A Simplified Game Engine for a Game Development Course

by

Rolf Weimar

supervised by

Patrick Marais

A dissertation submitted for the fulfillment
of the requirements of the degree
Master of Science

April 16, 2014

Plagiarism Declaration

I know the meaning of plagiarism and declare that all of the work in the dissertation, save for that which is properly acknowledged, is my own.

Abstract

The Video Game industry is maturing. Success in the video game industry relies on many things, including marketing, sound business practises, and top notch technical implementation. Games Engines are software systems that facilitate game production. The growth of the game industry has increased the demand for programmers trained in game development technologies. A simplified game engine, designed specifically for the game development courses which service the supply of graduates for the industry, could have many advantages.

This dissertation analysed the requirements of such a system. We found that such a game engine would need to be extensible, reusable, modular, be easy to learn, and be open source. It would also need to at least include graphics, audio, networking and pathfinding components. Our analysis found that no game engine currently exists that fulfills all these requirements. We designed and implemented a game engine to fulfill all these requirements. Our game engine is built around a module framework, where each task of the game engine is handled by a module. This modular design allows us to easily change functionality by adding, removing or updating modules.

All source code of the engine is available, thus any part of the engine can be changed if needed. Open source also means the engine is free for all to use. Game engines also need to be reusable so that in the industry the development costs of creating an engine can be amortised multiple projects, but also in a university context it means that time students can continue to use the system across multiple projects.

The system was tested by having students complete game development tasks using our game engine, ModEngine, and another comparable game engine. We used lines of code as a measure of code complexity and completion time as a measure of performance. We found that there is a statistically significant reduction in both the lines of code and the completion time of student's ModEngine assignments versus the comparison. Our p value (the probability that the data was due to chance alone) for lines of code is 9.662776×10^{-5} and for completion time is 0.018. Students were also given questionnaires to complete where they were asked about their experience using both engines. ModEngine was found to be easier to learn and was simpler to use; students can more easily explore game development concepts with ModEngine and can get started working with it much more easily.

Acknowledgements

A big thanks goes to Dr. Patrick Marais whose input was invaluable in completing this thesis. Also a big thanks goes to all the students who participated in the evaluation. Huge thanks goes to Dr. Hendranus Vermeulen who was a big help with the testing and results section.

Contents

List of Figures

Chapter 1

Introduction

The video game industry is maturing, and world wide incomes have grown from $27 billion in 2002 to $60.4 billion in 2009, and are estimated to reach $70 billion in 2015 [21, 20]. Games themselves have a long history, stretching back through time, but video and computer games are a far more recent development. One of the first computer games was *Spacewar!*, which was developed for the PDP-1. Distributed free of charge, the creators didn't make money from it [33]. *Pong*, developed by Atari, was one of the first arcade video games. By 1975, it made $40 million [12].

Since then, games have grown more complex, and the market has expanded. *Halo: Reach*, released September 14, 2010, earned $200 million on its first day on sale. *Call of Duty*, *Guitar Hero* and *World of Warcraft*, are all billion dollar franchises [27, 16]. Early games were tied to their hardware, where ROM chips would store the game data and code. Numerous hardware changes, and increased complexity meant that video game development in the early years was very different to modern game development [23].

Being an entertainment industry, production of video games is inherently risky, where the earning potential is decided by a large number of factors, including marketing, choosing the right demographics, sound business practices, and well designed and robust technical designs and implementation. Game Engines were designed to address some of the technical issues presented by video and computer games, promotes reusability, and reduce cost and production time [37].

One of the earliest Game Engine was *SCUMM* by LucasArts [30]. The *SCUMM* Engine was used in 12 LucasArts games, thus reducing the amount of code that needed to be written. This allowed LucasArts to focus mainly on content creation.

Game Engines, because they handle a lot of low level details, require specific domain knowledge in areas such as Graphics, AI, Audio, UI and Networking. Some companies, such as Epic Games, in addition to producing games, also produce flexible game engines that can be

used for certain types of games [17]. Each Game Engine has a particular type of gameplay that they specialise in, and due to the optimisation and design required to support these games, they do not normally handle very different genres of games easily, without significant code development.

1.1 Game engines in a game development course

Teaching game programming is a difficult endeavour, due to the complex and technically demanding knowledge being taught. Using a full-featured game engine such as *Unreal* [17] would be infeasible, due to the size of the engine, and the specifics of that engine, which require the code to perform in a way that is compatible with the *Unreal* engine. Also, using an engine like this would intimidate new students and discourage them from continuing with the course.

In order to couple our design and evaluation to a real-world use case, we chose a specific game development course as our focus: The Game Design co-major at the University of Cape Town's Computer Science department. It has the same courses as the Computer Science major in first year. In second year, the students begin some Game Design specific courses, which cover important areas of game programming, such as AI, Graphics, Design, and 2D game development, amongst others. In third year, students handle more advanced topics such as 3D graphics, as well as study more advanced techniques in topics they first started in second year. Tutorials and assignments present students with a specific problem to solve, but the assignments are all self-contained and do not interact with each other. A new approach is needed to give the students an experience of working with established code.

Full-featured game engines are infeasible to use in this context. While simple game production programs such as Game Maker [19] or Kodu Game Lab [28] allow a student to easily produce a game, they are very limited. They do not include a lot of functionality, and they either can't be extended, or they can only be extended through the use of a built in language made specifically for their program. A game engine made up of modules could fill the gap between full-featured game engines, and simple game production programs. It would allow any student to make a game, and to replace or adapt existing code for their own purposes. It could also be a central part of a game development course. A modular game engine could allow the finished assignments written by students to be connected to the game engine using the game engine's framework.

A game engine also needs to be extensible, which allows it to adapt to changing technology and requirement as the game industry changes. It must also support reuse over many projects as this means the cost of creating an engine can be spread across multiple projects. It also cuts down on development time as once a development team knows how to use an engine, they can more quickly start work on their next project.

1.2 A modular game engine

From these requirements we can construct a taxonomy of game engines. This taxonomy evaluates the degree to which currently available game engines satisfy the following requirements: Extensibility, Reusability, Modularity, Open Source and a shallow learning curve. This taxonomy is discussed in chapter 2.

To build a game engine that fulfills these requirements, we developed a game engine, ModEngine. ModEngine is built around a module framework, where each task of the game engine is handle by a module. This modular design gives us the advantage of easily changing the functionality of the engine by adding, removing or updating modules.

This architecture provides high level functionality for each of the tasks of a game engine namely, rendering, networking, physics, audio, pathfinding and networking. This design allows students to quickly get a prototype working, and it promotes fast iteration on design. The engine is open source, which is inherently extensible. Also, as is common with professional level game engines, it is designed for reuse across multiple projects.

Specifically, this thesis is exploring the following research question:
"Can a simplified and modular game engine developed for a game development course help students in the exploration of game development concepts?"

While this dissertation is about game engines in an educational context, it is not about game programming education: it is about the design and implementation of a game engine in this context. We have proposed a game engine that can be used across all sections of a game development course. We want to determine whether such a system allows students to more easily try out game development concepts. A game engine that is easy to use would benefit students.

1.3 Dissertation Layout

This dissertation is laid out as follows. Chapter 2 covers background material needed for the work covered in this dissertation Chapter 3 covers the design and implementation of the central framework of our game engine, and the other components that make it up. In chapter 4, we discuss our testing methodology, followed, in chapter 5, with a discussion of our results. In chapter 6 we discuss our conclusions and future work is explored. Finally, in chapter 7, we present the supporting documents used in our testing.

Chapter 2

Background

Game engines help developers to create games more easily. They accelerate development by managing much of the low level detail, allowing the developers to focus on content creation, rather than dealing with the underlying technology. Since game engines handle many low level details, creating a game engine requires specific domain knowledge in areas such as Graphics, AI, Audio, UI and Networking.

This chapter covers the background material of game engines and game development. We also gather requirements for our game engine, construct a taxonomy based on our requirements and test them against game engines currently available.

2.1 Computer Science Education

Teaching game programming is a difficult endeavour, due to the complex and technically challenging knowledge being taught. Educators and developers working in computer science have been developing systems that can help foster student's interest in computer science. One such example is GameMaker by YoYo Games. GameMaker is a program that allows users to easily create their own 2D and 3D games [19] [18] [9] . GameMaker has been used in educational research to explore its potential for teaching problem solving and critical thinking in high school. By using student's interest in games, GameMaker can introduce students to game design and programming concepts, and thus foster their interest in computer science [11].

An application called Alice, developed at Carnegie Mellon University, is being used in courses to teach fundamental programming concepts through a 3D graphics environment and an easy to use interface. This program is aimed at students with little to no programming experience. Their proof of concept study [32] showed that students involved in the Alice program and at high risk of leaving the computer science major had far higher retention rates than a

control group that were also of high risk but did not take part in the program. Since this dissertation is not about education, we do not consider this aspect further.

2.2 Game Engines

A game engine is a software system that facilitates development and content creation for computer and video games [39]. Game engine is a general term, encompassing a large variety of software that helps in game production, ranging from systems that only handle 3D graphics, all the way to complex systems that handle advanced 3D graphics, audio, networking and artificial intelligence.

Some companies, in addition to producing games, also produce game engines for use by professional game developers. These include Epic Games who produce the Unreal Engine [17], Valve who produce the Source Engine [15], and Crytek who produce the CryEngine [4].

2.3 Game Engine Design

A game engine for a game development course has very specific requirements. While a game engine for professional game production has requirements including state of the art rendering, high fidelity sound and complex scripting, among many other aspects, a game engine for a game development course should focus on suitability for a game programming course.

Professional game engines tend to have steep learning curves. These engines would not be useful in a game programming course, because students should be able to start working with the game engine at the start of the course. A game engine with a shallow code development learning curve allows students to start working with the game engine quickly.

Game engines were created to help with game development and content creation for games [37]. Game creation has become more iterative as game creators have discovered benefits to this development approach [42]. For a game engine to support iterative development, it must allow code to be adapted and changed to suit different technologies and requirements; it must be able to be reused over many projects; and it must allow different components (such as rendering or sound) to be swapped in and out. From these goals, come the requirements of extensibility, reusability and modularity.

Cost is also an aspect which should be minimized for a game engine intended for educational use. Ideally, each student should have access to the engine and its code. Using proprietary engines would be costly due to their high license fees, and their source code is often not available. An open source engine is available free of charge, and all the source code is available.

Game engines vary in the features they include [23], but the most common features are a 3D renderer, an audio system, a physics system, and networking functionality. These components are usually written from scratch for the game engine, but in some cases, middleware that provides the needed functionality is incorporated into the engine.

To summarize, a game engine suited for a game development course requires 5 features: Extensibility, Reusability, Modularity, Open Source and a shallow learning curve. We constructed a taxonomy to determine the degree to which currently available game engines match up to these requirements. We also checked whether the engines we examined contained any of the following components: Graphics, Physics, Networking or Audio. Each of these factors is now discussed in more detail.

Extensibility

An extensible game engine allows programmers to add new code and functionality to the engine [7]. As the game industry changes and new programming techniques are developed, an extensible game engine can more easily be upgraded to include these changes. This keeps the game engine up to date and relevant to real world game development.

Proprietary engines can only be modified by those on the game engine development team, and as such students or educators using the engine would have to wait for the engine developer to update the engine to include new engine functionality. Another problem with proprietary engines is that game engine development companies have full control over the functionality included in their engines, and educators would not be able to make changes to the engine to suit the needs of their game development courses.

Reusability

Reusability is a fundamental characteristic of game engines, being one of the main reasons they were first developed [37]. Game engines should promote code reuse between projects as this lowers development time. A reusable game engine needs a feature list that covers the main areas of game development, so that code that handles low level tasks does not need to

be rewritten for each project.

Within the domain of game development, it is difficult or impossible to develop a system that can handle the requirements of all games, as there are many different types of games. There are game engines that are built for specific types of games, but these are, by design, limited to that type, and not usable as a general system for game development. Consequently, most available game engines try to provide the most useful subset of features for the game industry as a whole. It is then up to individual game developers to add in the functionality needed for their game. This allows the game engine to be reused across many different projects, as the most commonly used features are included in the engine.

Modularity

Object oriented programming, or OOP for short, is built around the idea of objects and their relation to one another [35]. A useful feature of OOP is the formalisation of object interactions through access modifiers, which control to what degree the object's internals can be accessed and modified. This allows the language to enforce how objects and their data members are accessed. This is a huge advantage in game development as it allows the programmers of a system to abstract the use of a component away from its implementation, allowing changes to the internals to not affect other code that uses that component. This is a very useful feature for a game engine, and specifically for a game engine intended for use in a game programming course. It allows modules to be written that can be easily added or removed from the engine.

Open Source

A piece of software being open source means that source code for the software is available, and that access to such source code is available at no cost to the user [1]. For a game development course, cost is an important consideration for any software. It means that beyond the cost savings, each student can have access to their own copy of the software for their own use. Proprietary software is in some cases restricted to a certain number of users, or restricted to only working on a designated number of computers. Furthermore, costs associated with licenses can be very high, especially if the course seeks to provide a license for each student.

Open source has another advantage, in that all source code is available, and can be changed to whatever extent is required. Open source is not a requirement for extensibility, but it greatly enhances a piece of software's extensibility. Development houses may release a free

version of the software for use by students and hobbyist developers, or they may release small sections of code for study, but neither scenario can support full extensibility, nor can it provide the benefits of full access to the source code.

Full access allows any part of the game engine to be available for study. This means that even if a certain area is not currently a subject of study within a programming course, access to any part of the source code allows the programming course to more easily integrate it into the teaching of that course.

Shallow learning curve

A game engine suited for a game development course should have a shallow learning curve, which reduces the time needed to get a simple game running in the engine. Once a simple game is running, it can be expanded by writing more game code, or extensions to the game engine itself, which is made possible by the extensible and modular characteristics of the engine.

Without a shallow learning curve, much time of the course would have to be devoted to learning the specifics of the engine which could dampen student interest. By providing an engine with a shallow learning curve, a student can realise their game ideas more quickly, and it makes experimentation with the game engine functionality easier. Advanced topics are also easier to introduce since students will be more familiar with the engine by that point.

Graphics Component

The graphics component manages 3D rendering and handles the visual aspects of the game [31]. In addition to this basic requirement, the engine should be able to load 3D models into the engine for use in-game. 3D graphics are a fundamental part of video and computer games and the game engine must support basic graphics functionality to be useful as part of a game programming course.

Physics Component

The physics component of the game engine allows the game engine to support simulation of physical interactions between objects, as well as support simulation of forces such as gravity [37, 10]. Without this, either there is no interaction between objects, i.e. a grenade explosion would not move a box that was sitting near it, or these interactions would be very limited.

Physics systems add a dynamic aspect to the gameplay experience. Without physics, an

explosion's effect on players or objects is pre-computed, i.e. all explosions have the same effect, or explosions do not affect players or objects beyond applying damage. With a physics system, inanimate objects can be affected by player actions, for instance, an explosion could throw a heavy crate across a room, injuring a player it hits. These systems often require a lot of extra processing time, but they add to the gameplay experience, so their cost is often justified.

Networking Component

Networking allows two computers to communicate with each other, and in the case of games, it allows two user to play the same game together [37]. The networking functionality in the game engine allow the game to create connections and to send data between computers. The game engine abstracts the connection to hide the lower level details of the actual system, and only provide methods that are needed for game networking.

Audio Component

Audio is a fundamental part of modern game engines. Sound effects such as firing a weapon, footsteps, or the hum of a car's engine would be played using the audio system of a game engine. Music is also another important part of modern game engines. Music sets an emotional tone for a scene and can dramatically affect people's experience of a gameplay sequence. The audio component loads the sounds and music in the game engine, so that it can be played when needed.

Pathfinding Component

Pathfinding in the context of video games refers to the system that determines the shortest path between two points. It is an important component of a game engine as it governs how computer controlled entities in a 2D or 3D environment move around the map to get to their destinations.

Some game engines include a pathfinding component while others leave it up to the game developers to provide the pathfinding functionality. A game engine suited for a game development course should include it because of the importance of this component. There are a few pathfinding algorithms available, but A* is a very popular pathfinding algorithms used in many video games today [2].

2.4 A Game Engine Taxonomy

As discussed in the previous section, a game engine suited for a game development course requires 5 features: Extensibility, Reusability, Modularity, Open Source, Shallow Learning Curve. It should also have at least the following components: Graphics, Physics, Networking, Audio and Pathfinding.

We constructed a taxonomy of game engines around these requirements. From the taxonomy we can determine the degree to which currently available game engines match up to these requirements. The game engines we selected for this taxonomy represent the different kinds of game engines that are currently available: simple game production programs, like GameMaker, small game engines that provide functionality in only a couple of areas, like Ogre3D or Irrlicht, and more complete game engines used by professional game developers, such as Unreal 3 or Source.

We evaluated the following game engines: Ogre3D, Irrlicht, Unity, Torque, GameMaker, Unreal 3, Source, and XNA. Each characteristic was marked yes or no, except in cases where the game engine only partially met the requirements for that characteristic. The learning curve was judged based on the size of the engine, and what type of development the engine is usually used for.

Engine	Extensibility	Reusability	Modularity	Open Source	Learning Curve	Graphics	Physics	Networking	Audio	Path finding
Ogre3D	Yes[1]	Yes	Yes	Yes	Medium	Yes	No	No	No	No
Irrlicht	Yes[2]	Yes	Yes	Yes	Medium	Yes	No	No	No	No
Unity	No	Yes	Yes	No[3]	Medium	Yes	Yes	Yes	Yes	Yes
Torque	Yes[4]	Yes	Yes	Partial	Medium	Yes	Yes	Yes	No	No
GameMaker	Limited[5]	Yes	Yes	No	Shallow	Yes	No	No	No	No
Unreal 3	No[6]	Yes	Yes	No	Steep	Yes	Yes	Yes	Yes	Yes
Source	Yes[7]	Yes	Yes	Partial	Steep	Yes	Yes	Yes	Yes	Yes
XNA	No	Yes	Yes	No	Medium	Yes	No	Yes	Yes	No

Table 2.1: *Game Engine Taxonomy*

From the taxonomy, it is clear that there is no game engine that fulfills all the requirements listed earlier. Reusability and modularity are common characteristics of game engines, and

[1] Ogre3D, Features, http://www.ogre3d.org/about/features
[2] Features, Irrlicht Engine, http://irrlicht.sourceforge.net/features
[3] http://unity3d.com/unity/faq.html
[4] Torque 3D — Products — GarageGames, http://www.garagegames.com/products/torque-3d
[5] Overmars M., Designing Games with Game Maker, Version 8.0, Yoyo Games, 2010
[6] http://www.unrealengine.com/partners/
[7] http://source.valvesoftware.com/SourceBrochure.pdf

were found in all the game engines evaluated for this taxonomy. Graphics functionality was also found in all game engines tested. It is a fundamental part of any game or interactive 3D system such as a simulator. As such, there are myriad 3D graphic libraries available.

Some engines include physics, networking, audio and pathfinding. Many engines concentrate on graphics and rely on other libraries to provide any additional functionality required. Professional level game engines used in the industry, such as Source or Unreal, include all the required functionality, which is common for professional game engines. Unity, a popular tool for independent game developers, also provides this functionality.

None of these engines, however, are open source. We find that the biggest shortfall in terms of our stated requirements are for the Extensibility, Open Source, and Learning Curve requirements. Extensibility and Open Source tend to go together, since if the game engine is open source, it can be extended. For professional game engines, the source is available in some cases, but it is normally quite expensive to get a license. Companies that make these engines rely on sales of their engines for revenue, and while this allows them to allocate resources to the further development and improvement of their engines, it means that their engines aren't open source.

A primary concern when deciding which game engine to use for educational purposes is the learning curve. GameMaker was designed to be easy to use and has limited scope, and thus was given a shallow learning curve in the taxonomy. Source and Unreal 3 are professional level engines and were given a steep learning curve in the taxonomy. All the others are somewhere in between. Only GameMaker meets the shallow learning curve requirement, but is not open source, and has limited extensibility.

The game engine proposed by this dissertation seeks to fulfill these requirements. A modular framework forms the central component of the engine, where modules performing the different tasks of a game engine connect to the module framework. This provides high level functionality for rendering, audio, pathfinding, networking and physics. Students are able to quickly get a prototype working, and it promotes fast iteration on design. The engine is open source, which also means it is extensible. Finally, as is common with commercial game engines, it is designed for reuse and inherently supports modularity.

