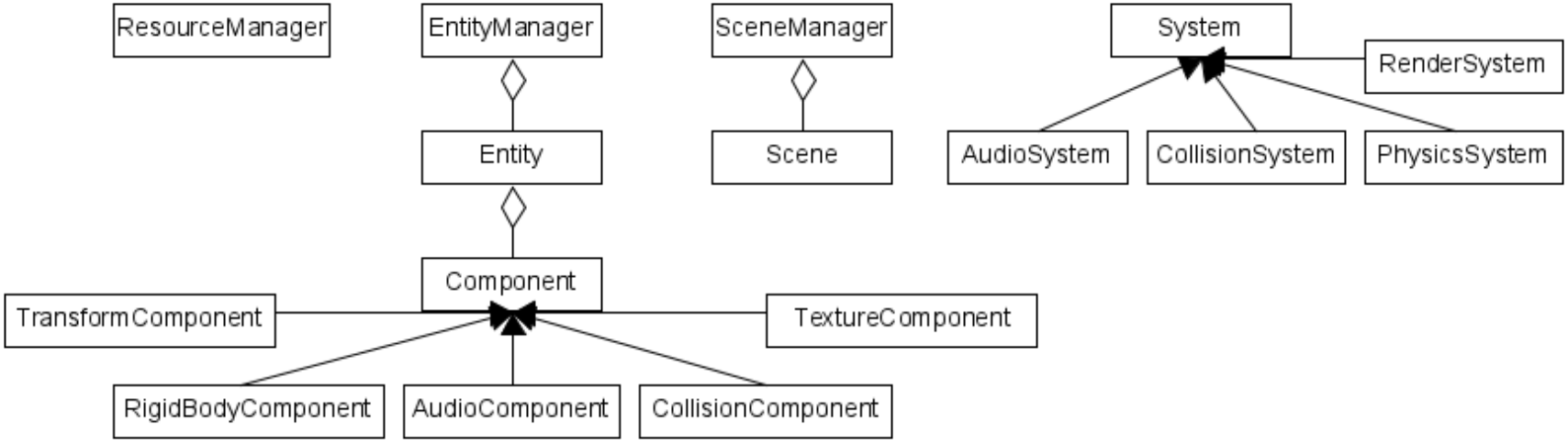
The objective of this project was to create one of these component based game engines, and this has been achieved as well as all apart from one of the initial objectives, for reasons discussed in the evaluation of this report. Over the process of developing this project, from initial planning to final testing, an enormous amount has been learnt about both the structure and functionality of a typical component based game engine, and the process of developing a large and complicated project.

The finished project meets all of the initial specifications, but could be expanded greatly if development were to continue. The engine has a few known issues which would need to be fixed, as well as important extra features that there was not enough time to implement in the available time.

If this project were to be started again, not very much would be changed. Entitles would be stored in a more dynamic and functional database structure, the features of the engine would be expanded to include things like and entity manager and scripting system, but also would support 3D graphics. When it was decided that components should be held inside entities instead of their relevant systems, it was with an attitude of compromise, however, it has been realised since this is a preferred method, because it promotes the principles of modularity that are so important to the structure of a game engine. The structure of this engine is very sound, so changes that would be made do not require the restarting of this project.

This project has overall been successful in meeting its goals and has provided a lot of learning experience with regards to game engines, game development and general software engineering and project management. If development should continue, there would be much to do, but considering the full scope of other game engines, it is unlikely to ever be truly completed.

