Honours Project Report

Analysing the Importance of Open Source Software and the Game Loop in the Design and Implementation of a Multithreaded Game Engine

Sean Packham
PCKSEA001

Supervised by
Ken MacGreggor

And
Edwin Blake

Abstract

This report discusses the design and implementation of a modular multithreaded game engine. Open source libraries were used to decrease the risk of game engine design and played an important role in the success of this project. Different multithreaded design techniques are discussed and are applied to various areas of game engine design. A well designed multithreaded game loop was essential to this project's success and the report explores how the final implemented game loop evolved. Various game loops are tested and the final implemented multithreaded game loop shows significant improvements over the single threaded equivalent.

Keywords: I.2.1 [Applications and Expert Systems]: Games; D.4.1 [Process Management]: Multiprocessing/multiprogramming/multitasking, Synchronization, Threads; D.2.2 [Design Tools and Techniques]: Object-oriented design methods;

Acknowledgments

Thanks to my supervisors for all the advice and to Edwin for reviewing my draft submission.

Table of Contents

1. Introduction

1.1 Multithreaded Game Engines

The games industry is the largest most successful entertainment industry. The thirst for games is continually increasing and has resulted in game developers having to make use of game engines to extract maximum performance and decrease development time.

A game engine is a highly specialised piece of software that separates the technology from the game. Traditionally game engines have been designed for single processor architectures. In more recent years improvements to single processor units have reached physical limits and hardware manufacturers have invested resources into multi-core and multi-processor architectures. More specialised software systems are required to utilise this new hardware if further performance increases are to be seen. This is where the importance of a game engine is realised. A well designed game engine can decrease the time and cost of development substantially but more importantly make efficient use of the available hardware.

Developers need to be aware of the various multithreading techniques that are available to utilise multi-core and multi-processor hardware. Threads allow a program to split a task into different paths of execution. This is useful even on single core hardware as it allows tasks to appear to happen simultaneously even though the threads are executed in a sequential order. The benefits are far more significant when threading is used on multi-core hardware. This is because the hardware can process multiple threads at the same time. The more cores that are available the more threads can be processed simultaneously.

1.2. Software Engineering Problem

The purpose of this software engineering project is to explore the complexities of a modular multithreaded 3D game engine design and implementation. The

engine will be designed and implemented using existing open source technologies where ever possible to decrease the risks of producing a game engine from scratch. The Scrum agile software engineering process [1] will be used to manage and control the risks of the project and to guide the development in an iterative incremental fashion.

The most obvious risk that existed for this project from the start was that the design and implementation of a multithreaded game engine was out of the scope of an honours project. An immediate affect of this would be that at the end of the project a game engine would not have been produced. Game engine development is a large time and resource consumer. It was therefore infeasible to develop the game from scratch and was necessary to make use of various open source libraries. The libraries used form the modular sections of the game engine.

The measurement of the success of this project will therefore be determined on whether a multithreaded game engine can be designed and implemented by interfacing open source libraries together. The usage of various threading techniques to extract efficient usage of multi-core hardware is another key success factor that will be measured. More specifically the efficient design and implementation of a multithreaded game loop was essential to make sure the engine showed a performance increase on multi-core hardware. Many generations of game loops were tested until a custom multithreaded game loop was settled on.

1.3. Implementation

In order to meet the success requirements early focus was put onto evolving the engine's game loop. From the early stages of the prototype a simple single threaded game loop was evolved to the custom multithreaded one the engine currently uses. The open source libraries that were used to decrease the risk of development were the Bullet Physics Library [2] and the Irrlicht Renderer [3].

1.4. Outline

In Chapter 2 the background research for this project is discussed and it is shown that game developers are moving towards multithreaded engine design. The design of the game engine as well as threading concepts and areas where threading can be introduced into a game engine are covered in Chapter 3. The implementation of the engine is in Chapter 4. The various implementations of the game loops tested are discussed along with the implementation of other engine components. In Chapter 5 the experimental methodology is explained and in Chapter 6 the results are presented. The relevance and importance of the results are discussed in Chapter 7. The report is concluded in Chapter 8 and the future works section follows in Chapter 9.

2. Background

2.1. Introduction

The idea of creating multithreaded game engines is still in its infancy. Only in the last few years have game consoles used multiple cores and processors. Most of the leading research into this area of computer science has been lead by the games industry. The rest of this chapter will now explore some of the related work and research that is available.