2.5 Game Engine Components

We now examine the components that compose a game engine more closely. Each component performs a specific task required by the game engine

These tasks are

1. 3D Rendering
2. Audio
3. Artificial Intelligence
4. Networking
5. Physics

2.5.1 3D Rendering

Rendering is the process of taking a 3D scene, and turning it into a 2D image, called a frame [31]. This is achieved through the manipulation and display of rendering primitives. Rendering primitives are simple shapes that can be used to build more complex shapes and can be rendered efficiently. The process used to turn a scene description into a frame is called the rendering pipeline. A brief overview of the rendering pipeline is shown in figure 2.1. This is discussed in more detail in the design chapter.

There are two main categories of rendering processes: real time rendering, and offline rendering. Offline rendering is used to produce high quality frames for movies, visual effects or some other video. There is no real time requirement, and as such, the 3D scenes involved can be very complicated, and rendering a single frame can take hours. Real time rendering, on the other hand, requires the frame to be rendering in real time. This is commonly either 30 or 60 frames a second for games. Due to their interactive nature, games require a fast render speed, and thus aim to achieve real time rendering.

A renderer typically provides a rendering API. An API, short for Application Programming Interface defines how software components should interact with each other [8]. Using an API cuts down on the work that needs to be done when creating a system. The rendering API handles the low level tasks of rendering, and communicates with the graphics hardware through a graphics driver. Direct3D and OpenGL, the most common rendering APIs used

in game development, use triangle rendering primitives. The primitive used by the rendering API determines the kind of data it expects. The scene file used to load the scene contains a representation of the objects that make up the scene. They are usually a list of triangles that make up the objects of the scene.

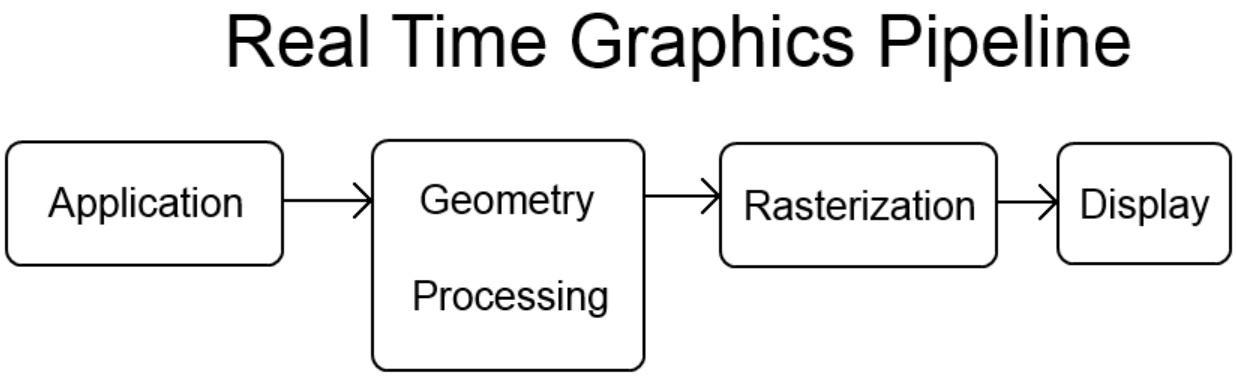

Figure 2.1: *The steps of the rendering pipeline. The application loads the 3D models and handles user input. Step 2 involves determining what to display based on the models in the scene, what the camera is looking at, and where it is. In step 3, rasterization involves producing a frame that represents the current state of the 3D environment. Once the frame is produced, it can finally be displayed on screen.*

Transformations of objects in a 3D scene, such as translations or rotations, are stored in the renderer as matrices. These 4x4 matrices operate on 3D points represented in 4D homogeneous coordinates [7]. The addition of a 4th 'w' coordinate allows one to represent linear transformation (translations, rotations, etc), as well as 3D to 2D projection operations by means of such matrices, which makes them convenient to use. They can also be concatenated together to aggregate multiple transformations into a single matrix.

Current rendering APIs also provide a lighting model that, while not completely physically accurate, is a good approximation, and can run at real time frame rates. This is referred to as the Phong lighting model. It includes three terms, *ambient*, *specular* and *diffuse*, which determine the characteristics of the light [31]. The ambient term is an approximation of the way light bounces around inside a room and provides general illumination. The diffuse term is an approximation of what happens when light hits a matte surface, and because of the roughness of that surface, the light is scattered in many directions. The specular term

accounts for how shiny objects have a small number of bright highlights on them. The size and intensity of these highlights depends on the specular term.

A texture is an image that can be "pasted" onto an object. The colour of a pixel on an object can be replaced or blended with a pixel from a texture (called a texel). Texture filtering is the process during rasterization of picking which part of the texture will be used in the final rasterised image [37]. The simplest form of texture filtering, point sampling, simply chooses the closest texel as the sample to be used. Other methods are more complex and computationally expensive; they include bilinear, trilinear, and anisotropic filtering [31].

2.5.2 Scene Graph

The Scene Graph is the system that represents the objects that make up the 3D scene, and their relationship to one another [7]. An example of how a scene graph can be used is shown in figure 3.2. A robot torso is added to the scene, and an arm can be added to the model on either side. Since the arm is added to the torso, when the torso rotates, the arms rotate with it. This simplifies use of the 3D models in a scene as users do not need to work out the positions and rotations of object or the positions and rotations of objects attached to other objects; these are handled automatically by the scene graph.

Another example shown is the tank with a turret and treads. The more complex models really see a benefit with the use of the scene graph as the scene graph handles all the positions and rotations. Working out the rotations and translations from scratch would be difficult and error prone. Sub objects, such as the turret, can have their own transformations applied to it, like rotation or translation, and the rotation and translation of the parent object can still affect the sub objects.

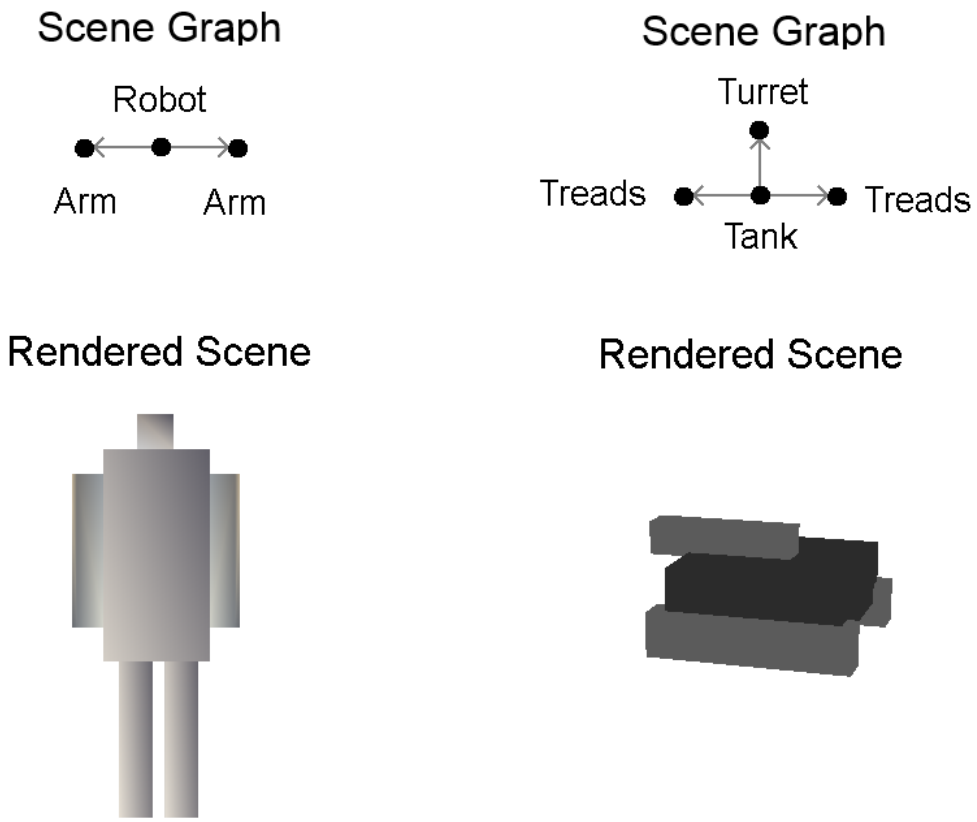

Figure 2.2: *Example of how the scene graph can be used. The first picture shows the nodes that make up the scene graph. The grey arrows indicate which is the parent and which are the child nodes. If the robot rotates, the arms rotate with it. The second picture shows how the objects look when rendered*

Each node has a parent (except the root node) and any number of children. The children can have children of their own. This nested structure allows scene graphs to accommodate complex scenes where nodes represents sub parts of objects that are themselves just a part of a larger object.

To organise the children of a node, each child is connected with the children next to it in the tree structure. The two siblings a node is connected to are referred to as its left and right sibling (the left and right have nothing to do with spatial position, but merely indicates the position of the sibling in the tree structure). So for example, if a node has three children, the first child of a node has no left sibling, but it does have a right sibling. The second child of a node has a left and a right sibling.

2.5.3 Audio

The audio component of the game engine manages the loading of sound files and ensures they are played at the appropriate time. Sounds used in games fall into two categories: sound effects and music. Examples of sound effects used in games are firing a weapon, footsteps, or the hum of a car's engine. Sound effects add to the experience of the game. In games that try to be realistic, they are used to attempt to match in the game what a player would experience in the real world. In more fantastical games, the sound effects are used to simulate what that world would be like.

Music sets an emotional tone for a scene and can dramatically affect people's experience of a gameplay sequence. Many game companies now employ experienced composers to help craft their music. The composer Christopher Tin won two grammys in 2009 for his album "Calling All Dawns". The album contained an arrangement of the song "Baba Yetu" which featured in Civilization IV, a strategy game from Firaxis Games [41].

Game engines tend to support a number of different audio file formats, such as WAV, MP3, or OGG. Many engines use third party programming libraries to handle the low level details relating to their audio. Examples include FMOD, OpenAL, irrKlang and DirectSound. Developers of these libraries sometimes directly support integration with a given engine. FMOD is one such example. Without this direct support, the engine developers may have to make changes to their engine to fully support a given audio library.

2.5.4 Game Scripting

In the context of game engines, scripting is the process of writing code in a scripting language, often an interpreted language, to form a script which can then be run inside a game engine [43]. These scripts allow programmers and designers to more quickly add or modify functionality in a game [43]. The term is used to differentiate between working on the actual game engine code itself, which is usually written in a compiled language, and scripting which uses the game engine and related tools once the game engine is ready for use.

One prominent example is Unreal Kismet [40], included in the Unreal Engine. Unreal Kismet is a visual scripting system that gives access to many different systems within the game engine. It is a useful tool for the whole development team as it can be used within the Unreal Engine's level editor, and doesn't require any coding. It allows level designers to

easily configure a level, giving them control over all aspects of the gameplay of the level. It also allows the artists to change the look of the level through visual controls. Programmers can define their own components to be used within Kismet, should extra functionality be required by the game.

Such a system has definite advantages not only to the programmers, but also to the whole team. It can speed up development by giving artists and level designers greater control over their level. It also benefits programmers as they can concentrate on work on the engine itself, and when additional functionality is required by artists or level designers, it can more easily by added to the visual scripting system, and given to the artists and level designers to use. However, this is a large and complex system, for use in professional development by experienced developers. Due to the size and complexity of this system, as well as the fact our game engine is aimed at students, a scripting system is beyond the scope of this research.

2.5.5 Game AI

Artificial Intelligence, or AI for short, controls or modifies the actions of all computer controlled entities.

A* pathfinding

Pathfinding is the process used to determine the shortest path between two points. It is an important component of game AI: it governs how these entities, often called agents in an AI context, move around the map to get to their destination.

A* is one of the most popular pathfinding algorithms used in games today [2]. A good pathfinding algorithm needs to balance two competing demands, that of moving towards a goal, and that of finding a good path, and minimizing costs such as algorithm execution time or distance [2]. A* algorithms work by combining the results from Dijkstra's algorithm which finds best paths, with a greedy algorithm such as Best-First-Search which always takes the most locally optimum movement [22].

Dijkstra is guaranteed to find a shortest path but it may take a while to compute the answer. Best-First-Search runs quickly, but is bad at avoiding obstacles. Game AI pathfinding should be good enough to add to the enjoyment of a game, and the algorithms used shouldn't dominate the processing time. A* uses the results of both algorithms but uses heuristics to change how much each algorithm contributes. A* can be configured to only use the results

from one or the other, or a mix of both. If more is known about the environments the agents are operating in, A* can be tweaked to produce a good path with a lot less processing time than using Dijkstra's algorithm.

2.5.6 Game Networking

Networking is a technology that allows computers to communicate with one another [37]. Game networking is networking used by games so that different players on different computers can play the same game together. Networking in this environment has specific requirements.

Game networking emphasises

1. fast feedback: a player performing an action and seeing a response to their action as soon as possible
2. reduced latency: minimising the time taken for the data representing the game at that moment, to pass from one computer to another and back[13]

Game state is the collection of data that define the game at that moment. When games are networked, the data needed to reproduce the player's current game state needs to be transmitted. This data is sent in a small, ordered unit of bytes called a packet. There are two types of networking connections: synchronous and asynchronous. A synchronous connection between two computers provides reliable, ordered transmission of packets. It handles the low level details of resending packets if they weren't received on the other side, and managing the connection.

The extra overhead is needed to maintain the reliable and ordered guarantee, however, it does slow down how fast packets can be transmitted. An asynchronous connection on the other hand, provides no guarantees on reliability or that packets arrive in the same order they were sent. Resending packets and handling those that were received out of order has to be handled by the application using the asynchronous connection. It provides however, a large reduction on overhead. Asynchronous connections [26] are, therefore, usually preferred because they emphasize reduced latency over reliably sending and receiving messages.

User Datagram Protocol, or UDP, is the protocol used for asynchronous connections. UDP is unreliable and does no error checking at the protocol level. The game must perform this error checking, but it can prioritise what messages must be sent. For games where each user controls one unit in a 3D space, the position of each unit is the most important data that

needs to be sent across the network. After that, other data relating to the user's game state can be transmitted.

Transmission Control Protocol, or TCP, is the protocol used for synchronous connections. It is reliable, and automatically handles ordering of packets. Due to the overhead in managing the connection, this protocol is not used in games, except in those cases where fast feedback is not required.

This prioritisation is very useful as some games have many objects in their 3D environments, and sending network updates for each of them would use too much bandwidth. The networking can be reconfigured to only update low priority items every other frame instead of every frame, and more advanced systems only update objects near players, and the other objects are updated a lot less frequently [3].

2.5.7 Game Physics

Game physics refer to the system that controls the physics of objects in virtual environments [37]. Before physics could be run as part of the game engine, the player and objects in the environment had no physical effect on each other. The visuals for explosions, for instance, were generated before-hand, and always had the same effect, such as damaging the player.

With real time physics, explosions can throw boxes into the air, which can then bounce off walls and smash into players causing damage. Physics systems handle movement caused by forces, and a collision detection system to determine when two objects are in collision [7], and handles what to do in the event of a collision. The physics calculation of force and movements are simplified to reduce processing time, and keep the gameplay at interactive frame rates. Collision detection also involves many optimisations and simplifications to reduce processing time [31].

2.6 Summary

Game engines are software systems designed to help developers to create games more easily. They handle much of the low level details such as 3D graphics, Physics, Networking, and AI. A game engine suited for a game programming course would have to be easy enough to

use for students, but also access to low level systems for use later on in the course, and in other more advanced courses.

This chapter reviewed requirements, such as the need for the game engine to be modular, and extensible, essential for keeping the engine current with the latest advances. We constructed a taxonomy based on these requirements and tested them against currently available game engines. We found that no game engine fully satisfied all our requirements, and that there is a place for a game engine that is both easy to use, but is also modular and extensible to allow the game engine to be used further on in the course.

Chapter 3

ModEngine Game Framework

This chapter explores the architecture of the game engine and examines the components that make up the game engine. The module framework of the game engine connects all the other components together. These components provide the functionality required by the game engine. The components that make up the game engine are: a 3D renderer, a scene graph, an audio system, a networking system, a 2D pathfinder, a physics system, a graphical user interface layer and a model and texture collection.

3.1 Design Patterns

Design Patterns describe common problems that occur in software development and codify the solutions for these commonly reccuring problems [25]. Each pattern has a name, describes when a pattern can be applied to a problem, and then provides the elements and their relationship with the other elements that make up the pattern's solution to the given problem.

For this dissertation, we concentrate on patterns that aid in the design of modular systems, as modules play a large part in the design of our game engine. We use the *facade* pattern [25] which provides a unified interface to a set of interfaces in a subsystem. This design pattern simplifies the use of a system, and reduces the learning curve by adding an interface to wrap a complicated subsystem.

Applied to our design, all components in the game engine provide a set of methods that need to be implemented by a plugin. This means that, despite the complexity of adding plugins that communicate with different libraries, through the common interface they can be added to the game engine without them affecting the student using the game engine.

3.2 Module Framework

The Windows operating system, developed by Microsoft, is the most widely used operating system across the PCs used by those who play computer games [6]. Our game engine takes advantage of this by using the Win32 API, which allows code to create and interact with windows in the Microsoft Windows operating system.

The game engine was written using C++, which is a common language used by game developers. It provides developers low level control of computer systems. This is a requirement for professional game developers as it allows them to more finely optimise the low level code used in professional level game engines.

As discussed in the background chapter, modularity is an important characteristic we want in our engine. By using C++, an object oriented programming language, we can enforce how objects and their internals are accessed. This allows us to abstract away details from the user, thus making a component easier to use, as only the required functionality of the component is exposed. Another advantage of this modular approach is that it allows us to define a set of rules that modules need to abide by. By using these rules, we can define what functionality a module must provide before it can be used by the engine.

Each task of the engine, some examples are rendering, sound or physics, is handled by a module. Using the modular architecture and by abiding by the rules specified for each type of module; a student can write their own module for a task and add whatever functionality they want to their module. As long as the module uses the rules defined for that task, it can interact with the engine. This allows modules to be added and removed, without having to rewrite any of the code in the rest of the engine whenever a module changes.

Figure 3.1 gives an overview of the engine. Each box represents a module which handles a task of the engine. All modules interact with the module framework which coordinates the work done by each module. The lines on the diagram indicates communication between modules. Some modules need information from other modules, such as the physics module and the scene graph module. When the physics simulation is updated, the scene graph must be updated as well. The interaction between the physics module and the scene graph is explained in more detail in section 3.10.