Appendix A: Overall basic engine structure



Appendix B: Engine testing document

What is being tested	Description	Prerequisites	Expected Result
Rendering / Resource Management	Do textures display	A game object with a texture component and a transform component set to default scale and rotation with a position of 0, 0	Texture will appear in as a square in the middle of the screen
Rendering	Can textures be scaled	A game object with a texture component and a transform component set to default rotation with a position of 0, 0, and a scale of 2,2	Texture will appear as a square in the middle of the screen 2 times bigger in both axes than in the previous test
Rendering	Can textures be scaled in only the x axis	A game object with a texture component and a transform component set to default rotation with a position of 0, 0, and a scale of 2,1	Texture will appear as a square in the middle of the screen 2 times wider than it is tall

Rendering	Can textures be scaled in only the y axis	A game object with a texture component and a transform component set to default rotation with a position of 0, 0, and a scale of 1,2	Texture will appear as a square in the middle of the screen 2 times taller than it is wide
Rendering	Can box be rotated	A game object with a texture component and a transform component set to default scale with a position of 0, 0 and a rotation of 45	Texture will appear as a square in the middle of the screen at default scale and rotated 45 degrees clockwise
Rendering	Can multiple textures be drawn at the same time	Two game objects with transform and texture components, transforms set to default scale and rotation, with the positions in different screen locations	Both textures should display as separate boxes in different locations on the screen
Rendering	Can textures with transparency draw over each other	Two game objects with different textures – one with not transparent areas and one with an area of transparency. The Non transparent object should be added first, and the transparent object should be added second. Both should have default size and rotation, with position 0, 0	Both textures should appear on top of each other, with the top texture revealing the texture below only in the transparent areas of the top texture.
Audio / Resource Management	Can sound be played	A game object with a sound component that loads a sound file, with the scene set up to change the sound components 'triggered' variable to true at the press of a button	When the specified button is pressed, a sound should be played.
Collision	Do objects detect collision	Three game objects with texture, transform and collision components. Transforms should be set to default rotation and scale, and position should be set to separate locations on the screen. One object should be set up to move with keyboard input	When one object moves to a location where it is overlapping another, debugging should show those two's collision component's 'isCollided' variables set to true, whereas the other object should be false
Collision	Do objects detect collision when one object is scaled	Three game objects with texture, transform and collision components. Transforms should be set to default rotation and scale, apart from one which should have its scale set to 2, 2, and positions should be set to separate locations on the screen. One object should be set up to move with keyboard input	When one object moves to a location where it is overlapping with the scaled object, debugging should show those two's collision component's 'isCollided' variables set to true, whereas the other object should be false

Collision	Do objects detect collision when one object is rotated	Three game objects with texture, transform and collision components. Transforms should be set to default scale and rotation, apart from one which should have its rotation set to 45, and positions should be set to separate locations on the screen. One object should be set up to move with keyboard input	When one object moves to a location where it is overlapping with the rotated object, debugging should show those two's collision component's 'isCollided' variables set to true, whereas the other object should be false
Collision	Do objects detect collision when one object is rotated and scaled	Three game objects with texture, transform and collision components. Transforms should be set to default scale and rotation, apart from one which should have its rotation set to 45 and its scale set to 2, 2, and positions should be set to separate locations on the screen. One object should be set up to move with keyboard input	When one object moves to a location where it is overlapping with the rotated and scaled object, debugging should show those two's collision component's 'isCollided' variables set to true, whereas the other object should be false
Collision	Do objects detect collision when one object is rotated and scaled only on the x axis	Three game objects with texture, transform and collision components. Transforms should be set to default scale and rotation, apart from one which should have its rotation set to 45 and its scale set to 2, 1, and positions should be set to separate locations on the screen. One object should be set up to move with keyboard input	As above
Collision	Do objects detect collision when one object is rotated and scaled only on the y axis	Three game objects with texture, transform and collision components. Transforms should be set to default scale and rotation, apart from one which should have its rotation set to 45 and its scale set to 1, 2, and positions should be set to separate locations on the screen. One object should be set up to move with keyboard input	As above
Collisions	Do objects detect collision when both objects are scaled	Three game objects with texture, transform and collision components. Transforms should be set to default rotation and scale set to 2, 2, and positions should be set to separate locations on the screen. One object should be set up to move with keyboard input	When one of the scaled objects is moved to a location where it is overlapping with the other scaled object, debugging should show those two's collision component's 'isCollided' variables set to true, whereas the other object should be false