2.2. Importance of Design

The design stage of software engineering is an important step that identifies possible risks and requirements, comes up with solutions to the risks, and lays down the foundation architecture design from these. Introducing new features late into the software complicates the implementation and is sometimes not feasible. This is true for adding threading to game engines. Writing multithreaded software systems is already a complicated task so adding threading to already implemented software is not recommended.

In “Multithreaded game engine design" [4] the authors show that if a game engine was not designed for multi core hardware it is not guaranteed to see a performance increase when running on multi-core hardware. They give examples of cases where performance even decreases. With a lot of gaming hardware moving to multi core technology, game engines need to make efficient use of this hardware. The paper goes on to examine Amdahl's law and the potential speed up multithreaded game engines can achieve. Amdahl's law states that the speed up depends on the number of processors and the percentage of sequential code [4].

2.3 Threading Techniques

There have traditionally been two threading techniques used when designing multithreaded software. In “Multithreaded game engine design" [4] these two

methods of parallelism are examined. The first is task parallelism and involves splitting the problem into separate tasks that execute independently and asynchronously. The second is data parallelism and splits the processing of a large group of data that has no dependencies amongst other data in the group into multiple threads.

Valve Software [5] has been creating multithreaded game engines for the last few years. In various online articles [6] [7] Valve Software explain their experience with multithreaded engine design. They have also identified the same two threading techniques mentioned already as approaches to designing multithreaded game engines as well as the need for efficient data structures. They stress the importance of sharing and controlling access to sensitive shared data.

2.4. Potential Threading Areas

In “Multithreaded game engine design" [4] various tasks that can be parallelised using a task parallelism scheme are examined. These tasks are resource management, input, networking, scene management, physics and artificial intelligence. The authors mention that the problem is not finding areas of a game engine that can be parallelised but controlling and scheduling the execution of the concurrently executing operations.

2.5. Dedicated Multi-Core Gaming Hardware

Game console manufacturers have been concentrating on multi-core and multi-processor hardware more recently. The XBOX360 [8], Playstation 2 and Playstation 3 [9] all use multi-core or multi-processor hardware.

Microsoft has also been pushing multithreaded engine design on its XNA framework [10]. The framework is a set of tools and managed libraries based on the Microsoft .NET Framework 2.0 that allow students and hobbyists to produce games for Windows and the XBOX360 [8].

3. Design of the Game Engine

A game engine is a highly specialised software system that separates the supporting technology from the game. A game engine is built from many sub components that all work together to become a powerful development tool. Each sub component is a specialised engine on its own that performs specific tasks. The need for game engines arose due to the complexity of modern day games and the large amount of resources that are required to develop them.

The modern day design of game engine follows the object orientated paradigm closely to allow for better encapsulation of the technology used to build the engine. The idea is to construct a tool for developers to allow them to focus more on game design and content creation rather than developing technology. A game engine often comes with custom development environments and editors to further simplify the development of a game. The custom Unreal Editor, shown in Diagram 1, allows the Engine to be used in What You See Is What You Get (WYSIWYG) manner.

Diagram 1: The Unreal 3 Engine Editor allows easy development of games

3.1. High Level Engine Architecture

With multi core hardware becoming more prevalent than ever before, game developers are continuously searching for more efficient ways to utilise the available processing power. It has become increasingly important that the design of the engine incorporates threading to take advantage of multi-core hardware from the beginning of the project's life. Trying to add threading to any piece of completed software is no small undertaking but with game engines the task might just be impossible. Below is a high level design of the engine that was produced for this project. The areas that were threaded will be explained after the general engine design. The threading library used was the Posix Threading Library.

3.1.1. Physics Engine

The physics engine is a software system that simulates Newtonian physics models. The Newtonian physics model uses a body's mass and the applied forces to calculate the velocity to be applied. The physics engine is one of the most important aspects of a game engine and therefore the efficient design of this component is crucial. In most games the physics engine is the foundation of the game.

There are generally two flavours of physics engines, real-time and high precision. High precision physics engines are used for accurately simulating scientific calculations. These types of simulations are very computationally heavy and are therefore infeasible to simulate game physics. Real-time physics engines are more concerned with performing a simulation in real-time on everyday hardware. This requirement can only be achieved by sacrificing the accuracy of the simulation but since these types of physics engines are used for non-critical simulations this is not that much of an issue. Real-time physics engines are therefore best suited for game development.