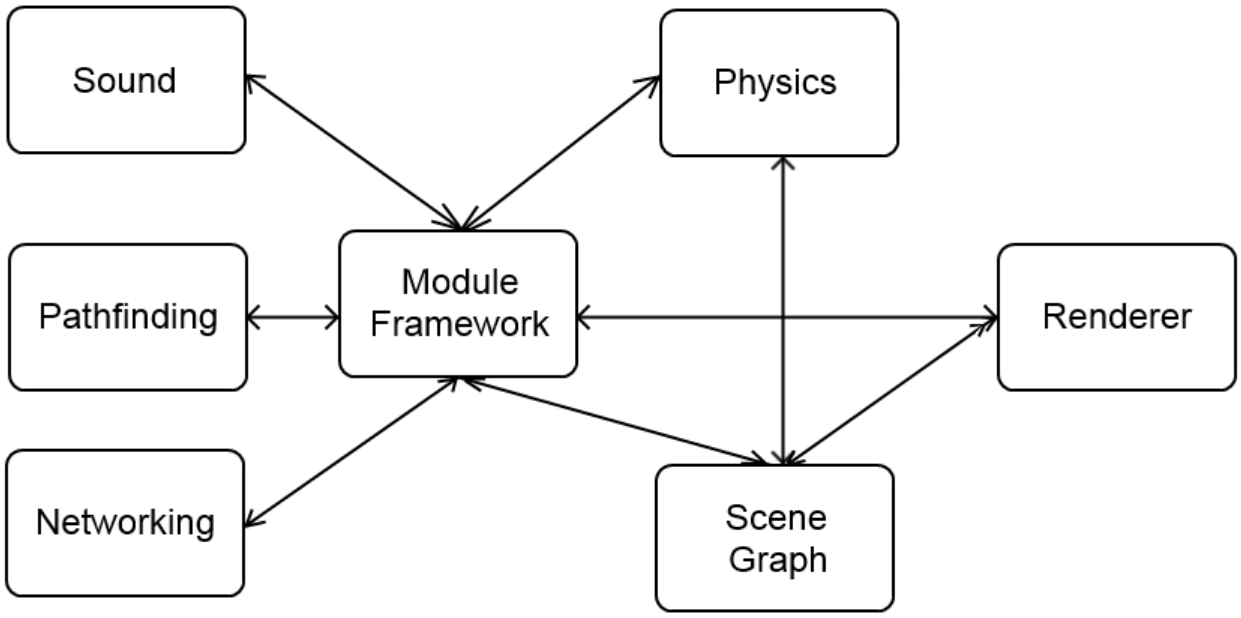

Figure 3.1: *Architecture diagram of ModEngine. Each box indicates a module in ModEngine. Each module handles a task, such as rendering or physics. Each module is explained in more detail in its own section. The lines indicate which other modules a module interacts with*

```
while game is running do
    if there are any Win32 messages to be handled then
        translate and dispatch message;
    end
    calculate time since last frame was rendered;
    if it is time to generate another frame then
        run the loaded game's game loop;
        if game is not paused then
            render a frame;
            update the loaded game's physics simulation;
            update the scene graph with results from the physics simulation;
        end
        if game network is running then
            if game is the host of the network game then
                receive a network packet as host;
                send packet to the loaded game to handle as host;
            else
                receive a network packet as a client;
                send packet to the loaded game to handle as a client;
            end
        end
    end
    if loaded game is finished then
        close down game
    end
end
```

Algorithm 1: Pseudo code for the game loop in the WinMain class

The game loop

The method `WinMain` in `WinMainClass` defines the entry point of the program. An overview of the `WinMain` method is given in Algorithm 2. Each game made using the game engine is made by setting its main game class to inherit from the `GameEngine` class. At the beginning of the `WinMain` method, a pointer from the `LoadGameEngine` function is stored. This pointer points to the current game to be loaded by the game engine. This simplifies loading games

into the game engine, as all that needs to be done to load a game, is to return a pointer from the `LoadGameEngine` function to the `WinMainClass`.

The console, used for any output that the game engine generates, is initialised here. Once the console has been initialised, the game engine, defined in the `GameEngine` class, loads all the components connected to it. The renderer is an important component as it handles the game window. The renderer runs the Win32 commands needed to create the render window.

A loop is then started, with pseudo code shown in Algorithm 1, which handles messages generated by the Win32 system. These messages pertain to window interaction, such as window activation, movement, resize or closing the window, as well as other forms of interaction, such as key press, mouse presses, mouse movement or mouse wheel movement. These messages are translated and dispatched as shown in Algorithm 1 to the method that handles these messages. Once the messages have been checked, the `gameLoop` method on the current game engine object is called, which iterates the game's systems.

The render method handles the graphical component of the game. We limit the frame rate of the renderer by only calling the render method a certain number of times a second, at a rate equal to the desired frame rate data member defined in the current game class. The render method produces one frame from the game and displays it on the window generated by the renderer.

The loop inside the `WinMainClass` also updates the physics system of the game engine as well as run the networking system. Once the `m_done` member variable on the current game class is set to true, the loop quits, and the game engine quits. The `m_done` member variable is set inside the game loop of the current game. The creator of that game determines when the game engine should quit, and then sets the `m_done` variable to true.

get game engine pointer;
load console;
load game engine components;
run game loop, refer to algorithm 1 for more details;
close down and release resources used by the game engine;

Algorithm 2: Pseudo code for the `WinMain` method in `WinMainClass`, which is the entry point of the program

3.3 Scene Graph

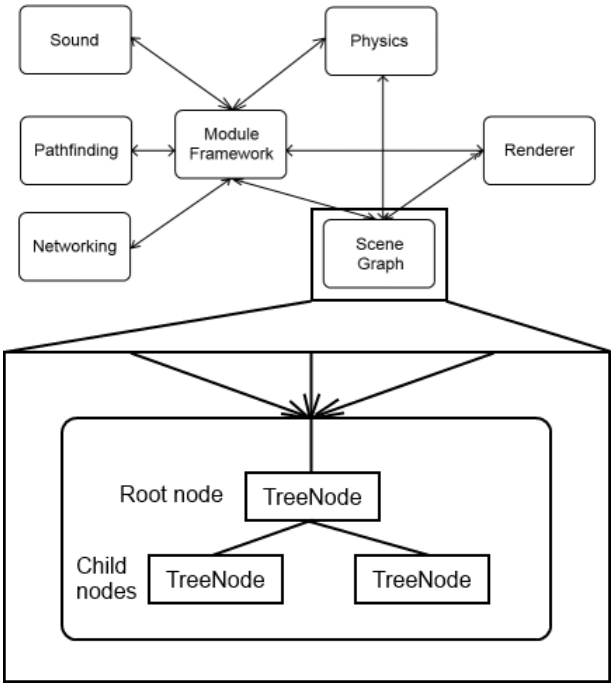

Figure 3.2: *The architecture diagram with the scene graph module highlighted. This section focuses on the scene graph module. The lines indicate that the scene graph module interacts with the module framework, the physics module, and the renderer*

The Scene Graph is the system that organises objects in a 3D scene. A scene graph is typically organised in a directed acyclic graph, or DAG. However, to simplify the use of the scene graph in particular, and the game engine in general, we decided to use a tree to organise the scene graph. In a DAG, a node can have a connection with any other node, as long as no subset of edges form a cycle. In a tree, a node can only have a connection with its parent node, and with its direct children.

DAGs are useful in game development as they allow an object to have a connection with, and thus affect the rotation and translation of, multiple objects. In a tree however, objects are only connected to their direct parent, and their children. Thus only changes in the parent's rotation or translation affect the object. In ModEngine, there is no way to have more than one object affect the rotation or translation of an object.

This simplifies the use of the scene graph as the student only has to use one command to

add an object to the scene graph, and the connections are handled automatically by the tree structure. In a DAG, a student would be required to first define all the objects needed, and then define the relationships between those objects. While restricting the scene graph to a tree structure reduces functionality, we feel this is justified as it supports our goal of creating a simplified game engine.

```
void function traverseNode(Node node)

currentMatrix = get matrix from the top of the stack;
push the current transformation matrix onto the top of the stack;
newMatrix = calculate matrix to represent the current node's transformations;
currentMatrix = concatenate newMatrix and currentMatrix;
render current node with the current transformation matrix;
if current node has a child then
    traverseNode(current node's first child);
end
pop the matrix on the top of the stack off;
if current node has a sibling to the right then
    traverseNode(current node's right sibling);
end
```

Algorithm 3: Pseudo code for the calculate transform method

Transformation matrices, which hold the translations, rotations and scaling in a scene, are stored in a stack, where the first matrix is at the bottom and the rest of the matrices are above it on the stack. Calculating the transformation matrices is a recursive process, which starts at the root node. A pseudo code overview of the process is given in Algorithm 3.

By concatenating the matrix that represents the current node's transformation with the matrix from the top of the stack, the transformations from the parent nodes propagate down to the children. This is a very useful property of scene graph as a user just needs to specify where an object is in relation to its parent, and the scene graph can determine where that object appears in space.

The scene graph is implemented in the `TreeNode` class. This class provides all the functionality needed for working with the scene graph. This functionality includes adding or removing nodes, accessing or updating transformations and storing the 3D model and texture information required by the renderer.

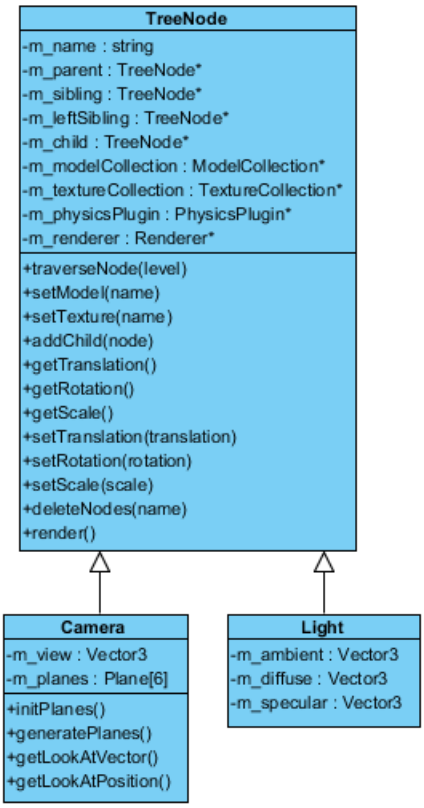

Figure 3.3: *A UML diagram showing an overview of the* `TreeNode`, `Light` *and* `Camera` *classes. Only the important member variables and methods are shown.*

3.4 Model and Texture Collection

The model and texture collections are two separate components, but since they are so similar, they are covered together. Figure 3.4 shows an example of how the model and texture collection can be used. This system was designed with the goal of simplifying loading of 3D models and textures. Loading 3D models and textures can get complicated. There are different formats for models and textures, and there are many different conditions to handle, such as the user requesting to load a file that does not exist. The model and texture collections handle these situations for the user. They contain a list of models and textures that can easily be added to, or retrieved from.

The user adds a model to the model collection, and the collection loads the relevant file,

and adds it to the collection. If errors should arise, such as specifying a file that does not exist, the model collection fails to load the model and does not add it to the list. The user is notified, and can then work out how to handle the error.

Loading the model manually without the collection and encountering an error could either lead to a crash, or attempting to render a model that didn't load properly will cause render problems. If a model can't be loaded, or encounters an error, then it isn't added to the collection, preventing the user from trying to use an invalid model. These issues pertaining to loading files apply equally to the texture collection system.

A benefit of these two systems is that they load all the relevant files at the game's startup. Whenever a model or texture is needed, it can be easily retrieved from the collection, rather than needing to load the file again. This speeds up loading of scenes. Furthermore, if a model is used by different 3D objects in the scene, then the same model from the collection is used, rather than requiring the same file to be loaded for each object.

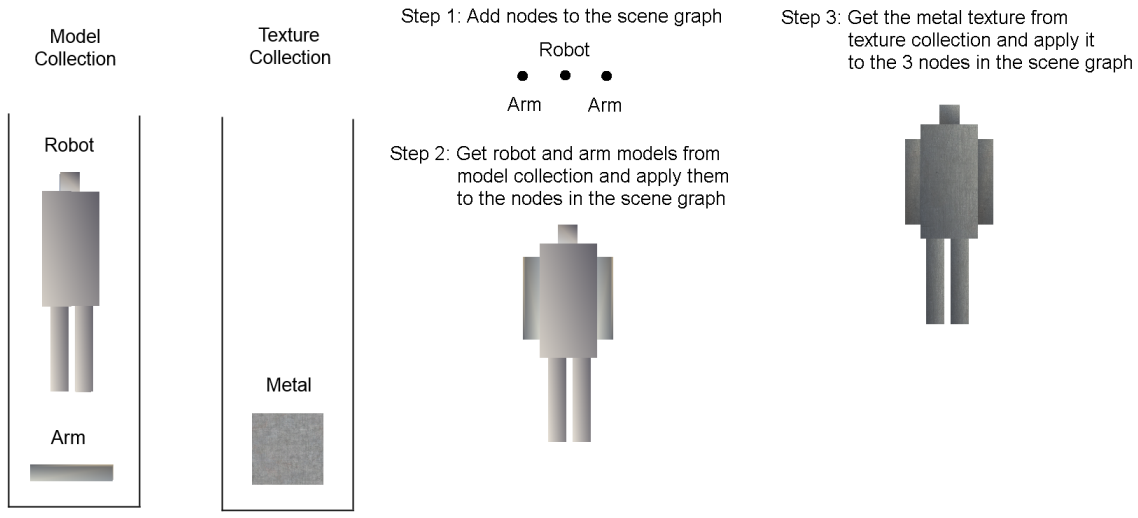

Figure 3.4: *Example of how the model and texture collection can be used. The box on the left shows the models that have been loaded into the model collection. The box next to it shows the textures that have been loaded into the texture collection.*

A disadvantage to this system is that a file reader must be written for each new type of texture of 3D model file format a user wants to use in a game and be integrated into the loading phase of the collection in question. The scene graph uses the model and texture collection for loading models and textures. Users of the system, therefore, have to use this

system if they want to load models and textures, but this aids in the goal of simplifying the use of the game engine.

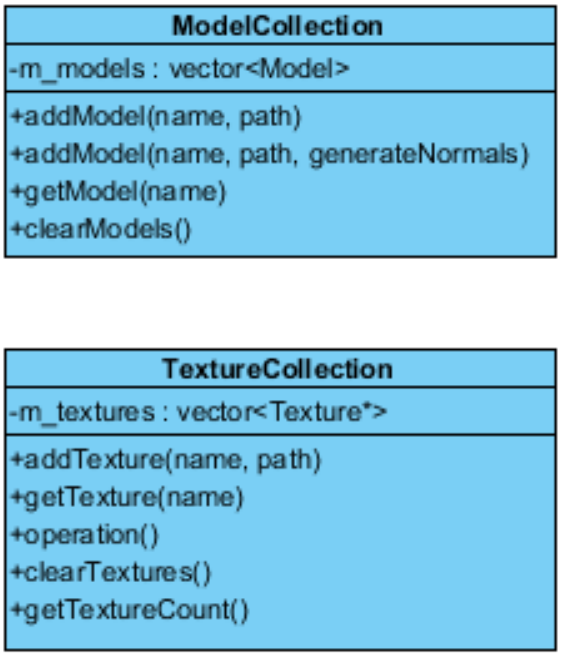

Figure 3.5: *A UML diagram of the model and texture collections.*

3.5 3D Rendering

The renderer is the component that handles the processing and display of the visual component of the 3D environment. The 3D game environment itself is controlled by the scene graph. This is a data structure that holds all the 3D models that make up a scene, and encapsulates any relationship between the models.

Using a scene graph, a model can easily be added to another model, so that, when the base model is moved, any attached models move along with it. This is explained in greater detail in the scene graph section.

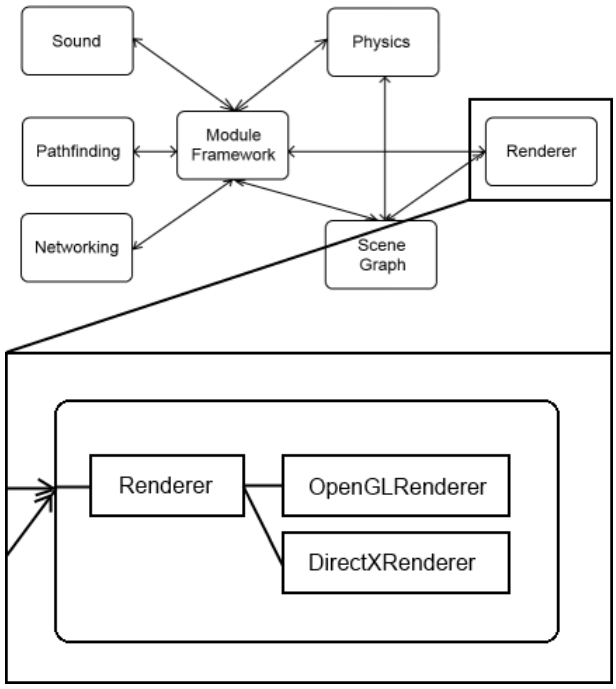

Figure 3.6: *The architecture diagram with the renderer module highlighted. This section focuses on the renderer module. The lines indicate that the renderer module interacts with the module framework and the scene graph*

The process used to turn a scene description into a frame is called the rendering pipeline. This pipeline executes on the graphics hardware and returns results through the graphics driver[34] to the rendering API being used. The rendering pipeline is made up of a number of stages, and each stage is explored in more detail next.

Rendering pipeline

Transformation: Each model in the 3D scene is made up of triangles. A scene is a collection of these models, so the coordinates are defined as being in the model space. This stage in the pipeline transforms all triangles from model space to world space, which is the coordinate space of the 3D environment being rendered. The application can also apply other transformations to models at this stage. For instance, if a small and large version of an object are required, the same model file can be used; just different transformations applied to them.

Per vertex lighting: Triangles in the 3D scene are shaded (or "lit") according to the specified location of light sources, as well as any other surface properties that may have been

applied to specific triangles. The stage is referred to as per vertex lighting, because lighting calculations are only performed for each vertex, and then interpolated across the triangle during rasterization The graphics hardware currently does this per vertex lighting, however modern graphics cards can perform per pixel (per fragment) lighting through the use of a shader program.

Viewing transformation: Objects are transformed from world space coordinates to camera space. This means all objects are transformed based on the position and orientation of the 3D application's virtual camera and thus produces a scene from the camera's point of view. This space can also be referred to as eye space.

Primitive generation: Now that transformation is completed, new primitives are generated from the primitives that were submitted to the rendering pipeline at the first stage. Graphics hardware with programmable pipeline can use a geometry shader at this point, which takes in primitives as input and can output as many primitives as are required.

Projection transformation: Primitives at this stage now need to be transformed from the camera space into a space called clip space. There are two main type of projection transformation: perspective projection, and orthographic projection. In perspective projection, objects further away from the camera are made smaller, where-as in orthographic projection objects retain their size no matter how far away from the camera they are. The specifics of the clip space used varies based on the rendering API used.

Culling: Any primitives outside of the camera frustum are now discarded, and primitives that intersect the frustum are culled so that only the part that lies within the frustum remains. Culling algorithms, for example occlusion culling, can be more sophisticated, in that they can take into account the complexity of the scene. It may even be able to cull primitives that are within the viewing frustum but are not visible because they are blocked from view by other objects in front of them.

Viewport transformation: At this stage, primitives are transformed from clip space into 2D window space. This space has the same dimensions as the user's window.

Rasterization: The 2D window space representation of scene now needs to be converted into what is called a raster format, hence the name rasterisation, and the values for each pixel need to be determined. A raster is a rectangular grid of pixels, which hold colour information for each point in that grid. At this point, each of the pixels are assigned colours based on the texture of the corresponding triangle (this process is called texture filtering),

or if a texture is not being used, it is assigned a colour based on colour values interpolated between vertices.

Display: The final pixel values can now be displayed on the user's monitor.

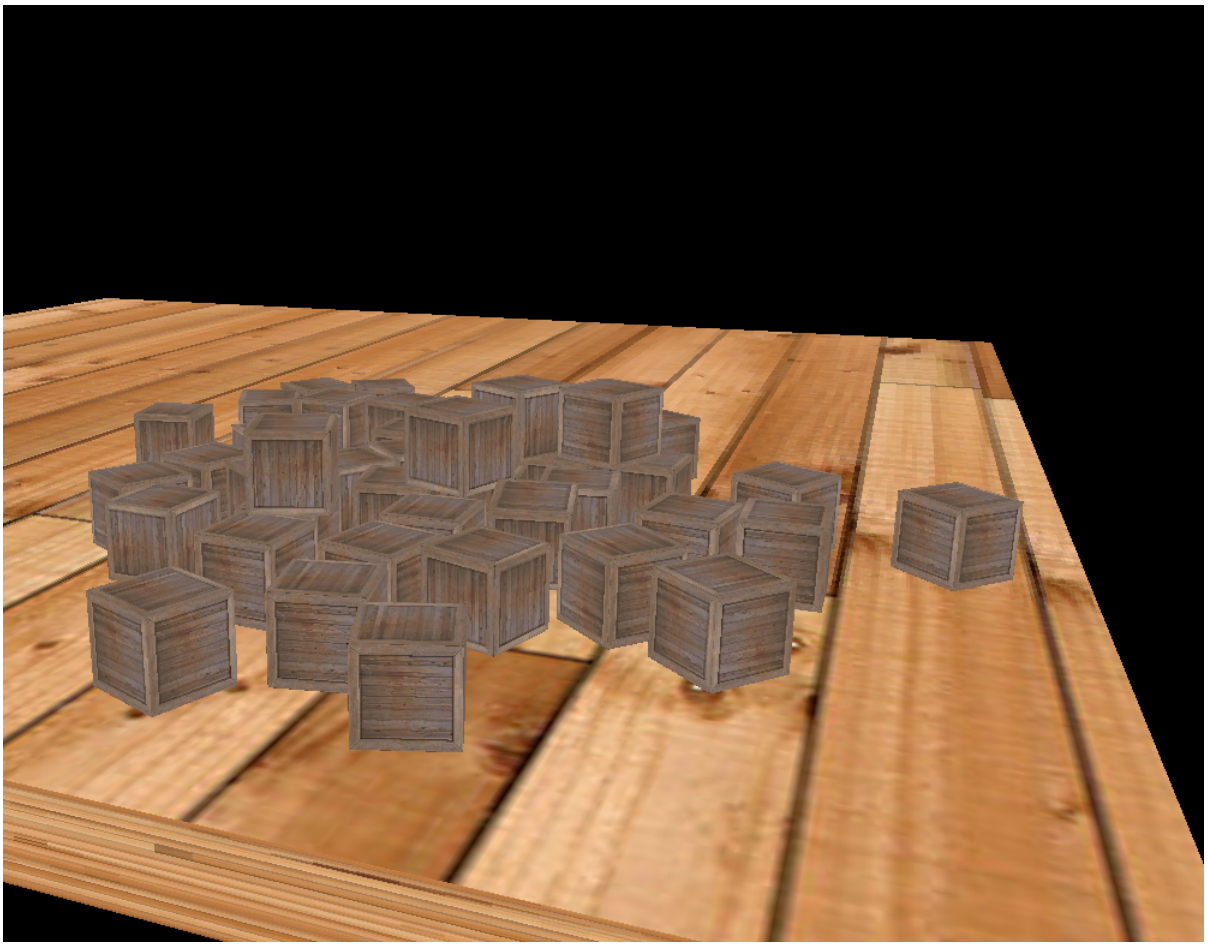

Figure 3.7: *Example of a rendered scene. This is a screenshot of the physics demo included with ModEngine, after a box was shot at a group of boxes and all the boxes fell to the floor, or landed on other boxes after they fell*

The rendering is done by the `Renderer` class. This class defines the methods that need to be implemented to make a new rendering class.