Collision	Do objects detect collision when both objects are scaled only on the x axis	Three game objects with texture, transform and collision components. Transforms should be set to default rotation and scale set to 2, 1, and positions should be set to separate locations on the screen. One object should be set up to move with keyboard input	As above
Collision	Do objects detect collision when both objects are scaled only on the y axis	Three game objects with texture, transform and collision components. Transforms should be set to default rotation and scale set to 1, 2, and positions should be set to separate locations on the screen. One object should be set up to move with keyboard input	As above
Collision	Do object detect collision when both objects are scaled on different axes	Three game objects with texture, transform and collision components. Transforms should be set to default rotation and scale set to 1, 2 on one object, and 2, 1 on the other (the scale of the third is not important), and positions should be set to separate locations on the screen. One object should be set up to move with keyboard input	As above
Collision	Do object detect collision when both objects are scaled on different axes and one is rotated	Three game objects with texture, transform and collision components. Transforms should be set to default rotation on one object and 45 on the other and scale set to 1, 2 on one object, and 2, 1 on the other (the scale of the third is not important), and positions should be set to separate locations on the screen. One object should be set up to move with keyboard input	When the object that is scaled and rotated is moved to a position where it is overlapping with the not rotated and scaled object, debugging should show those two's collision component's 'isCollided' variables set to true, whereas the other object should be false
Collision	Do object detect collision when both objects are scaled on different axes and both are rotated	Three game objects with texture, transform and collision components. Transforms should be set to 45 on the other and scale set to 1, 2 on one object, and 2, 1 on the other (the scale of the third is not important), and positions should be set to separate locations on the screen. One object should be set up to move with keyboard input	When the object that is scaled and rotated is moved to a position where it is overlapping other scaled and rotated object, debugging should show those two's collision component's 'isCollided' variables set to true, whereas the other object should be false

Physics	Does gravity affect objects	Two game objects with transform, texture and rigid body components. One should have a gravity of at least 1 in any direction. Transforms should be default for scale and rotation, with different positions on the screen	Object with gravity vector should be moving without input, while the other should be still
Physics	Do two objects respond to collisions	Two game objects with transform, texture, collision and rigid body components. Transforms should be default for scale and rotation, with different positions on the screen. One object should be set up to move with keyboard input	It should not be possible to overlap the two objects, but it should be possible to immediately move the objects away from each other after they have collided
Scene Management	Can scenes be switched between	Two scenes completed with game objects that display different textures to the screen, and some amount of movement happening in each, including buttons set to switch between scenes	When the appropriate button is pressed, the relevant scene will display, when a button is pushed to switch back, the previous scene will display back in its original configuration

Appendix C: Quick start user guide

Open the template project. In this project you should see the files, ‘Program.cs’, ‘Game.cs’ and the folders audio, scenes and textures. Audio and textures are where you should put your audio and texture files respectively. Do not delete ‘missingTexture.bmp’; this is required by the game engine. When you import new textures and audio files, make sure you put them in the correct folders and set them to copy to the output directory of your game when you compile audio files should be in wave format only. If the texture you have specified is not found, it will be replaced with ‘missingTexture.bmp’, which looks like this:

Inside the scenes folder you should see a scene called ‘SceneDefault.cs’. This is a base scene class that is blank. To create new scenes, it is recommended that you copy this scene in order to create new scenes, as it is set up correctly already.

To switch to a new scene within a scene, you should use the following code:

```
sceneManager.setScene(new Scene2(sceneManager));
```

In this instance, ‘Scene2’ is a new scene that has been copied from scene default.