A physics engine uses what is known as a discrete step simulation to step the world forward. All object interactions are modelled for small time steps or

ticks. The result of this is that objects only move a very small distance each tick. The larger the time frame for each tick the more inaccurate the simulation becomes and an objects motion becomes jumpy or erratic. Usually games perform between 25 and 60 physics engine ticks per second as this is the range in which the human eye is less able to detect frame changes.

With the increase in the capabilities of gaming hardware developers have the opportunity to create more believable, realistic and fun games. Games have become increasingly complex in the way they represent their digital worlds and have more than ever required the functionality of a physics engine. Games use real-time physics engines to model interactions within the gaming world more specifically the motion and collisions of objects. Objects can be split into two groups, static game objects and dynamic game objects. Static objects represent objects that don't move and therefore are perfect for modelling buildings or scenery. Dynamic objects are used to model objects that have mass and can have forces applied to them. Dynamic objects are used to represent players, weapons, cars, bullets etc. Basically anything that has to move is a dynamic object.

Physics engines use specialised data structures to represent the dynamics world. In both computer graphics and physics bounding volumes are important data structures for simplifying the complexity of calculations. A bounding volume is a closed volume that completely contains a group of objects. Generally the volume covers a set of objects with the smallest shape possible. Bounding volumes are predominantly used for object culling, this is the removal of an object or group of objects from a scene thereby decreasing the amount of objects that need to be processed when the scene is rendered. This has great benefits in other algorithms such as object collision tests.

The most common bounding volume used in a physics engine is an Axis Aligned Bounding Box (AABB). An Axis Aligned Bounding Box is a bounding volume that is aligned to the axis of the coordinate system. This means that the sides of the box fall directly on the Cartesian planes and are not rotated. This alignment makes computation simple but also means that if the object rotates and now no longer fits in the AABB it has to be recomputed.