The benefits of a modular design, specifically relating to rendering, is that it decouples the rendering from the internals of the scene graph. These are complex systems, so limiting coupling is very important, not only from the perspective of code complexity, which affects how easy the code is to understand, but it also has a positive affect on stability and

maintainability.

The easier the code is to understand, and the more self contained each system is, the easier it is to maintain over time. As long as the scene graph presents data to the user of the system in the same way, another scene graph system can be swapped out with the current one and it wouldn't affect the renderer.

As discussed earlier, the model and texture collection contain a list of models and textures for use in the game. Users can easily add to, or retrieve from these collections. It simplifies loading and use of models in a 3D environment. Before a renderer can start rendering, it must load all the textures that are needed by the game.

For this task, it goes through each texture in the texture collection, and loads it in the manner specific to that renderer. Since all the textures have been loaded, any texture can be used at any point during the game. The user must simply set their 3D object to use a certain model and texture and the renderer renders their object using the specified values.

Figure 3.8: *Screenshot taken from the Zombie game included in ModEngine. A number of zombies are randomly placed around the maze, and they try to find the player, using the pathfinding system included in ModEngine. The player can either attack them, or avoid them, finding another route. The player has to get to the other side of the map to win*

Each rendering API also has its own lighting system, so at startup each renderer sets up the lighting according to the specific of that rendering API. The lights are defined in a `Light` class, which inherit from the `TreeNode` class, which is the class used to represent the objects in the scene graph. So lights can be added to the scene using the `Light` class. The renderers use these `Light` objects to define the position and ambient, specular and diffuse components of the light.

Various types of texture filtering is offered by both rendering APIs. Simple texture filtering is fast, but doesn't produce the best visual results. More complex texture filtering is slower, but produces better visual results. The renderer system in ModEngine supports nearest point, bilinear, trilinear and anisotropic texture filtering [31]. The renderer also supports 2D rendering. This includes displaying 2D text on the render window, as well as drawing various shapes, such as lines, squares and polygons. This 2D rendering is used to display the GUI elements that are a part of ModEngine.

To summarize, the renderer in ModEngine allows users to:

1. render models
2. apply textures to models
3. set up ambient, direction and spot lights
4. enable/disable lighting
5. enable/disable backface culling
6. change texture filtering
7. draw 2D text and shapes

3.6 OpenGL and DirectX Renderers

There are two popular rendering APIs: OpenGL and DirectX. This game engine supports both of them. The `OpenGLRenderer` class inherits from the `Renderer` class and handles OpenGL rendering, and the `DirectXRenderer` also inherits from the `Renderer` class and handles DirectX rendering. Each rendering API does things differently, so in each renderer, the render method is different. The render method is defined as a virtual method in the

`Renderer` base class, and implemented in the `OpenGLRenderer` and the `DirectXRenderer` class. A UML diagram of the renderers is given in figure 3.9.

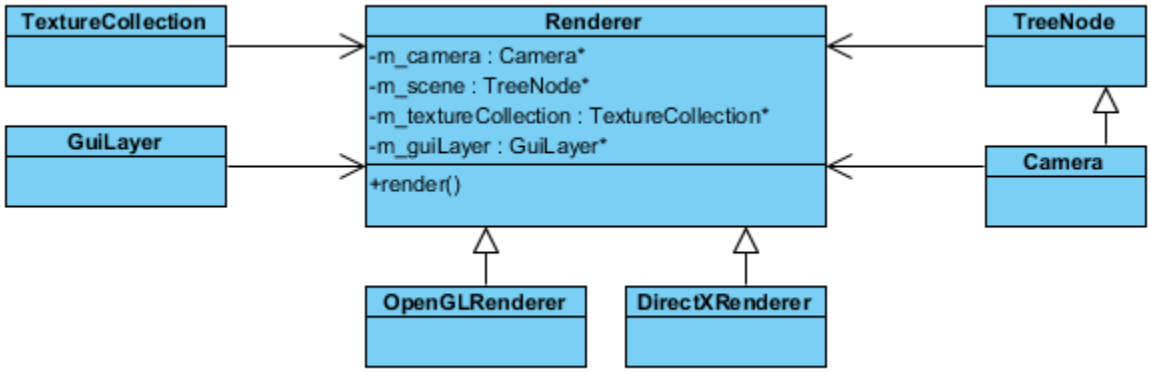

Figure 3.9: *A UML diagram showing an overview of the* `Renderer` *class and associated classes. Only the important methods and member variables are shown. The* `OpenGLRenderer` *and* `DirectXRenderer` *classes inherit from the* `Renderer` *class. The* `Renderer` *class also has access, through pointers, to the camera, scene graph, texture collection, and the GUI layer*

The render method

An overview of the render method is given in Algorithm 4. The render method first needs to setup the transformations. The transformation matrix is first initialised to the identity matrix. The scene is viewed through a camera, so the camera's position and direction is used to generate a view matrix that is applied to the respective renderer's transformation matrix. More on scene graph in section 3.3, but as a quick overview, the scene graph is made up of nodes that represent nodes, connected together in a tree structure.

if *scene graph has not been initialised* **then**
  return from the render method;
end
clear the display buffer;
set the current transformation matrix to the identity matrix;
calculate the camera's look at vector;
traverse the scene graph, starting at the root node, more detail in algorithm 3;
update the frames per second counter;
draw Graphics User Interface layer;

Algorithm 4: Pseudo code for the render method

Any nodes that are attached to other nodes become children of that node. Any nodes that are not attached to other nodes, and are thus "free standing" are children of the root node. The scene graph starts with a root node, and nodes are then added to it to build up a scene.

Rendering the scene involves starting at the root node, and the traversing through the tree by visiting each node and rendering it. The default functionality renders a node by rendering the model attached to that node, using the texture that has been applied to the node's model. This functionality can be overriden by implementing the render method in a class that has inherited from the `TreeNode` class.

The last thing that the render method does is draw the Graphical User Interface (or GUI for short) layer. The GUI layer is drawn using 2D graphics method defined in the `Renderer` class and implemented in both the `OpenGLRenderer` and the `DirectXRenderer`. These 2D graphics method can also be used to do custom 2D graphics for a 2D game. They can also be used to augment 3D graphics with 2D overlays, such as drawing a status bar below a game agent.

Handling transformation matrices

OpenGL handles its matrices internally, so the `OpenGLRenderer` uses commands to modify OpenGL's matrices according to the current node in the scene graph. DirectX puts the work of handling matrices on the programmer using it, so the `DirectXRenderer` stores its own matrices, and uses them when traversing the tree and performing the transformations.

Rendering scene graph nodes

By default, a node is rendered if it has a model and texture assigned to it. These models

and textures come from the model and texture collection. These collections were discussed in more detail earlier in this chapter. Nodes are rendered using the render method on the `TreeNode` class, which represents a node in the tree. This method can be overriden in a class that has inherited from the `TreeNode` class.

It is possible to create a game without making classes that inherit from the `TreeNode` class, but it can be a useful way to organise code. All the logic for a player can be put into one class, and the logic for enemies can be put into another class. Overriding the render method also allows the programmer to add additional functionality to the render process of that node. Adding 2D elements to a node, such as health bar, or icons denoting the node's type, or the team it's on in a competitive game, can be added to the render method.

2D games can be made easily with the scene graph. There is a lot of basic functionality built into the game engine, such as draw lines and shapes, and setting the colours of those 2D elements. A node that represents a 2D object would just need to inherit from the `TreeNode` class, and override the render method and then use the 2D methods built into the renderer.

Each of the two rendering APIs support texture filtering but need to be configured differently. This configuration is done in the implementation of the respective renderer class. This design allows the renderer to be changed by simply changing what renderer module the module framework uses for the rendering task.

Plugin architecture

The plugin architecture enforces the way the components communicate with the module framework and with each other. A different renderer module may have a different internal implementation, but since it works through the plugin architecture, any renderer communicates with the framework and other components in the same way.

The modular approach and plugin architecture abstracts the complexity of the rendering away from the user, which is especially important for students who have just started a game development course. The plugin architecture, since it requires both renderers to provide the same level of functionality and makes them communicate with the framework in the same way, simplifies using the renderer for more custom tasks.

Examples include overriding the render method, which is needed for adding 2D or 3D overlays for instance, or to add custom graphical effects to an object. This design also allows the game engine to support the two most widely used rendering APIs, and through the plugin

architecture allows the game engine to support future rendering APIs. The modular approach also makes changing the renderer being used very easy, simplifying the use of the game engine.

3.7 Audio

The audio component manages the loading of WAV format sound files and ensures they are played them at the appropriate time. DirectSound, a sound API developed by Microsoft, is used by the audio system to handle the low level details of storing and playing audio data. With the modular design of the game engine, the `SoundPlugin` class defines method that need to be implemented in a class that inherits from it.

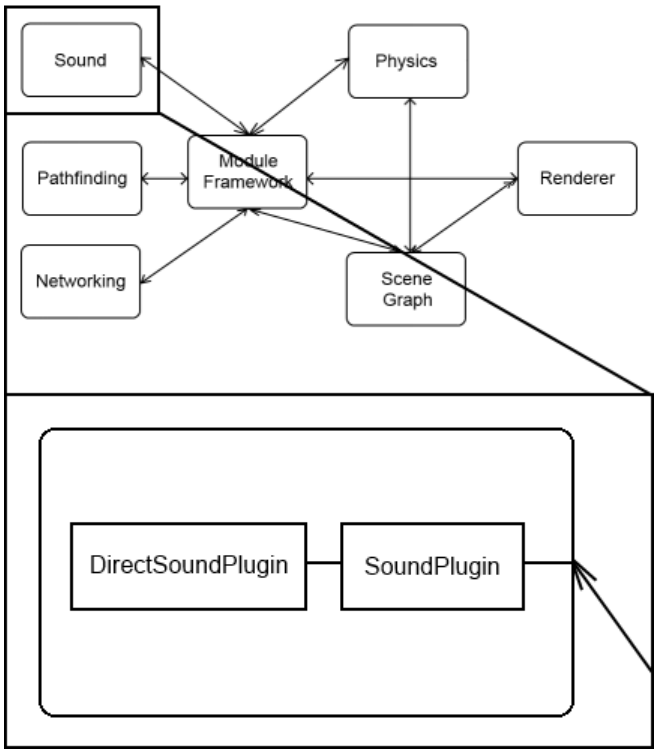

Figure 3.10: *The architecture diagram with the sound module highlighted. This section focuses on the sound module*

This modular design allows the game engine to interact with any sound library, as long as a plugin has been written for it. The goal of our game engine is to provide simple functionality, so the audio plugin allows the programmer to load an audio file, to play an audio file or to set the volume.

DirectSound works on attenuation. This requires that, as the input volume, a number between 0 and 1 (that represents a percent), needs to be translated into a value that indicates how to attenuate the sound to get the desired volume. This makes the `DirectSoundPlugin` a bit more complicated, but the advantage of the modular system is that this complexity is hidden within the `DirectSoundPlugin`. A UML diagram of the audio classes is provided in figure 3.11.

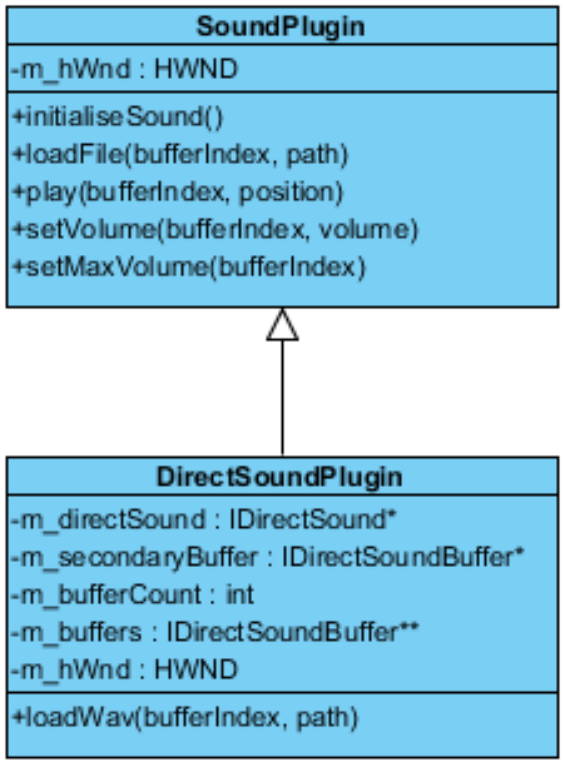

Figure 3.11: *A UML diagram of the* `SoundPlugin` *and* `DirectSoundPlugin` *classes.*

3.8 Networking

Networking, as discussed in the background chapter, is the technology that allows computers to communicate with each other. An overview of the networking system is given in figure 3.12. When games are networked, the data needed to reproduce the current state of the

player's game needs to be transmitted. This data is sent in a small, ordered unit of bytes called a packet. Packets are encoded in a `NetworkPacket` class, which stores the source IP address, destination IP address, port as well as the most important part, the data of the packet itself. The `NetworkPacket` objects are sent and received by the `NetworkPlugin` class. A UML diagram for the networking system is shown in figure 3.13.

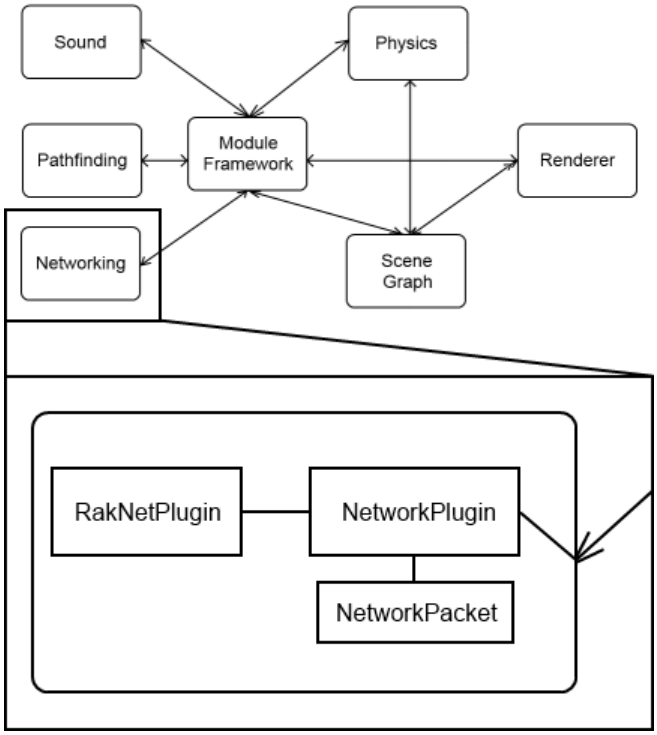

Figure 3.12: *The architecture diagram with the networking module highlighted. This section focuses on the networking module*

Integrating a networking library into the game engine requires a network plugin to be written. The `NetworkPlugin` class defines the methods that need to be defined to make a network plugin. Setting up a client or server connection requires different code, so the `NetworkPlugin` class separates the client and server tasks, such as initiating a connection, receiving and sending packets, into client and server methods. It is then the task of the network plugin to implement the different methods.

The class that inherited from the `GameEngine` class can then override a method that handles network packets and use that add network functionality to that game. The advantage of the modular design is that it allows network plugins to be easily added and removed. The

network plugin included with the game engine is built with RakNet, a popular networking library.

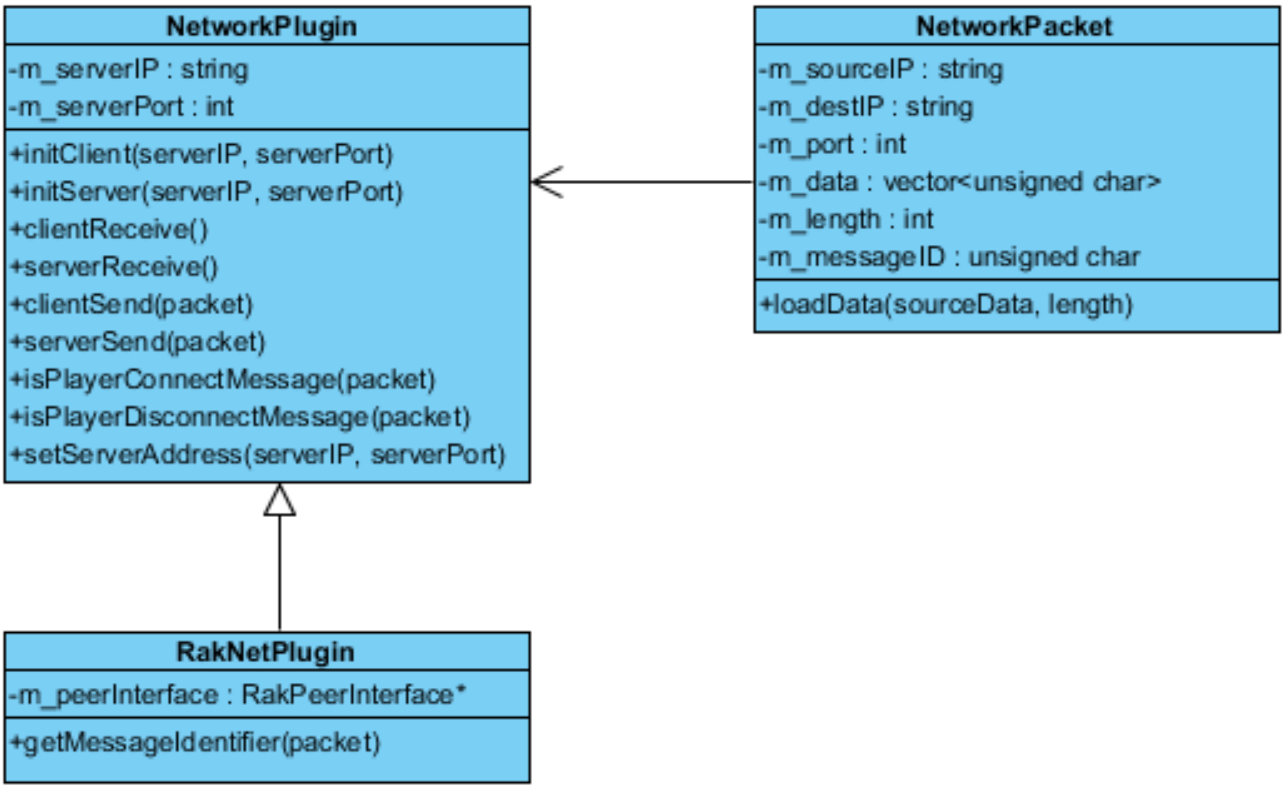

Figure 3.13: *A UML diagram of the* `NetworkPacket`, `NetworkPlugin` *and* `RakNetPlugin` *classes.*

The networking system is run in the same loop as the `gameLoop` method is run, in the `WinMainClass` file. This is described in pseudocode in Algorithm 5. It is run outside of the `GameEngine` class to abstract some of the detail away from the user of the game engine so they can concentrate on their game. This allows the programmer to call the network methods to start and close connections and handle network packets more easily than having to perform all the low level calls on RakNet [38], or other networking library themselves.