The following is an example of how to create a new entity with all components:

```
bubbleHandle = EntityManager.createNewBlankEntity();

randTrans = new TransformComponent(Matrix4.Identity)
{
    Position = new Vector2(250, -375),
    Scale = new Vector2(0.5f, 0.5f)
};
randTex = new TextureComponent("Bubble1.png");
randColl = new CollisionComponent();
randRig = new RigidBodyComponent()
{
    Gravity = new Vector2(0, 0.6f)
};
AudioComponent bubblePopSound = new AudioComponent("bubblepop.wav");

int entityRef = EntityManager.entityRefFromID(bubbleHandle);

EntityManager.Entities[entityRef].addComponent(randTrans);
EntityManager.Entities[entityRef].addComponent(randTex);
EntityManager.Entities[entityRef].addComponent(randColl);
EntityManager.Entities[entityRef].addComponent(randRig);
EntityManager.Entities[entityRef].addComponent(bubblePopSound);
```

You do not need to include all components in an entity.

The following is an example of how to manipulate an entity:

```
foreach(Entity ent in EntityManager.Entities)
{
    if(ent.Handle == boxHandle)
    {
        int boxRef = EntityManager.entityRefFromID(boxHandle);
        int transformRef = EntityManager.Entities[boxRef].compRefFromType(ComponentType.Transform);

        ((TransformComponent)ent.Components[transformRef]).addToPosition(movement * (float)e.Time);
    }
}
```

The variable, ‘movement’ is a vector 2 that contains the intended movement for this update.

Properties of the game window can be set in the ‘Game.cs’ constructor:

```
public Game() : base
    (
    1152,
    640,
    GraphicsMode.Default,
    "My Game",
    GameWindowFlags.FixedWindow,
    DisplayDevice.Default,
    3,
    3,
    GraphicsContextFlags.ForwardCompatible
    )
{ }
```

The first two variables are the window width and height, the fourth variable is the title of the window, the fifth specifies whether the window is resizable or not, or if it should be full screen. The other variables should not be changed apart from for advanced scenarios.

References

Andrews, J., 2009. *Designing the Framework of a Parallel Game Engine,* Santa Clara, California: Intel Corporation.

Blackman, S., 2011. *Beginning 3D Game Development with Unity.* New York, New York: Paul Manning.

Busby, J., Parrish, Z. & Van Eenwyk, J., 2005. *Mastering Unreal Technology.* Indianapolis, Indiana: Sams Publishing.

Eigenmann, R. & Lilja, D. J., 1998. *Vonn Neumann Computers.* [Online]
Available at:
https://web.sonoma.edu/users/f/farahman/sonoma/courses/es310/resources/10.1.1.78.4336.pdf
[Accessed 8 May 2018].

Epic Games, n.d. *Actors.* [Online]
Available at: https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Actors
[Accessed 7 May 2018].

Epic Games, n.d. *Components.* [Online]
Available at: https://docs.unrealengine.com/en-US/Engine/Components
[Accessed 7 May 2018].

Epic Games, n.d. *Frequently Asked Questions (FAQ).* [Online]
Available at: https://www.unrealengine.com/en-US/faq
[Accessed 7 May 2018].

Epic Games, n.d. *Levels.* [Online]
Available at: https://docs.unrealengine.com/en-us/Engine/Levels
[Accessed 7 May 2018].

Epic Games, n.d. *Unreal Engine Features.* [Online]
Available at: https://www.unrealengine.com/en-US/features
[Accessed 7 May 2018].

Freeman, E., Freeman, E., Sierra, K. & Bates, B., 2004. *Head First Design Patterns.* Sebastopol, California: O'Reilly Media.

Gestwicki, P., 2012. *The entity system architecture and its application in an undergraduate game development studio.* Raleigh, North Carolina, ACM.

Hearn, D., Baker, M. P. & Carithers, W. R., 2011. *Computer Graphics with OpenGL.* 4th ed. Boston, Massachusetts: Pearson Education.

Hughes, J., 2011. What to Look for When Evaluationg Middleware for Integration. In: E. Lengyel, ed. *Game Engine Gems 1.* Mississauga, Ontario: Jones and Bartlett Publishers, pp. 3-10.

MonoGame, n.d. *About.* [Online]
Available at: http://www.monogame.net/about/
[Accessed 7 May 2018].

MonoGame, n.d. *Understanding the Code.* [Online]
Available at: http://www.monogame.net/documentation/?page=understanding_the_code
[Accessed 7 May 2018].