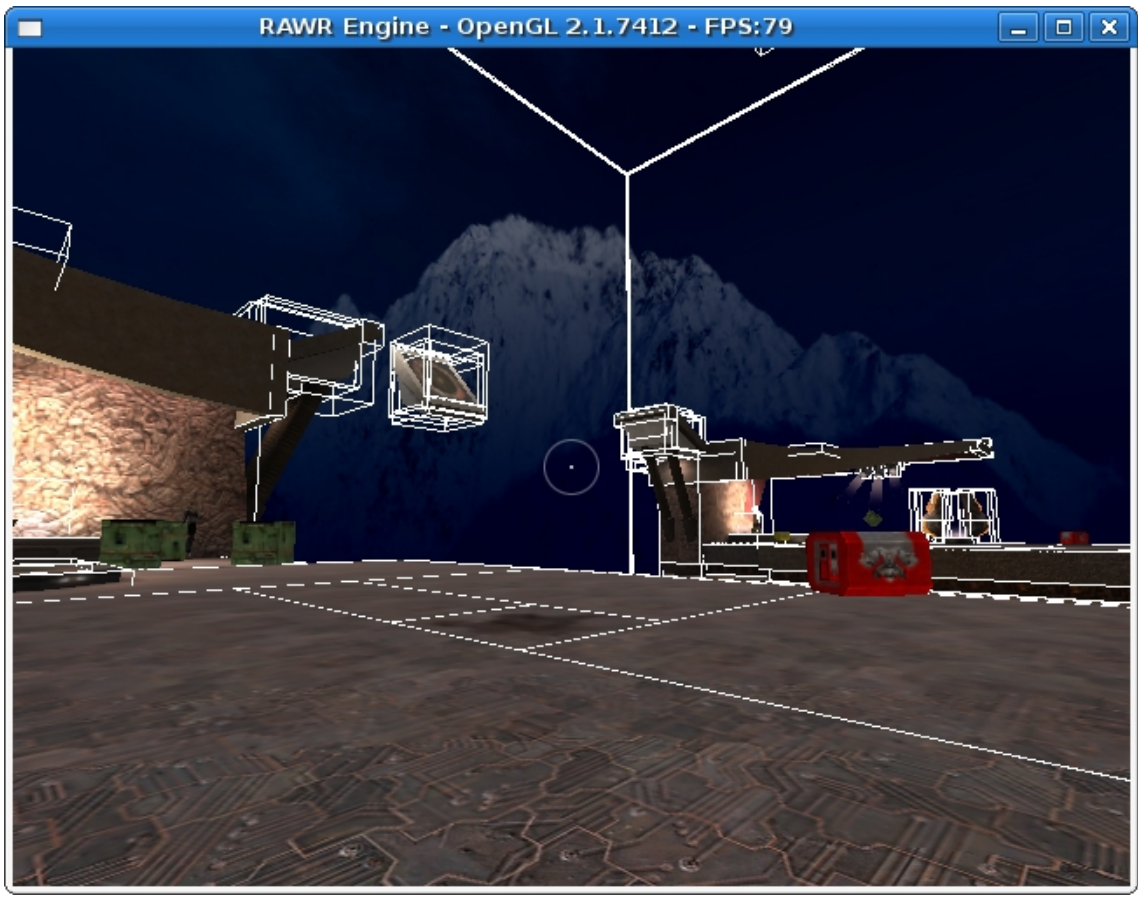

Diagram 2: The white boxes are the AABB of the static physics objects

A physics engine makes very little or no use of the other components of a game engine and this makes it perfect to form the foundation of a game engine. It is much easier to swap the other components of a game engine for an alternative than it is to swap physics engines. Having a well designed and fully featured physics engine is therefore essential to ensure the longevity of the engine. The render and artificial intelligence components will access the physics engine more than any other component.

3.1.2. Renderer

A renderer or rendering engine provides the necessary features required to manage and display a scene. The need for a specialised rendering engine came from the increasing complexities of game graphics. A specialised engine was needed to encapsulate the graphics related technology and provide highly

tuned tools. One of the most important features offered by a rendering engine is the scene manager. The scene manager is a data structure that is used to represent and organise all graphical items in a scene. Specialised scene managers exist for the various types of scenes. Items in the scene are typically referred to as scene nodes. Some of the most notable scene managers are the Binary Space Partition (BSP) Tree and the octree. These advantages and disadvantages of both these scene managers will be examined in the implementation section.

The rendering engine makes heavy usage of the physics engine to render game objects and special effects when collisions occur. The rendering engine also accesses Graphical User Interface (GUI) systems which are not part of the scene. The rendering engine also has the capability of managing nodes that are not equally represented in the physics world.

3.1.3. Entity

The entity object is the core game object that can be represent in both the physics engine and in the rendering engine. For example an entity can be any object that should physically interact with other objects and the world and should also be displayed graphically to the user. An entity therefore requires internal data structures necessary to be represented in both of these engines. It also serves as a framework to extend to add extra game specific entities. An entity does not use the other systems of the engine directly but is used by them. It serves as data storage for a game object with minimal functionality that applies only to itself. All extended objects should stick to this strict design decision. The reason for this design decision is to decrease the size of an entity in memory. There could be thousands or tens of thousands of entities in the game at one time. This requires a lot of memory and is why it is important to only include the data needed to represent its state in the game world.

3.1.4. Character

The character is an extension of the entity object. Its purpose is to provide extra functionality and state data to represent a game character such as the player or a monster. A character object needs to expose an interface to control its movement in the physics world. Standard operations such as rotation, movement along various vectors and jumping should be supported. The character object is also a framework for building game specific characters.

3.1.5. Brain

The brain object is the control logic and intelligence of a character. It can access the controlling interface of the character, update animations and perform general control logic. The brain object is the ideal place to perform artificial intelligence processing. A brain should at least have the ability to manipulate the standard character object. This design decision was made so that any brain can control any character. More specific brain objects can be created to control more complex extended characters.

3.1.6. Entity Manager

The entity manager is responsible for managing a group of entities. It is the memory manager for all entities and its job is to ensure that a pool of available entities exist in memory so that entities are fetched from the pool rather than being created when needed.

The need for a custom entity memory manager can be explained by a simple example. Every time the player clicks the left mouse button the gun fires a bullet. Where are these bullets created in the game engine? They could be created on the fly, loading the necessary game assets and allocating memory every time the gun is fired. This process involves a lot of I/O operations which are slow. This slow down might not be visible with one player armed with a hand gun but give a network of players a few machine guns and the performance impact will be severe. Therefore there needs to be a better way to

create and use entity objects so that I/O operations do not impact the game play. The entity manager is used to create pool of entities from a prototype entity. The prototype is cloned a requested number of times to create a pool for that type of entity. A specific pool should also be able to be increased in size, decreased in size or deleted. Most importantly entities need to be accessed. Going back to the example of shooting bullets every time the gun is fired a bullet is simply fetched from the entity manager instead of being created.