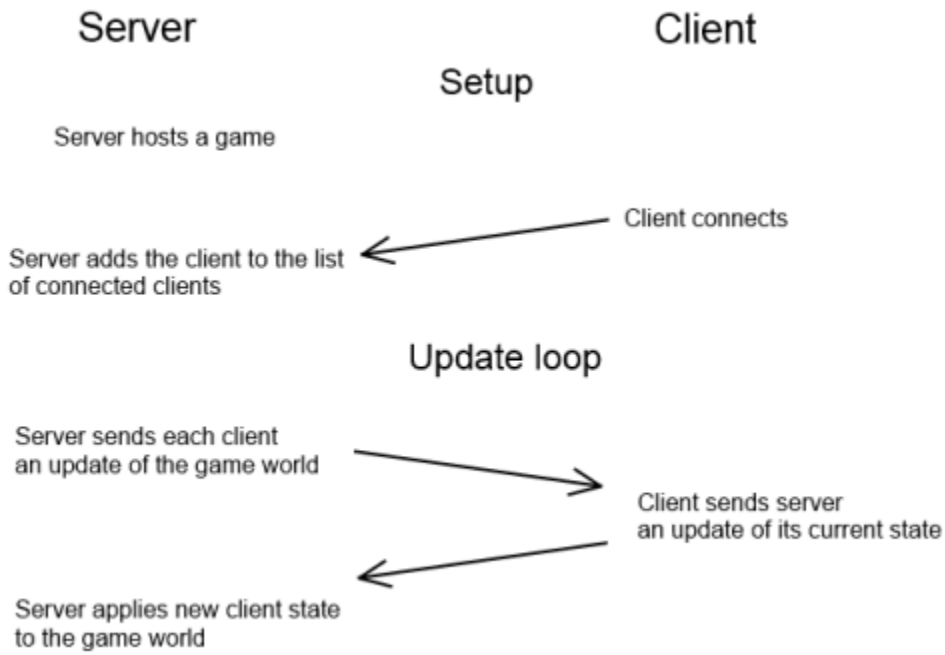

Figure 3.14: *Overview of network setup and the update loop. Arrows show the the process flow.*

```
if game network is running then
    if game is the host of the network game then
        receive a network packet as host;
        send packet to the loaded game to handle as host;
    else
        receive a network packet as a client;
        send packet to the loaded game to handle as a client;
    end
end
```

Algorithm 5: Pseudo code for the networking section of the game loop in `WinMainClass`

3.9 Pathfinding

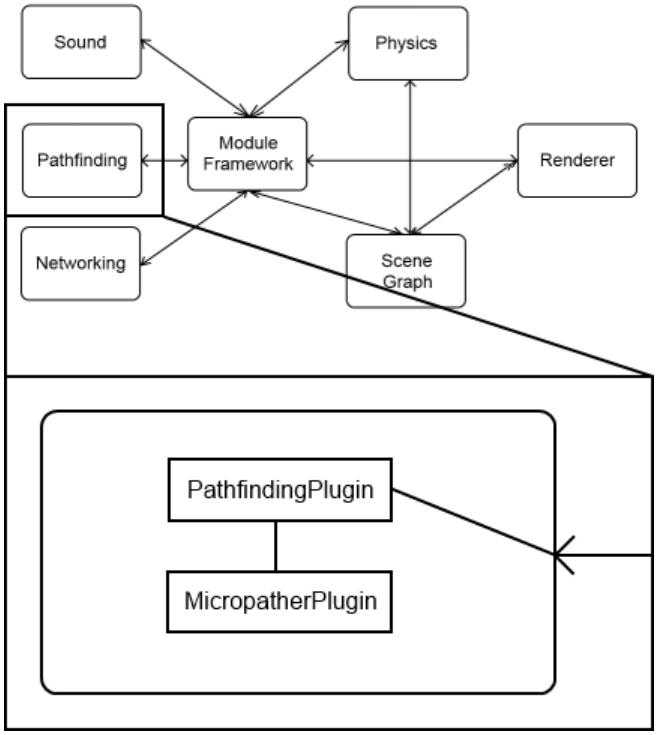

Figure 3.15: *The architecture diagram with the pathfinding module highlighted. This section focuses on the pathfinding module*

Pathfinding in the context of games, usually refers to the process used to determine the shortest path between two points. It is an important component of game AI: it governs how entities in a 2D or 3D environment move around the map to get to their destinations. The `PathfindingPlugin` class defines the methods that need to be implemented to make a pathfinding plugin. The plugin included with ModEngine uses the Micropather pathfinding library [29] to perform the pathfinding.

The pathfinding system in the game engine is designed to perform 2D pathfinding. The pathfinding plugin accepts a 2D map of characters, with a width and height, and stores the map internally. The A* algorithm, one of the most popular pathfinding algorithms [2], is used by Micropather to perform the pathfinding.

The pathfinding plugin has been designed to be easy to use, so the programmer only needs to set the starting position, desired destination, and then call the `findPath` method that

performs the pathfinding. The pathfinding plugin translates the data inputted into the form required by Micropather, which then performs the pathfinding and returns the path, which is then converted by the pathfinding plugin into a list of 2D vectors.

Once you have run the `findPath` method, the list of 2D vectors can be used to define a path for entities in the game. A overview of the use of the pathfinding module is given in pseudocode in Algorithm 6. The entity follows the path by starting at the first point and then moving from there to each point in the path generated by the pathfinding plugin.

initialise the pathfinding plugin;
set start position to the inputted location;
set end position to the inputted location;
find path;

Algorithm 6: Pseudo code for the use of the pathfinding plugin

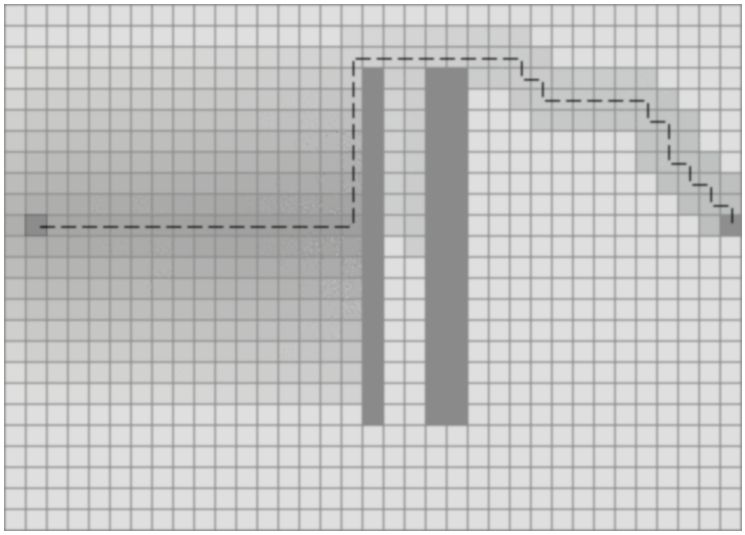

Figure 3.16: *Example of pathfinding done with A*. The dark gray block on the left indicates the starting point, and the dark block on the right indicates the goal. Any other node that has been filled is a node that was tested by the algorithm while finding the solution. The dotted lines indicate the solution generated by the algorithm*[1].

[1]http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

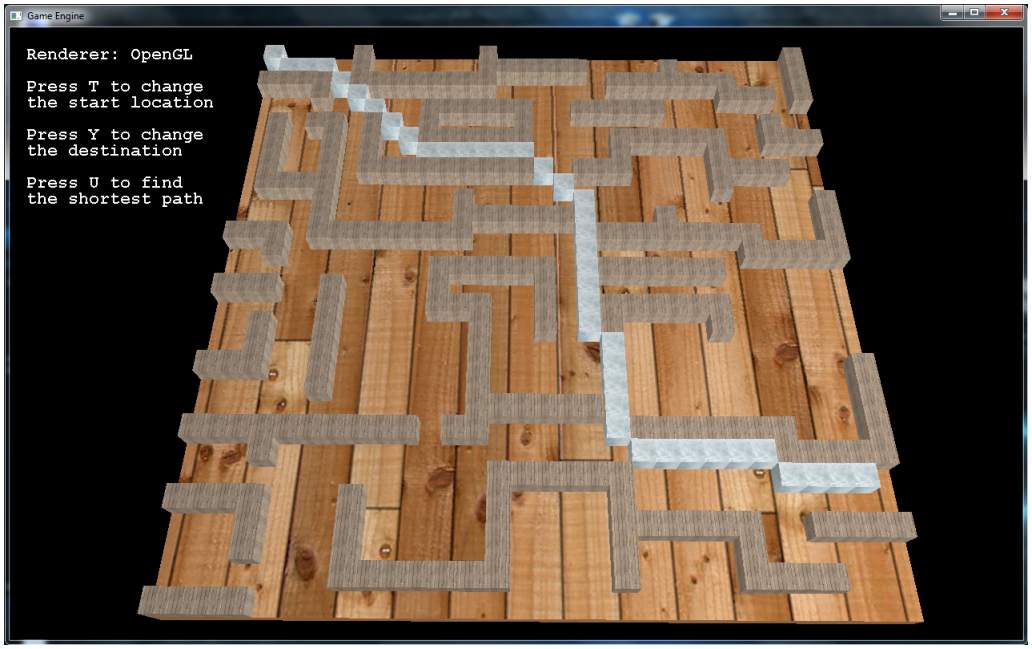

Figure 3.17: *Example of pathfinding being used in ModEngine. This is a screenshot from the pathfinding demo included with ModEngine*

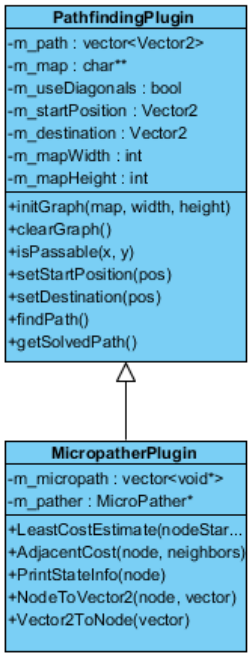

Figure 3.18: *A UML diagram of the* `PathfindingPlugin` *class and the* `MicroPatherPlugin` *class.*

```
create the open list;
create the closed list;
add the starting node to the open list;
while open list is not empty do
    find the node with the lowest f in the open list;
    remove current node from the open list;
    for each neighbour do
        if neighbour is the goal then
            destination has been reached;
        end
        neighbour.g = (current node).g + distance from current node to neighbour;
        neighbour.h = distance from neighbour to goal;
        neighbour.f = neighbour.g + neighbour.h;
        if a node with the same position as neighbour is in the open list and has a lower F
        than neighbour then
            skip this neighbour;
        end
        else if a node with the same position as neighbour is in the closed list and has a
        lower F than neighbour then
            skip this neighbour;
        end
        else
            add the neighbour to the open list;
        end
    end
    add current node to the closed list;
end
```

Algorithm 7: Pseudo code for the A* algorithm

3.10 Physics

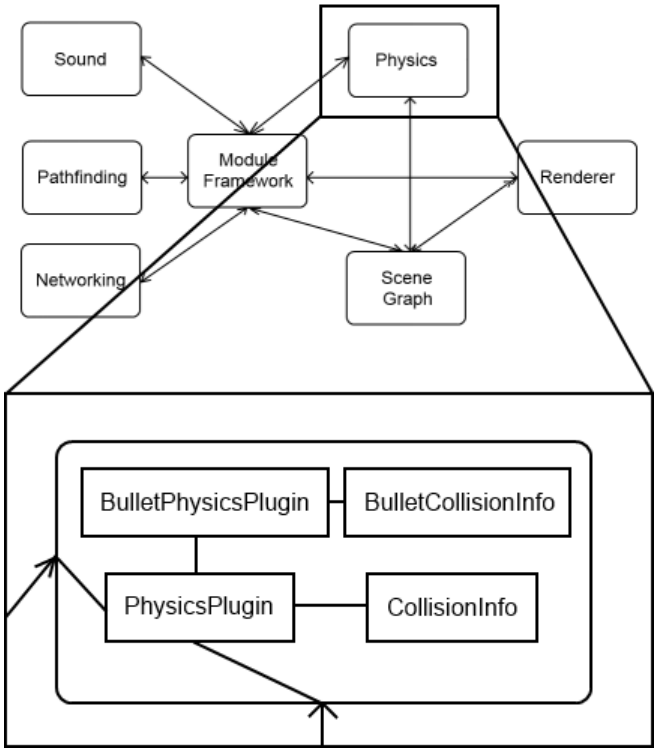

Figure 3.19: *The architecture diagram with the physics module highlighted. This section focuses on the physics module. The lines indicate that the module interacts with the module framework and the scene graph*

Game physics refer to the system that controls the physics of objects in virtual environments [37]. For our game engine, we want to add simple physics functionality that allows students to easily a physics simulation to their game. The physics system in ModEngine is configurable, allowing a student to add individual objects to the physics simulation, or even to not use it at all, if their game does not require it.

The mass of objects, and the friction they have with surfaces in the environment can be easily configured by using methods on the physics plugin. This allows students to have access to physics functionality, without them having to struggle working with and integrating a third party library. The physics plugin architecture defines a set functionality that needs to be implemented by a physics plugin.

Through the plugin architecture any physics library can be integrated into the game engine,

as long as a plugin is written for the physics library that allows it to interact with the module framework. This is very useful for the future as new physics libraries that are made can subsequently be integrated into the engine a lot more easily.

We have looked at the physics systems used in games and decided on a subset of features that are useful for users of a simplified game engine. Game engine typically support a number of physics functionality such as:

- Adding objects to the physics simulation
- Changing the velocity and position of objects
- Support for various collision shapes (the shape that is used as a simplified representation of an object used to perform the collision detection)
- Support for gravity
- Support for different types of forces (such as wind)
- Support for attractors, which are objects that attract other objects to them, the same way a magnet would

We chose the most important and fundamental functionality to add to our physics plugin architecture: objects can be added to the physics simulation, the gravity can be changed and the velocity and position of objects can be updated. The rest of the commonly available physics functionality offered by other game engines was out of scope of this thesis, but can be added through the modular framework provided by ModEngine.

The physics system stores the translation and rotation data for each of the objects that has been added to the physics simulation. During each iteration of the main loop, the data for each object in the simulation is read and applied to the objects in the scene graph. This keeps the visual component of the game up to date with what is happening in the physics simulation.

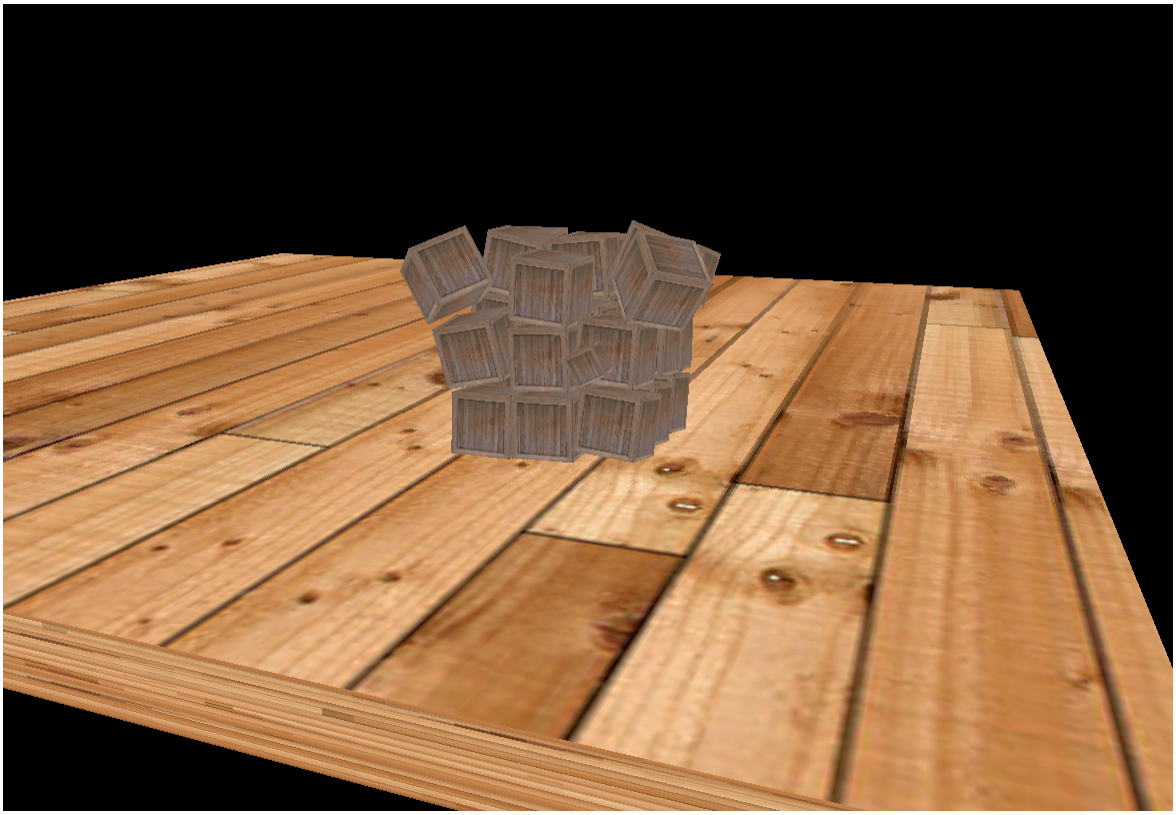

Figure 3.20: *A high speed object hitting a group of boxes. This is a screenshot from the physics demo included in ModEngine*

The `PhysicsPlugin` class defines the method that needs to be implemented to make a physics plugin. The physics plugin included in the game engine uses the Bullet Physics library, implemented using the `BulletPhysicsPlugin` class, to perform the physics simulation. The `PhysicsPlugin` allows the programmer to add a node in the scene graph to the physics simulation. A UML diagram of the physics system is given in figure 3.21.

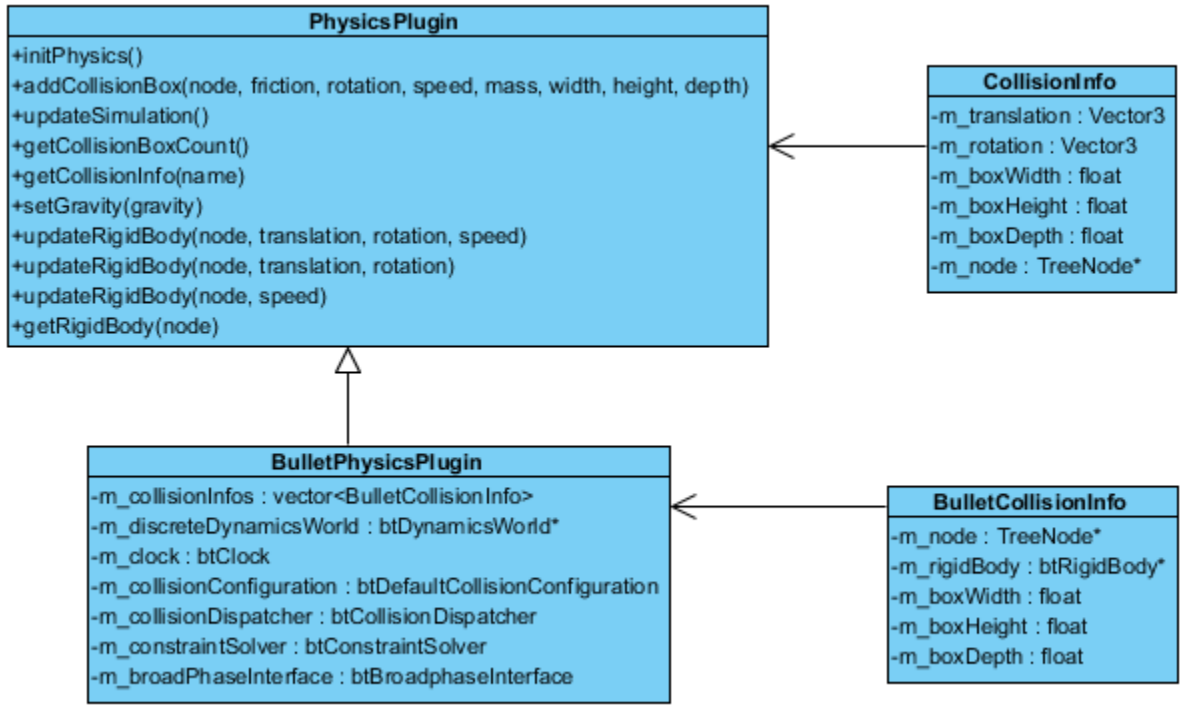

Figure 3.21: *A UML diagram of the physics system.*

run the loaded game's game loop;

if *game is not paused* **then**

render a frame;

update the loaded game's physics simulation;

update the scene graph with results from the physics simulation;

end

Algorithm 8: A subset of the loop run in `WinMainClass`, showing where the physics simulation is updated and results passed to the scene graph. The full pseudo code description of the loop can be found in the Module Framework section

When a scene graph node has been added to the physics simulation, the physics simulation takes control of the nodes rotation and translation. The rotation and translation can be subsequently changed in the scene graph, but the new position and rotation need to be updated by calling the method on the physics plugin which updates the internal state of the physics simulation.