Nuget, 2013. *openal.redist.* [Online]
Available at: https://www.nuget.org/packages/openal.redist/
[Accessed 7 May 2018].

OpenAL, 2005. *OpenAL 1.1 Specification and Reference.* [Online]
Available at: https://www.openal.org/documentation/openal-1.1-specification.pdf
[Accessed 7 May 2018].

OpenAL, n.d. *OpenAL.* [Online]
Available at: https://www.openal.org/
[Accessed 7 May 2018].

OpenTK, n.d. *FAQ.* [Online]
Available at: https://opentk.github.io/faq/
[Accessed 7 May 2018].

Perez, A. & Royer, D., 2000. *Advanced 3-D game programming using DirectX 7.0.* Plano, Texas: Worldwide Publishing.

Pethrus, L., 2016. *Middleware in Game Development.* [Online]
Available at: https://software.intel.com/en-us/articles/middleware-in-game-development
[Accessed 6 May 2018].

Rene, B., 2005. Component Based Object Management. In: K. Pallister, ed. *Game Programming Gems 5.* Hingham, Massachusetts: Charles River Media, pp. 25-37.

Ronai, A., 2017. *The Pros And Cons Of Using The Unreal Engine For Archviz Animations.* [Online]
Available at: https://andrasronai.com/2017/06/the-pros-and-cons-of-using-ue4-for-archviz-animations/
[Accessed 7 May 2018].

Stoy, C., 2006. Game Object Component System. In: M. Dickheiser, ed. *Game Programming Gems 6.* Boston, Massachusetts: Charles River Media, pp. 393-403.

Tatarchuk, N. & Licea-Kane, B., 2005. GLSL Real-Time Shader Development. In: W. Engel, ed. *ShaderX3.* Hingham, Massachusetts: Charles River Media, pp. 57-84.

Thorn, A., 2011. *Game engine design and implementation.* London: Jones & Bartlett Learning.

Tulip, J., Bekkema, J. & Nesbitt, K., 2006. *Multi-threaded game engine design.* Perth, Murdoch University.

Unity Technologies, 2017. *Scenes.* [Online]
Available at: https://docs.unity3d.com/Manual/CreatingScenes.html
[Accessed 7 May 2018].

Unity Technologies, 2018. *Prefabs.* [Online]
Available at: https://docs.unity3d.com/Manual/Prefabs.html
[Accessed 7 May 2018].

Unity Technologies, 2018. *The leading global game industry software.* [Online]
Available at: https://unity3d.com/public-relations
[Accessed 7 May 2018].

Unity Technologies, 2018. *The world's leading content-creation engine.* [Online]
Available at: https://unity3d.com/unity
[Accessed 7 May 2018].

Unity Technologies, 2018. *Using Components.* [Online]
Available at: https://docs.unity3d.com/Manual/UsingComponents.html
[Accessed 7 May 2018].

West, M., 2007. *Evolve Your Hierarchy.* [Online]
Available at: http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy
[Accessed 6 May 2018].

Wilcox, M., 2014. *Top Game Development Tools: Pros and Cons.* [Online]
Available at: https://www.developereconomics.com/top-game-development-tools-pros-cons
[Accessed 7 May 2018].

Wright, R. S., Haemel, N., Sellers, G. & Lipchak, B., 2011. *OpenGL SuperBible.* 5th ed. Boston, Massachusetts: Pearson Education.

Xamarin, n.d. *OpenTK.Audio.OpenAL Namespace.* [Online]
Available at: https://developer.xamarin.com/api/namespace/OpenTK.Audio.OpenAL/
[Accessed 7 May 2018].

Xie, J., 2012. *Research on Key Technologies Base Unity3D Game.* Melbourne, IEEE.

Yeeply, 2014. *Unity 3D Game Development: Advantages & Disadvantages.* [Online]
Available at: https://en.yeeply.com/blog/unity-3d-game-development-advantages-disadvantages/
[Accessed 7 May 2018].