The physics simulation is updated by calling the `updateSimulation` method, which is run in the same loop as the render, game loop and network methods. Algorithm 8 describes this in pseudocode. The gravity setting used by the physics simulation can be changed by calling the `setGravity` method on the `PhysicsPlugin` class.

The physics system and the objects in the scene graph that are a part of the physics simulation are updated in the main loop inside `WinMainClass`. This abstracts away some of the complexity of getting data from the physics simulation and applying it the nodes in the scene graph, as student do not need to change or work with `WinMainClass`. This greatly simplifies the use of the physics system since to add an object in the scene graph to the physics simulation only requires one line of code, and all the necessary work to add an object to the physics simulation is handled by the game engine itself.

3.11 Graphical User Interface Layer

The Graphical User Interface (or GUI for short) layer allows users to add 2D elements, such as labels and buttons to their game window. These elements make up what is called the Graphical User Interface. 3D rendering is used for rendering the game environment, but any 2D element would be added to the GUI layer. The renderer first renders the 3D environment, and then renders the GUI layer afterwards, overwriting the display buffer in the parts used by the GUI layer, so that it always appears on front of any object in the 3D environment.

GUIs are used for status displays, such as displaying the player's health, or the player's point total. It can also be used for information, such as describing the game keys when they are needed. Buttons are also a part of the GUI system in this game engine. Buttons are useful for providing the user with a simple means to executing commands outside of the keyboard button presses.

The GUI layer is implemented in the `GuiLayer` class. Controls, such as buttons and labels, are derived from the `Control class`. Labels are implemented in the `Label` class, and buttons in the `Button class`. A UML diagram of the GUI system is given in figure 3.22.

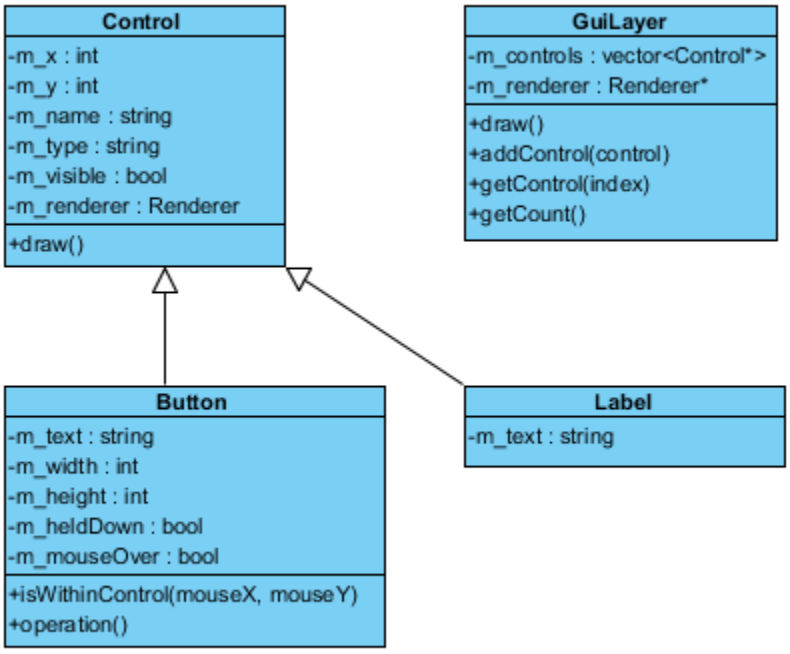

Figure 3.22: *A UML diagram of the graphics user interface system.*

Figure 3.23: *An example of a Graphical User Interface in ModEngine. This is the interface for the physics demo*

They provide the user with additional options, such as changing the texture filtering options in game. They can also be used contextually, only showing up when they are needed. A chest is one example, two buttons can be shown, one to look inside the chest, and another to simply take everything inside it.

3.12 Summary

In this chapter, we discussed the design principles and implementation details of ModEngine's central module framework and the components that make it up. We concentrated on a modular and simplified design that would allow students to quickly get started using ModEngine. This would allow them to make more changes to the functionality of the engine once their proficiency with the engine had grown.

Each task of the game engine is handled by a different module, which connects to the game engine through the module framework. The game engine handles the following tasks: Rendering, Audio, Networking, Pathfinding, Physics and Graphical User Interface. The plugin architecture enforces the way the components communicate with the module framework and with each other.

A new module for a task may have a different internal implementation than an existing module for that task, but since it works through the plugin architecture, any module for that task communicates with the framework and other components in the same way. The modular approach and plugin architecture abstracts the complexity of the task in question away from the user, which is especially important for students who have just started a game development course. The modular approach also allows students and other users of the game engine to easily change which modules the game engine is using and thus easily change the functionality of the game engine.

Chapter 4

Testing Methodology

4.1 Aim

A simplified game engine should make exploring game development concepts easier and simplify implementing the code needed for these concepts, while still providing enough functionality for the game engine to be used across all areas of the course. One of the main goals of our game engine is for students to be able to easily learn it.

Another goal of our game engine is ease of use. This allows students to concentrate on working with game development concepts and their game ideas, rather than being frustrated with low level code or a complex engine. This is particularly important at the start of their game development course.

4.2 Methodology

We can test our game engine by evaluating how students use it. This game engine is designed to be used in a game development course, where they would be given assignments and tutorials. These assignments would test their understanding and ability to work with game development concepts. We therefore decided to use this format of assessment as the main form of evaluation of students, as it allows us to test the students' ability to complete game development tasks.

While keeping the tasks to be completed in the assignment the same, the game engine used in the evaluation can be changed. This allows us to evaluate to what degree the game engine affects how well the students complete the tasks of the assignment. We gave the students a set of game development tasks to complete in our engine, ModEngine, and another game engine, XNA, as comparison. Students had already used XNA for their coursework and were

familiar with its workflow.

The tasks were completed in one hour sessions, one session for ModEngine, and another session for XNA. We gave an hour talk before the sessions where we gave an overview of ModEngine. We used a screen capture program to record what was happening on the computer screens of the students during the experiment. We also used a microphone to record the audio from the testing room. Consent was obtained from the students to permit recording and data collection.

Evaluation metrics

In [5], the authors show that code complexity is linearly dependent on lines of code. Further, they say code complexity and lines of code "have a stable practically perfect linear relationship that holds across programmers, languages, code paradigms (procedural versus object-oriented), and software processes". We therefore use lines of code as a measure of code complexity in the student's assignments.

In the game industry in particular, and the software industry in general, productivity is an important metric. We want to evaluate how well students perform with our engine compared to the other engine in our evaluation. How fast a student completes the assignment is useful information as it also shows how easy or hard a system is to use.

If there is significant difference in completion time between a student's ModEngine assignment, and the XNA assignment, the only factor here that has changed is the engine, as the tasks are the same for each engine. This gives us another way of measuring the ease of use of the game engine. Therefore we use time of completion as a metric of performance.

Game Development Tasks

For the tasks students were given in the assignment, we gave them basic tasks that are the first tasks a programmer would need to handle before moving on to more complex challenges in making a game. Simple tasks that need to be handled at the beginning of development include, setting up a scene and adding objects to it and interacting with objects in the scene.

Therefore we chose tasks that tested the students' ability to solve simple game development problems. Specifically, the assignment tested a student's ability to load a model, add a model to a 3D scene, apply a texture, move an object around a scene, rotate an object and add objects as sub-objects to a parent object. The assignments themselves can be found in the Appendix.

The tasks given to the students were as follows:

1. Load in a 3D model of a robot that was provided with the engine, and place it into the 3D scene.
2. Apply a metal texture to the robot using a texture provided with the engine.
3. Add code to support moving the robot around the scene using the keyboard.
4. Add a left and a right arm to the robot using a 3D model of a robot arm that was provided with the engine
5. Once the arms have been added, add code to make the arms swing when the robot moves around.

Documentation

For ModEngine, we wrote a high level document where we gave an overview of ModEngine. It included simple code samples showing how to do basic setup of objects. It covered each of the components of the engine and how they can be used. We also produced html-based documentation which explained in more detail each class and method. Both these resources were made available to the students at the start of the evaluation session.

For XNA, Microsoft, who developed XNA, had already written extensive documentation on their engine through their MSDN website. A link to the XNA documentation was made available to the students in the XNA evaluation session. Students were also allowed to search on the internet for any information that would help them with the task at hand.

Once a student had completed all tasks, or time ran out, the student would be given a questionnaire to complete. The questionnaire covered which tasks the students found difficult and asked them to elaborate on what they found difficult about those tasks. It also provided a space for more general comments to be shared.

In experimental design, this form of testing is referred to as a repeated measures design or a within subjects design [24]. A repeated measures experiment involves testing each participant in the experiment against each of the conditions being tested for. In our experiment, we want to determine the effect the game engine has on the lines of code a student produces when completing the assignment, so our two conditions are ModEngine, and XNA. We also want to determine if and how much the chosen game engine has an effect on completion time. For this metric, we also use ModEngine and XNA as our two conditions.

We use a one-way analysis of variance (often shortened to ANOVA) with repeated measures [14] to determine if the game engine being used has a statistically significant effect on lines of code. We use the same type of analysis when looking at completion time. Analysis of variance is used to analyse the differences between the means of two or more sets of data. If the means are significantly different, we can determine the probability that the data being tested could have been produced by chance.

ANOVA operates on the same population which allows us to discover the affect different conditions have on the population. In our case, we are looking at whether the choice of game engine has a statistically significant effect on lines of code and completion time.

In keeping with common practice [36], 5% is commonly used as a threshold for statistical significance. That is, a finding is statistically significant if there is only a 5% or lower chance that the data was produced by chance. If the probability is below this threshold, it provides strong evidence that the data shows that choice of game engine influences code complexity.

Chapter 5

Results

In this chapter, we present the results from our evaluation. We report how many lines of code each of the student's assignments contained, and how long they took to do each assignment. We then perform a statistical analysis on both groups of data to see if there is any statistical significance to our results.

Participant Recruitment

To recruit participants for this evaluation, we needed students that had game development experience, had previously used XNA and knew how to program in C++. Students first learned C++ in 3rd year, and by this year had started studying game development. We thus advertised amongst 3rd year, and post graduate computer science students that were currently in the game development course, or had previously completed it. None of the students who participated had used ModEngine before this evaluation. We recruited 10 students for our evaluation. An advantage of ANOVA is that it provides results with sufficient statistical power even with small sample sizes.

Data collection

We used lines of code as a measure of code complexity, and time of completion as a metric of performance. Students were given a maximum of one hour to complete each set of tasks. After the testing was complete, we applied our chosen metrics to our data. Below we present the results from each of our metrics.

5.1 Lines of code

Lines of code, or line count, is a measure of code complexity. Line count measures how difficult the tasks are, but since the tasks to complete for each engine are the same, any difference in line count is due to the engine being used to complete the tasks. This means

that if an engine shows a higher average line count, it indicates that the tasks were more difficult to complete with that engine. ANOVA allows us to measure the effect the different game engines have on the lines of code of the assignment.

Students	ModEngine	XNA
Student 1	155	263
Student 2	160	191
Student 3	177	262
Student 4	140	201
Student 5	152	278
Student 6	174	212
Student 7	162	191
Student 8	166	229
Student 9	161	244
Student 10	160	216

Table 5.1: *Raw data from the evaluation showing lines of code*

In figure 5.1, we present a graph of the lines of code using the data we gathered from our evaluation.

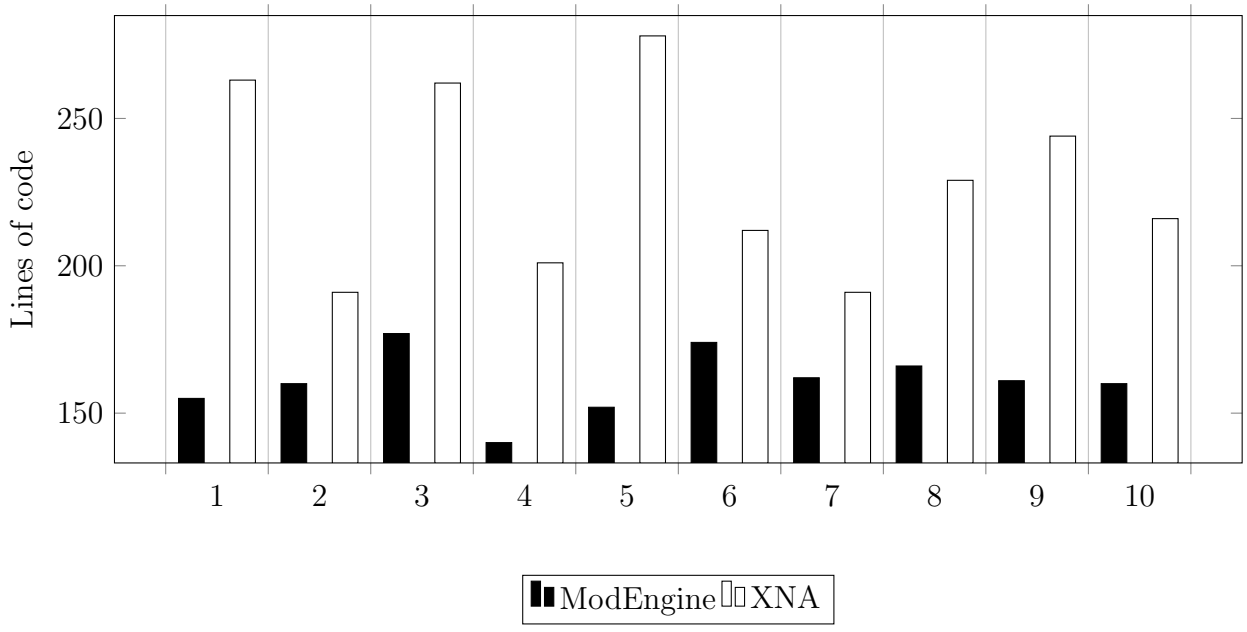

Figure 5.1: *A graph of lines of code. The graph shows that all the XNA assignments used more lines of code, and that there is a large margin between them across all assignments.*

	Mean	Std. Deviation	N
ModEngine	160.70	10.59402	10
XNA	228.70	31.62295	10

Table 5.2: *Descriptive Statistics*

ModEngine has a mean for lines of code of 160.7, and XNA has a mean for lines of code of 228.70. Students used, on average, 42% more lines of code when using XNA than using ModEngine.

Source	Sum of Squares	Degrees of Freedom	Mean Square	F	Partial Eta Squared
Lines of code	23120.00	1	23120.00	43.871	0.830
Error (Lines of code)	4743.00	9	527.00		

Table 5.3: *ANOVA with repeated measures[14] - Tests of Within-Subjects Effects.*

The sums of squares is a measure of dispersion. When this measure is scaled by the degrees of freedom, it is an estimate of variance from the mean. The mean square is the sums of squares divided by the degrees of freedom. The F-test, where we get the listed F value from, is used to test the variance of two groups, which allows us to determine if there is a statistically significant difference between the two groups. Partial Eta Squared is a measure of the size of the effect being measured [14].

The one-way ANOVA with repeated measures shows that there is a statistically significant difference between the lines of code of the ModEngine and XNA assignments ($F(1, 9) = 43.871$, $p < 0.001$). The F test with 1 degrees of freedom in the numerator and 9 degrees of freedom in the denominator applied to our data produces a value of 43.871.

We can calculate the probability that the data was not produced by chance alone (called the statistical significance) from our F value of 43.871, and this value is referred to as p. Our p value in this case is 9.662776×10^{-5}. This shows that it is highly unlikely the data is produced by chance, and that our data is highly statistically significant. This means that the choice of game engine has a statistically significant effect on lines of code, and thus on code complexity.

5.2 Completion time

Completion time indicates how difficult the tasks were to complete in general, but also how long it took a student to work out the solution to the problems presented. A shorter completion time indicates that students were able to more quickly solve the problems presented by the given tasks. Since the tasks to complete for each engine were the same, any difference in completion time is due to the different engines being used.

Students	ModEngine	XNA
Student 1	00:35:44	01:00:00
Student 2	00:58:30	01:00:00
Student 3	00:32:39	00:34:45
Student 4	01:00:00	01:00:00
Student 5	00:41:57	00:46:33
Student 6	00:47:38	01:00:00
Student 7	00:31:43	00:39:52
Student 8	00:33:17	01:00:00
Student 9	00:33:46	01:00:00
Student 10	00:48:41	00:46:42

Table 5.4: *Raw data from the evaluation showing completion time*

Below we present a graph of the completion time (in minutes), the line above is XNA, and the line below is ModEngine. If a participant ran out of time, the completion time was recorded as 60 minutes.

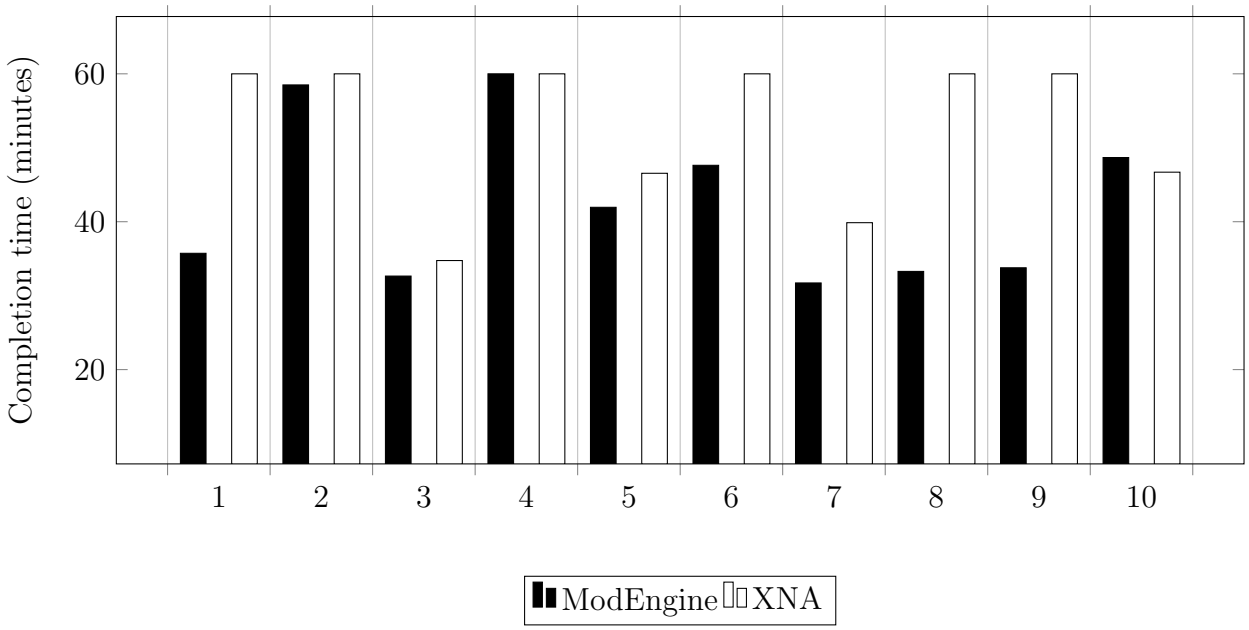

Figure 5.2: *A graph of completion time. The graph shows that in all but one case, students needed more time for the XNA assignment than the ModEngine assignment.*

	Mean	Std. Deviation	N
ModEngine	42.39	10.79	10
XNA	52.79	9.89	10

Table 5.5: *Descriptive Statistics*

Students, on average, took 24% longer to complete the XNA assignment than the ModEngine assignment, which equates to about 10 minutes more. This shows that students were more easily able to complete the tasks using ModEngine.

Source	Sum of Squares	Degrees of Freedom	Mean Square	F	Partial Eta Squared
Lines of code	540.384	1	540.384	8.375	0.482
Error (Lines of code)	580.708	9	64.523		

Table 5.6: *ANOVA with repeated measures - Tests of Within-Subjects Effects*

As we did with lines of code, we again use a one-way ANOVA with repeated measures to analyse our completion time data. It shows that there is a statistically significant difference between the completion times of the ModEngine and XNA assignments ($F(1, 9) = 8.375$, p

= 0.018). We have the same degrees of freedom in both the numerator and the denominator as our previous F test. When the F test is applied to our completion time data, it produces a value of 8.375.

The probability our completion time data was produced by chance alone, based on our F value of 8.375, is 0.018. This shows that it is unlikely the data is produced by chance, and that our data is statistically significant. Our analysis of completion time shows that choice of game engine has a statistically significant effect on completion time.

5.3 Overall task completion

In a game development course, students would be given game development tasks to complete. We tested students' ability to complete game development tasks given to them using ModEngine and XNA. We tracked how many students completed all the tasks for each section. Only one student did not complete all the ModEngine tasks, but we believe this student was an outlier, as this student was not used to the operating system or the setup of the computer being used for testing. This student had equal problems with the operating system and computer setup with both sets of tasks. All other students were experienced with the Windows operating system and the computer setup, and did not voice any difficulties with the computer setup.

9 out of **10** students finished the ModEngine tasks
5 out of **10** students finished the XNA tasks

Our results show that a far larger percentage of the students were able to complete all the tasks using ModEngine than using XNA. Since the students completed the same set of tasks for each engine, this completion ratio is due to the engine being used. This shows in general that our engine is easier to use.

5.4 Survey results

In the survey, the students were asked which step they found the most difficult. They could choose from among the tasks presented to them in the evaluation. The tasks given to them in the evaluation are numbered, and explained below.

List of steps

1. Load in a 3D model of a robot that was provided with the engine, and place it into the 3D scene.
2. Apply a metal texture to the robot using a texture provided with the engine.
3. Add code to support moving the robot around the scene using the keyboard.
4. Add a left and a right arm to the robot using a 3D model of a robot arm that was provided with the engine
5. Once the arms have been added, add code to make the arms swing when the robot moves around.

Students were then asked to explain what aspect of that task they found made it difficult. They could choose one or multiple options from among those detailed below, or they could choose "Other" and were provided space to give their own comment on what they found difficult about the task.

Reason for difficulties

1. Lack of familiarity with the subject area
2. The engine was hard to work with
3. Complex syntax needed
4. A lot of code required
5. Conceptual difficulties
6. Other

The labels on the graphs have been shortened to make the graph easier to read. First the results from the ModEngine survey are evaluated, followed by XNA.

5.4.1 ModEngine

Most difficult step

The following histogram shows how many students while doing the ModEngine tasks found the respective step the most difficult.

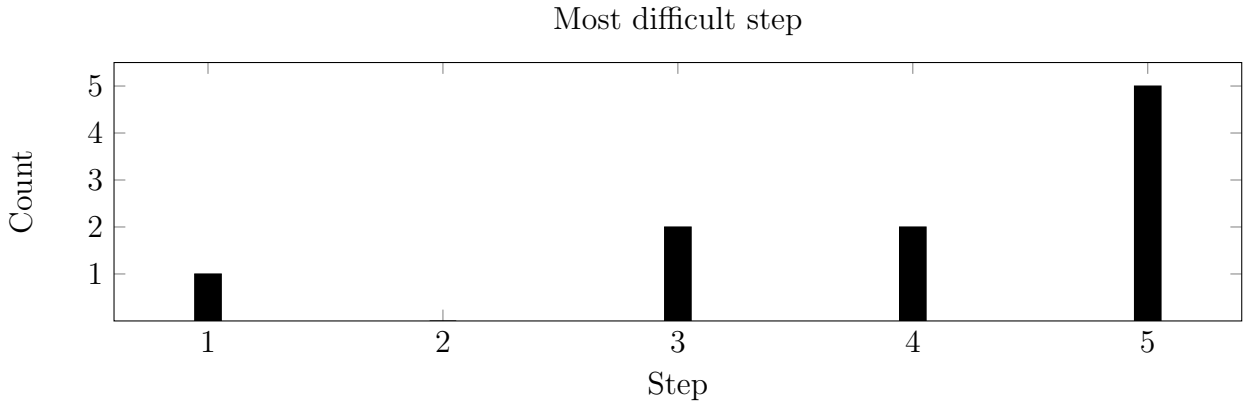

Figure 5.3: *Histogram showing count of students by reported difficulty step*

The step that most of the students found most difficult during the ModEngine evaluation was step 5. This step required students to write code to make the arms swing while the robot was moving around the scene. This shows students found it difficult to work out the formulae required to get the arms to swing correctly, which was the core of this task.

Reasons for difficulties

The following histogram shows how many students chose the respective reason as the explanation for why they found that task the most difficult.

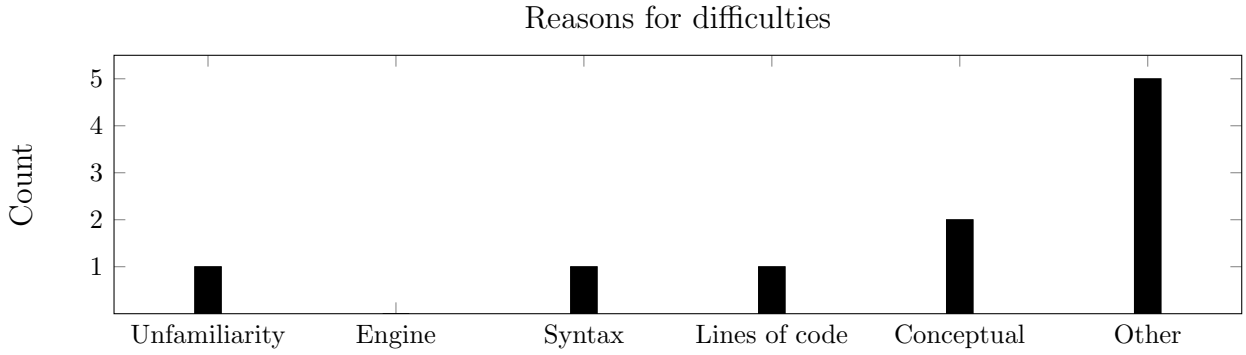

Figure 5.4: *Histogram showing count of students by reported reason for difficulty*

Most students found that none of the available options fully explained why they found that task the most difficult and explained their reasons on the survey. These students found that it was the mathematical formulae required by that task that made it difficult, as given by their explanation at the "Other" option on the questionnaire. This was a good result, as it

showed the students were dealing with the task at hand. This task required them to work out the formulae they needed to get the arms to swing. So rather than having to fiddle with low level code which was not directly related to the task they need to complete, they were able to concentrate on developing code for the task at hand.

5.4.2 XNA

Most difficult step

The following histogram shows how many students while doing the XNA tasks found the respective step the most difficult.

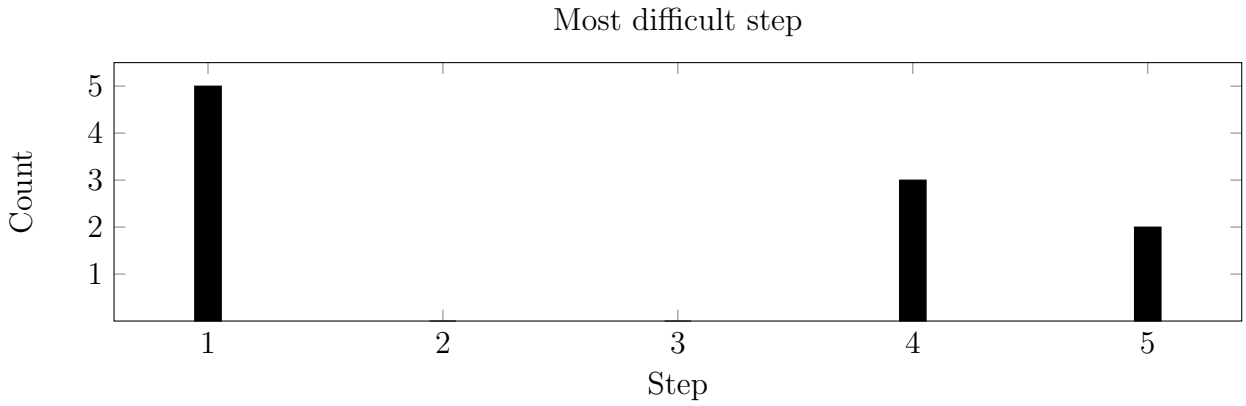

Figure 5.5: *Histogram showing count of students by reported difficulty step*

The step that most students found most difficult was step 1. This step required students to setup the scene and add the robot model to it. This shows the students found it difficult in XNA to perform the initial setup of a scene, and found adding objects into a scene more difficult.

Reasons for difficulties

The following histogram shows how many students chose the respective reason as the explanation for why they found that task the most difficult.

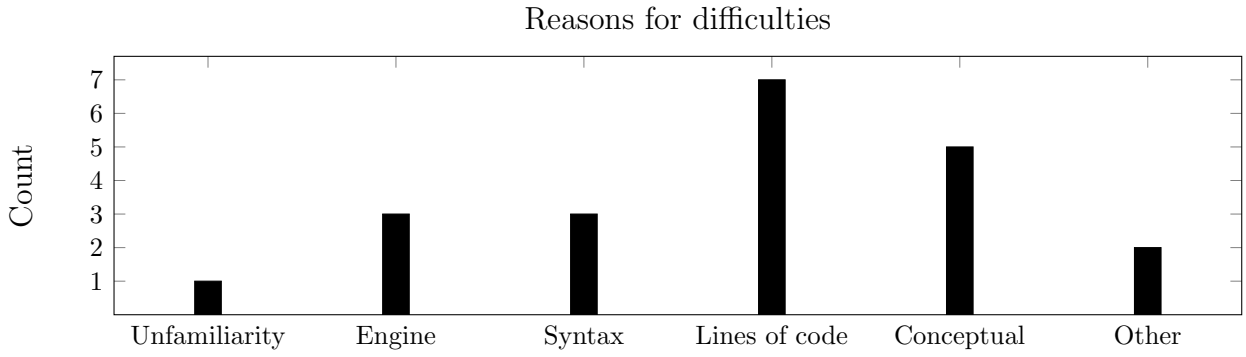

Figure 5.6: *Histogram showing count of students by reported reason for difficulty*

The majority of students found that XNA required a lot of lines of code to perform tasks with the engine. If a task requires more lines of code, it takes more time to complete, and increases the complexity of the task to complete. It also makes adding in new functionality more difficult, as well as it taking more time to add code into a larger code base.

5.5 ModEngine overview

Ease of use and understanding

On the last page of the ModEngine survey, students were asked questions about their experience using ModEngine. They were asked to remark on the ease of use of ModEngine's syntax, and if it the syntax was easy to understand. All students found that ModEngine's syntax was both easy to get a grasp of, and easy to use. Five students used the word "intuitive" to describe the syntax of ModEngine. To quote one student, "[the] system was intuitive and consistent". To quote another, "functions were intuitively named". One student said it "was not difficult to comprehend".

Other students remarked on the standards and conventions employed by ModEngine. One student remarked that it "hold to standards and is self explanatory". Another said that "clean and standardised coding conventions are used". Two students remarked on the naming of methods saying, respectively, "the methods were clearly named and described" and that "methods are clearly identifiable".

ModEngine components

Students were then asked, provided they make games in their own time, which components they would use of ModEngine their own game. This was to evaluate whether students, if they would use any components of ModEngine, found those components useful enough to want to make use of them. Only one student did not make games, so marked that section N/A. Another said he had already made his own engine, and would use it for his own games.

All of the remaining students however, said they would use at least one of the components of ModEngine. Seven of the remaining eight students said they would use the graphics components of the engine. Two of the eight students said they would use the sound component, and five of the eight students said they would use the physics component. One student said he would use the pathfinding component.

Evaluation of components

After selecting components that they would use, students were asked what about those components would make them want to use them. For those who chose the graphics component, the comments generally spoke of the ease of use of the system and that the scene graph was easy to use. To quote one student, "[The] integrated scene graph is pretty simple to use". Another liked "the ease of loading models and textures". One of the students described the graphics component as "intuitive, quick and easy" and found that it was "easy to set up [the] basics". Two students both said that what they liked of the graphics component was its "simplicity".

For those who chose the physics component, the comments overall spoke of the fact that they liked that the physics system was built-in and available as part of the engine. One student stated he "usually has a lot of issues integrating third party libraries". Another stated that the "physics are somewhat abstracted from the user" and that "this is simpler than a game framework like XNA". The student who chose the pathfinding component said he would use it "because it is very easy to use in general".

Positive and negative aspects

Students were asked to mention which aspects of ModEngine did they like and which aspects they did not like. In general, students commented on how easy the engine was to use, and that it was very intuitive. They also found model and texture loading convenient. One student stated that he liked the "intuitive setup, helper libraries and integrated functionality". He also mentioned that the "API provides useful examples".

Another student state that he liked that the engine was “intuitive, fast” and had “brevity of code”. For the texture and model collection, a student said that it “made setting up objects and resources easy”. Another student liked the “fast set up time”. For the engine as a whole, a student stated that it had “clean syntax”, and was “easy to learn and achieve tasks”. Another stated that he “liked the simplicity of everything”.

With regards to elements the students didn’t like, the most common comment was the fact that the engine was written in C++. One student stated that an “engine based in C++ makes it harder to use than an engine in a simpler language”. Three students brought up C++ as a negative. One student wanted a skybox to be included with the engine for use in outdoor scenes. Another wanted more keyboard shortcuts to be used in the game loop.

General comments

At the end of the survey, students were given a space to give any last comments they wished to make about ModEngine. The comments here were very varied as is to be expected with a question of this nature. One student stated that it was “nice to see so many features bundled into a single engine” and that he liked the “easy syntax”. Another liked the “convenient model and texture loading”. In reference to the engine as a whole, one student stated he “thought it was exceptionally clean and easy to use”. Another student saw the engine “being very useful for prototyping”.

Chapter 6

Conclusion

The Game Industry has grown dramatically over the last few years and now contains many billion dollar franchises. As the complexity of game production has increased, programmers and game designers have looked for ways to effectively manage this complexity. Game Engines were created to help in this regard, as they facilitate game production. As the game industry grows, so too does the demand for programmers. Many educational institutions now offer a game development course.

Since game engines are a commonly used tool in the game industry, it would be useful to give students experience using a game engine during the game development course. Using the game engine across the whole course would also mimic how the game industry uses a game engine for the duration of a game's production. It would also mean students can use their experience and knowledge of the game engine for the entire course, rather than having to learn multiple systems.

Full-featured game engines, such as Unreal, provide enough functionality to be useful in a game development course. Unfortunately, they are too complex and take too long to learn as students should be able to start working with the engine at the start of the game development course. Simple game production programs such as Game Maker are easy to learn and easy to use, but they lack extensibility and cannot be used for more advanced topics in the game development course. There is thus a need for a game engine that bridges the gap between full-featured game engines and more simple game production programs.

This dissertation explored the design and creation of such an intermediate level game engine. We analysed the requirements for such an engine and found that all engines currently available are lacking to some extent. We designed our game engine based on the requirements gathered during our taxonomy analysis.

We designed our engine to be used in a game programming course. We evaluated our engine using game development students to determine if it is easy to use and how well students could

complete game programming tasks using our game engine. With the aid of a survey, we asked students to evaluate the syntax and ease of use of our game engine. We set students a series of tasks to complete, and had them complete those same tasks with another comparator game engine, XNA.

To measure the code complexity of the student's assignment, we used lines of code as a metric. In [5], they show that lines of code is a good measure of code complexity. We used an analysis of variance (shortened to ANOVA) with repeated measures to analyse our lines of code data. Our p value (the chance that the data was due to chance alone) for lines of code is 9.662776×10^{-5}. This shows that there is a statistically significant difference between the lines of code of the ModEngine and XNA assignments. Students used, on average, 42 % more lines of code using the comparator game engine.

We used the number of tasks that were completed and completion time as metrics for performance. Here we again used ANOVA to do our statistical analysis. Our p value for completion time was 0.018. This shows that the game engine has a statistically significant effect on completion time as well. Students took, on average, 24% longer to complete the assignments with the comparator game engine.

9 out 10 students finished the ModEngine assignment, whereas only 5 out of 10 finished the XNA assignment. To summarise, students used, on average, fewer lines of code using our game engine, needed less time to complete the tasks, and more students completed all tasks with our game engine than using the comparator game engine XNA. This shows our game engine was easier to learn, and also easier to use.

This demonstrates that students can more easily explore game development concepts with ModEngine and can get started working with it much more easily. This also makes ModEngine ideal for use at the beginning of the game development course as students do not need to spend many months learning a complex game engine. The modular nature of ModEngine also means that it can continue to evolve as the game industry changes.

This engine has been designed from the start to be easy to learn and easy to use. As such, wherever code could be changed to be more performant, such as making optimizations, these changes were made as long they did not make the game engine harder to learn or harder to use. It is thus not as performant as full-featured game engines, but we feel this is an acceptable trade off given our goals.

Being a simplified game engine, ModEngine would not be suitable for professional level game

development, as it is not fully optimised nor does it contain the functionality a game engine on that level requires. This is, however, a suitable trade-off as we have designed a simplified game engine that is specifically suited for this environment.

In the introduction, we presented our research question: *"Can a simplified and modular game engine developed for a game development course help students in the exploration of game development concepts?"* Our lines of code, completion time and task completion metrics, backed up as well by our surveys, all point to the same conclusion: ModEngine is easier to use than the comparison and is easier to learn. This means ModEngine allows students to more easily explore game development concepts.

6.1 Future work

More work could be done on optimization to see where, or if, more performance can achieved from ModEngine without sacrificing any of the benefits the current system offers. More optimized code can be difficult to understand and use which may hinder student understanding. Additional functionality can also be added to ModEngine for other advanced components of game development, such as 3D animation. These systems tend to be quite complex, and that is why they are not a part of our current design.

A user development environment for ModEngine, where users can see their game world and edit it in real time without having to recompile and launch their game each time could be a useful addition to ModEngine. This is a common feature on a lot of modern full-featured game engines. Stereoscopic 3D, now being offered by devices such as Oculus Rift, could also be integrated into ModEngine using the modular framework. Scripting features for Artificial Intelligence would be another addition that could be added to the engine in the future. Scripts are useful as they do not require the program to be recompiled whenever script is changed.

Game Engines are very diverse, so determining what specific features to offer can be difficult. As the industry advances, and certain functionality becomes standard, users of our engine can add more functionality to the engine through its flexible module architecture thus keeping ModEngine useful and up to date.

Chapter 7

Appendix

In this chapter we present the supporting documents used during the testing.

The documents are:

1. ModEngine Programming Manual
2. ModEngine Survey
3. XNA Survey

ModEngine

Programming Manual

Getting started

Making a game with ModEngine is easy. Create a new class, and publicly inherit from the GameEngine class. In the constructor of your class, options that are defined in the GameEngine class are made available to you. Just set the option in your constructor and it will override the default value.

Here is a table showing all the options available in the constructor

Name	Description	Member name	Default value
Backface Culling	Turns backface culling on or off	m_backfaceCulling	false
Fullscreen	Makes the game window fullscreen	m_fullscreen	false
Lighting	Turns lighting on or off	m_lighting	true
Show console	Enables the console	m_showConsole	true
Show cursor	Displays the cursor	m_showCursor	true
Show frames per second	Show the frames per second counter on the game window	m_showFps	true
Window width	Sets the width of the game window	m_windowWidth	1280
Window height	Sets the height of the game window	m_windowHeight	800
Window x position	Sets the x position of the game window	m_windowX	0
Window y position	Sets the y position of the game window	m_windowY	0
Field of view	Sets the field of view of the renderer	m_windowFov	45
Near plane	Sets the near plane of the renderer	m_windowNearPlane	1
Far plane	Sets the far plane of the renderer	m_windowFarPlane	100
Target frame rate	Sets the frame rate that the renderer should try to maintain	m_targetFrameRate	60
Render id	Sets which renderer to use. Use 0 or OPENGL for OpenGL. Use 1 or DIRECTX for DirectX	m_renderId	0

Using the game engine components

ModEngine is a modular open source engine designed to be easy to use, but also very configurable and customisable. If you want to use a component, you simply add a method to your class that allows you to initialise and use that component.

The following is a table of the methods that load components for your game

Component Name	Method
Camera	loadCamera
Model collection	loadModelCollection
Texture collection	loadTextureCollection
Physics	loadPhysics
Scene graph	loadSceneGraph
Pathfinding	loadPathfinding
Sound	loadSound
GUI	loadGui

Game engine components

Each section below will detail each component and discuss how they can be used

Camera

```
void MyGame::loadCamera()
{
   m_currentCamera = new Camera(Vector3(0, 0, 0), Vector3(0, 0, 0),
      m_windowWidth, m_windowHeight, m_windowFov,
      m_windowNearPlane, m_windowFarPlane, 0.0f);
}
```

Here is an implementation of the loadCamera method in a new game called MyGame. All we need to do to initialise the camera is create a new camera object and assign it to m_currentCamera. Here are the camera's parameters:

Parameter	Description
Position	Position stored as a 3D vector
Rotation	X component used as pitch Y component used as yaw Z component used as roll
Window width	The width of the window
Window height	The height of the window
Window field of view	The field of view of the window
Near plane	The renderer's near plane
Far plane	The renderer's far plane
Movement speed	If the camera needs to move at a steady rate, it will use this value

In the example shown, we have used the built-in window width and window height to input to the camera.

Model collection

```
void MyGame::loadModelCollection()
{
   m_modelCollection = new ModelCollection();
   m_modelCollection->addModel("cube", "cube.obj", true);
}
```

The model collection simplifies loading and using 3D models. Create a new model collection, and set it to m_modelCollection. Add 3D models by using the addModel. The first parameter is the name the collection will use to access the model, the second parameter is the path to the 3D model, and the third parameter is whether you want the engine to generate normals for you, or use the ones in the file.

You can also call the model collection without the third option, as in

```
m_modelCollection->addModel("cube", "cube.obj");
```

Once you have loaded the model, you can call the setModel method on your scene graph node to make it use the model. More on scene graph nodes later.

```
node->setModel("cube");
```

Texture collection

The texture collection simplifies loading and using textures. Create a new texture collection, and set it to m_textureCollection.

```
void MyGame::loadTextureCollection()
{
   m_textureCollection = new TextureCollection();
   m_textureCollection->addTexture("texture", "texture.bmp");
}
```

Once you have loaded the texture, you can call the setTexture method on your scene graph node to make it use the texture. More on scene graph nodes later.

Physics

The physics systems controls the physical interaction between objects. Create a new physics plugin, and assign it to m_physicsPlugin, and then call initPhysics.

```
void MyGame::loadPhysics()
{
   m_physicsPlugin = new BulletPhysicsPlugin();
   m_physicsPlugin->initPhysics();
}
```

You can probably just use the default loadPhysics method, and not have to make your own one, unless you want to change the physics plugin. You have to apply a collision box to each object that you want to be a part of your physics simulation. You do this by calling the addCollisionBox method on the physics plugin.

```
m_physicsPlugin->addCollisionBox(node, 10.0f, ZeroVector3, ZeroVector3,
   0.0f, scale.x, scale.y, scale.z);
```

The parameters are as follows

Parameter	Description
Scene graph node	Pointer to the tree node to apply the collision box to
Friction	Sets the friction of the collision box's surface
Rotation	X component used as pitch Y component used as yaw Z component used as roll
Speed	The speed of the node
Mass	The mass of the node
Width	The width of the collision box
Height	The height of the collision box
Depth	The depth of the collision box

If you set the mass to 0, the object is added to the physics simulation, but it will become static, so it is not affected by gravity of the force of other objects hitting it.

Scene graph

The scene graph manages the objects that make up the 3D scene. Each object is represented by a node in the scene graph. Create a new node and set it to m_scene. This will be our root node. All nodes that are added to the scene must be added to the root node. You can pass in the texture collection, model collection and the physics plugin if you are using them.

```
m_scene = new TreeNode("root", m_modelCollection,
   m_textureCollection, m_physicsPlugin);
```

Once you have your root node, nodes that represent objects in the scene can be added to it.

```
m_floor = new TreeNode("floor", Vector3(-scale / 5.0f, -5, -scale / 5.0f),
   ZeroVector3, Vector3(scale, 1, scale));
m_scene->addChild(m_floor);
m_floor->setModel("cube");
m_floor->setTexture("floor");

Vector3 floorScale = m_floor->getScale();
m_physicsPlugin->addCollisionBox(m_floor, m_friction, ZeroVector3,
   ZeroVector3, 0.0f, floorScale.x, floorScale.y, floorScale.z);
```

Leaf nodes have different parameters than the root node. Any information leaf nodes need

are propagated from the root node once the node has been added to the scene graph. The first parameter in the tree node constructor to create leaf nodes, is the name of the node. The second parameter is the rotation of the node. The third parameter is the scale of the node.

Add the node to the scene graph by calling the addChild method on the root node. As shown in the model and texture collection sections, call setModel or setTexture to set the model or texture respectively. Call the addCollisionBox method if you want to add the node to the physics simulation.

Pathfinding

The pathfinding plugin handles any 2D pathfinding that needs to be done by your game. Create a new pathfinding plugin and set it to m_pathfindingPlugin. Initialise it by calling initGraph, and pass it a 2D map of the area, then the width and height.

```
void MyGame::loadPathfinding()
{
   m_pathfindingPlugin = new MicropatherPlugin();
   m_pathfindingPlugin->initGraph(m_map2d, m_mapWidth, m_mapHeight);
}
```

Using the pathfinder is a simple matter of setting the start position, the destination, and then call findPath. Use the getSolvedPath method to get a vector of 2D positions, showing the path from the start to the destination.

```
m_pathfindingPlugin->setStartPosition(m_startPosition);
m_pathfindingPlugin->setDestination(m_endPosition);
m_pathfindingPlugin->findPath();
vector<Vector2> path = m_pathfindingPlugin->getSolvedPath();
```

Sound

The sound system handles the loading and playing of sound files. Create a sound plugin and set it to m_soundPlugin. The first parameter is a handle to the renderer window, which you can get by calling getWndHandle on the renderer object. The second parameter is the size of the buffer, which is how many sound files you want to load and have available during your game.

```
void MyGame::loadSound()
{
   m_soundPlugin = new DirectSoundPlugin(m_renderer->getWndHandle(), 1);
}
```

Once we have created our object, we need to initialise it using the initialiseSound method. Load a file using the loadFile method, whose first parameter is what buffer index to load the file into, and the second parameter is what path to load the file from.

```
m_soundPlugin->initialiseSound();
m_soundPlugin->loadFile(0, "photon.wav");
```

The game can play the sound using the play method on the sound plugin.

```
m_soundPlugin->play(0, 0);
```

The first parameter is which the buffer index to play from, and the second parameter is where in that buffer to start playing.

GUI

The GUI layer handles buttons and labels that a game needs. Create a new GUI layer and set it to m_guiLayer.

```
m_guiLayer = new GuiLayer();
```

The GUI system allows buttons or labels to be added to the GUI layer. Here is how you create a label

```
Label* label = new Label("label", "New label", 0, 0, m_renderer);
```

The first parameter in the constructor is the name of the label. The second parameter is the text that forms the label. The third and fourth parameters are the x and y positions of the label. The last parameter is pointer to the renderer, so use m_renderer there. Add the label to the GUI layer by calling the addControl method on the GUI layer object.

```
m_guiLayer->addControl(m_rendererLabel);
```

Here is how you create a button

```
Button* button = new Button("button", "Button text",
   0, 0, 300, 50, m_renderer);
```

The first parameter is the name of the button. The second parameter is the text that displays on the button. The third and fourth parameters are the x and y positions. The fifth and sixth parameters are the width and height of the button. The last parameter is a pointer to the renderer.

This is an overview of the game engine, and how to add and use components. For more detailed information about the classes, and the methods and members of those classes, refer to the full documentation.

Networking

Handling network packets and player connects and disconnects is easily handled in the game engine. Use the openClientConnection method on a client machine to connect to a server.

```
openClientConnection("192.168.0.1", "43560");
```

For a server, use the openServerConnection method to start the server.

```
openServerConnection("192.168.0.1", "43560");
```

These methods should be run only when the connection needs to be made. For instance, if a button is added for users to click when they want to connect, run the openClientConnection method when the button is clicked. The game engine handles the low level details of player's connecting. When a player connects, there may be other tasks that need to be performed, such as displaying a message that a player has joined or add a message to the server log.

Use the clientHandlePlayerConnect or serverHandlePlayerConnect methods to perform these tasks. The clientHandlePlayerConnect is run on the client's side and serverHandlePlayer-Connect is run on the server's side.

```
void MyGame::clientHandlePlayerConnect(string playerName)
{
   cout << playerName << " has joined" << endl;
}
```

Once a player has disconnected, there are the clientHandlePlayerDisconnect and serverHandlePlayerDisconnect methods to handle any tasks that need to be performed at that time.

The most important part of the networking system is the clientHandleNetworkPacket and serverHandleNetworkPacket methods that handle the actual packets received over the net-

work. As before, the clientHandleNetworkPacket is run on the client's side, and serverHandleNetworkPacket is handled on the server's side.

```
void GameEngine::clientHandleNetworkPacket(NetworkPacket packet)
{
   cout << "Client received a network packet" << endl;
}
```

Handling mouse and keyboard input

Mouse input is an important part of game input. Add the onMouse Move method to handle mouse movement.

```
void MyGame::onMouseMove()
{
   Vector2 mousePos = getMousePos();
   Vector2 prevMousePos = getPrevMousePos();
}
```

Use the getMousePos method to get the current mouse position. If you want the previous mouse position, for use in, for example, a first person game, use the getPrevMousePos. The samples included with the game engine show different ways of using the mouseMove method. ModelDemo uses onMouseMove to implement mouse look, so the user can move around the model and look at it from different angles.

Handling mouse clicks is done by adding the onLeftMouseButtonClick and onRightMouseButtonClick methods. If you use this method without parameters, it is run whenever the user clicks the appropriate mouse button. Here is what the onLeftMouseButtonClick and onRightMouseButtonClick looks like in your game implementation.

```
void MyGame::onLeftMouseButtonClick()
{
   cout << "Left button click" << endl;
}
```

```
void MyGame::onRightMouseButtonClick()
{
   cout << "Right button click" << endl;
}
```

There is another use of onLeftMouseButtonClick and onRightMouseButtonClick. If you have added any GUI buttons to your game, you want to know when users click on them. You do this by using the mouse click methods, but by adding button as parameters. The game engine will find the button that was clicked on, as pass it in as a parameter. Here is a sample implementation of onLeftMouseButtonClick

```
void MyGame::onLeftMouseButtonClick(Button* button)
{
   cout << "left clicked on " << button->m_text << endl;
}
```

You may want to handle mouse wheel movement. Add the mouseWheelRotation method with the increments as a parameter.

```
void MyGame::mouseWheelRotation(int delta)
{
   cout << "Mouse has been rotated " << delta << " increments" << endl;
}
```

There is one last built in mouse method. If you want to know when the user mouses over a button, you can use the onButtonMoveOver method.

```
void MyGame::onButtonMoveOver(Button* button)
{
   cout << "move over button" << endl;
}
```

Keys that you want to be processed every frame should be added to your game loop implementation. However, if you only want a key to be processed once, then you can use the handleKeypress method.

```
void MyGame::handleKeypress(int key)
{
   if (key == 'A')
```

```
   {
      cout << "Hello world" << endl;
   }
}
```

Game loop

It all comes together in the game loop. Once all the components have been setup, the game loop is run every frame until the program exits. All the gameplay code goes into the game loop. Here is where code must be added to move the player's character, update enemy positions, and run any other needed for the game. Add the gameLoop method to your class to implement your game loop. Here is a sample implementation of the gameLoop method, where the up and down keys moves the camera forward or backwards, and the left and right keys moves the camera to the left or right.

```
void MyGame::gameLoop()
{
     float radians = MathUtil::getRadians(m_currentCamera->getRotateY());
     float radians90 = MathUtil::getRadians(m_currentCamera->getRotateY() + 90);

     //up
     if (m_keys[UP_KEY] || m_keys[DOWN_KEY])
     {
          float sign = (m_keys[UP_KEY] ? 1.0f : -1.0f);

          m_currentCamera->translateX(cos(radians) *
                m_currentCamera->getSpeed() * sign);
          m_currentCamera->translateZ(sin(radians) *
                m_currentCamera->getSpeed() * sign);
     }

     if (m_keys[LEFT_KEY] || m_keys[RIGHT_KEY])
     {
          float sign = (m_keys[LEFT_KEY] ? 1.0f : -1.0f);

          m_currentCamera->translateX(-cos(radians90) *
```

```
                m_currentCamera->getSpeed() * sign);
          m_currentCamera->translateZ(-sin(radians90) *
                m_currentCamera->getSpeed() * sign);
     }
}
```

Refer to the samples included with the GameEngine to see more complex examples of using the gameLoop method.

Questionnaire for the ModEngine Practical

Name: Email:

ModEngine

1. How many lines of C++ source code (excluding header files) did you require for your ModEngine assignment?

Answer:

2. Which step of the ModEngine assignment did you find was the most difficult to implement?

Answer:

3. Referring to question 2 above, what about that section made it difficult (you may select multiple answers)?

[ ] Lack of familiarity with the subject area
[ ] The engine was hard to work with
[ ] Complex syntax needed
[ ] A lot of code required for each task in that section
[ ] Conceptual difficulties
[ ] Other (explain):

Answer:

4. If you didn't finish the ModEngine practical, was it due to

A) insufficient time, B) difficulty using the engine, C) other (explain)

Answer:

5. If you selected B to question 4, what difficulties did you face using ModEngine?

[ ] Syntax was hard to work with
[ ] Required lots of code for each task in the assignment
[ ] Insufficient documentation
[ ] Other (explain):

Answer:

The ModEngine Game Engine

1. Was the syntax of ModEngine easy to understand? Explain

Answer:

2. Was the syntax of ModEngine easy to use? Explain

Answer:

3. If you develop games in your own time, would you use ModEngine for the following components? Enter an X to specify which components you would use, mark N/A if you don't develop games in your own time, mark none if you wouldn't use any of the components in ModEngine

[ ] Graphics
[ ] Sound
[ ] Physics
[ ] None
[ ] N/A

4. Referring to question 4, if you would use component(s) of ModEngine, what aspects of ModEngine would make you choose it over another engine?

Answer:

5. What aspects of ModEngine did you like? What aspects did you not like? Please provide

comments.

Answer:

6. Do you have any other comments to make about ModEngine or your experience using it?

Answer:

Questionnaire for the XNA Practical

Name: Email:

XNA

1. How many lines of source code did you require for your XNA assignment?

Answer:

2. Which step of the XNA assignment did you find was the most difficult to implement?

Answer:

3. Referring to question 2 above, what about that section made it difficult (you may select multiple answers)?

[ ] Lack of familiarity with the subject area
[ ] The engine was hard to work with
[ ] Complex syntax needed
[ ] A lot of code required for each task in that section
[ ] Conceptual difficulties
[ ] Other (explain):

Answer:

4. If you didn't finish the XNA practical, was it due to

A) insufficient time, B) difficulty using the engine, C) other (explain)

Answer:

5. If you selected B to question 4, what difficulties did you face using XNA?

[ ] Syntax was hard to work with
[ ] Required lots of code for each task in the assignment
[ ] Insufficient documentation
[ ] Other (explain):

Answer:

References

[1] Laurent A., *Understanding open source and free software licensing*, 2004.

[2] Nareyek A., *Ai in computer games*, ACM Queue (2004).

[3] Bungie, *I shot you first! gameplay networking in halo: Reach*, Game Developer Conference Moscone Center (2011).

[4] Evans C., *World creation in cryengine*, SIGGRAPH (2011).

[5] Jay G. Hale J. Smith R. Hale D. Kraft N. Ward C., *Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship*, Journal of Software Engineering and Applications (2009).

[6] Valve Corporation, *Steam hardware and software survey*, `http://store.steampowered.com/hwsurvey/?platform=pc`, 2013.

[7] Eberly D., *3d game engine design*, Morgan Kaufmann Publishers, 2001.

[8] Jacobson D., *Apis: A strategy guide*, O'Reilly, 2012.

[9] Devmag.org.za, *3d graphics in game maker part 1*, 2009.

[10] Coumans E., *Optimizing proximity queries for cpu, spu and gpu*, SIGGRAPH (2010).

[11] Trees F., *Motivating and engaging students with game maker, csit '09 presentation*, 2009.

[12] Idea finder, *Pong video game history*, `http://www.ideafinder.com/history/inventions/pong.htm`, 2007.

[13] Armitage G., *Networking and online games: Understanding and engineering multiplayer internet games*, 2006.

[14] Gamst G., *Analysis of variance designs: A conceptual and computational approach with spss and sas*, 2008.

[15] McTaggart G., *Half-life 2 / valve source shading*, Game Development Conference Moscone Center (2004).

[16] Gamerant, *Call of duty surpasses $3 billion in revenue*, http://gamerant.com/call-duty-surpasses-3-billion-seb-3595/, 2009.

[17] Epic Games, *Unreal engine 4 infiltrator demonstration*, SIGGRAPH (2013).

[18] Yoyo Games, *Working with 3d*, 2009.

[19] ———, *Compare gamemaker version — yoyo games*, http://www.yoyogames.com/make, 2012.

[20] Wikia Gaming, *Video game industry*, 2010.

[21] The NPD Group, *Press release - video game sales 2009*, http://www.npd.com/press/releases/press_100114.html, 2009.

[22] Millington I., *Artificial intelligence for games*, 2009.

[23] Blow J., *Game development: Harder than you think*, ACM Queue (2004).

[24] Creswell J., *Research design: Qualitative, quantitative, and mixed methods approaches*, Sage Publications, 2009.

[25] Gamma E. Helm R. Johnson R. Vlissides J., *Design patterns elements of reusable object-oriented software*, Addison-Wesley, 1995.

[26] Kurose J., *Computer networking: A top-down approach*, 2012.

[27] Joystiq, *Guitar hero franchise passes the $1b mark*, http://www.joystiq.com/2008/01/21/guitar-hero-franchise-passes-the-1b-mark/, 2008.

[28] MSR Kodu, *Kodu game lab*, http://marketplace.xbox.com/en-US/games/media/66acd000-77fe-1000-9115-d8025855024c/, 2009.

[29] Grinning Lizard, *Micropather*, http://www.grinninglizard.com/MicroPather/.

[30] Bevan M., *The scumm diary: Stories behind one of the greatest game engines ever made*, http://www.gamasutra.com/view/feature/196009/the_scumm_diary_stories_behind_.php, 2013.

[31] Haines E. Moeller T., *Real-time rendering*, A.K. Peters Ltd., 2008.

[32] Cooper S. Moskal B., Lurie D., *Evaluating the effectiveness of a new instructional approach*, SIGCSE (2004), 75–79.

[33] Next-gen, *The making of: Spacewar!*, http://www.next-gen.biz/features/the-making-of-spacewar, 2009.

[34] University of Wisconsin, *Understanding modern device drivers*, 2012.

[35] Pressman R., *Software engineering: A practitioner's approach*, McGraw-Hill Higher Education, 2001.

[36] Witte R., *Statistics*, Wiley, 2009.

[37] Rabin S., *Introduction to game development*, Charles River Media, 2010.

[38] Jenkin's Software, *Raknet - multiplayer game network engine*, http://www.jenkinssoftware.com/.

[39] Kavakli M. Szilas N., Barles J., *An implementation of real-time 3d interactive drama*, Interactive Entertainment **5** (2007), no. 1.

[40] Mooney T., *Unreal development kit game design cookbook*, 2012.

[41] Christopher Tin, *Christopher tin - biography*, http://christophertin.com/, 2012.

[42] Luton W., *Making better games through iteration*, http://www.gamasutra.com/view/feature/132554/making_better_games_through_.php?page=1, 2009, pp. 1–5.

[43] Demers A. White W. Koch C. Gehrke J., *Better scripts, better games*, ACM Queue (2008).