The life cycle of an entity and the functionality of the entity manager can be explained as follows. After the prototype has been given to the entity manager along with the number of items to be created the entity manager creates the entity pool for the specific entity type. When the entity pool is first created all the entities are available and are marked as dead. When an entity is requested from the entity manager one can only be returned if the pool has an available entity. If an entity has been returned it is put onto a ready queue. This ready queue exists so that entities can be added to the game only in between a physics engine tick. This is to ensure the stability of the engine. Adding an entity to the game usually requires the entity to be put into the physics engine and the render and for initial parameters to be set. Once an entity has been added to the game it moves to a list of live entities. At the end of a physic's engine tick the live entity list is scanned to find any entities which need to be recycled. An entity is recycled by putting it back into the related pool so that it can be reused in the future. When the entity manager is destroyed it should delete all entities in the pools, ready queue and live entity list. In conclusion the entity manager allows for the effective management of entities to reduce the effect of I/O operations on the game engine's performance.

3.1.7. Camera

The camera is the viewport to the game world and is needed to visualise the game. It also plays an important role in processing which scene nodes are visible and which ones don't have to be rendered. For a camera to be fully functional it should have the ability to be added statically to the scene or to be

attached to a dynamic object such as an entity. Allowing a camera to be attached to an entity and use its view vector to determine the visible scene takes away the need to manually manage a camera's movement and rotation. It also allows the game world to be viewed from the perspective of any entity whether it is the player, a monster or even a bullet.

3.1.8. Map

The map is a data structure to store the predefined placement of game objects and their attributes. Maps are needed to save created worlds. These game objects can be static physics objects or dynamic physics objects such as monsters. The attributes of the object are usually map specific and are usually properties to describe the type of material and how it should be displayed.

3.1.9. State

The engine state represents a specific chunk or division of game logic. It is a building block of the game that is a grouping of similar aspects in to one logical area. For example a state would be created for the main menu, for the game play, high scores or map editor. It allows the games complexity to be divided and for designers to be able to work on specific aspects of a game without knowing the internals of another state. States should have the ability to perform custom draw steps and process events such as input. The physics engine will also call the state in between ticks to allow the state to perform physics engine related tasks while it is safe to do so. Additionally the state must be able to paused and resumed. This is useful to pause and resume the game. This engine design allows for only one state to be updated at a time.

3.1.10. State Manager

The state manager manages all the created game states. It allows states to be added, deleted and for the current state to be set. The current state is the state that the engine is processing at that moment. A state change can be requested from anywhere and as long as the state exists it will become the current state

the engine processes. The state manager is only aware of its contained states and accesses no other engine components.

3.1.11. Engine Kernel

The engine kernel is the object that is needed to group all the engines sub components together and provides a framework to construct games. The engine's game loop that steps the game forward is contained in the kernel. The game loop begins execution once the engine enters its run state and should have the ability to be paused and resumed. If the game is minimized or loses context it is necessary to pause the game loop. The kernel uses a multithreaded game loop that is more complex than the traditional sequential game loop. The engine will continue to update and draw the game as long as the window has focus and the engine is running. The design decisions around the engines game loop will be discussed later. If a game state wants to exit the game it needs to be able to set the engine's running state to false. The engine will then exit at the end of the current game loop iteration. The kernel is therefore important for the encapsulation of the other engine components. It also contains the game loop which is the controlling hand of the engine.

3.2. Potential Areas for Threading

There are many ways that threading can be added to the game engine design above. It is crucial to a design game engine with threading from the beginning and that is exactly what was done for this project. This section will explore various essential threading patterns and tools for engine design and then delve into various sections of the engine to examine the use of threading. First the alternative forms of threading that are possible are discussed then various areas where threading can be applied are examined.

3.2.1. The Concept of Threads

Threads allow a program to split a task into different paths of execution. When a program starts up it performs its execution in the main program thread.

From this thread new child threads can be created or 'forked' to perform additional tasks. Only one thread can be executed on a single processor core at a time. Having more cores or processors allows for more threads to be executed simultaneously.

3.2.2. Coarse Threading Approach

The coarse threading approach is also known as task parallelism. Task parallelism involves splitting the problem into separate tasks that execute independently and asynchronously. The engine that was designed implemented task parallelism as the main threading approach. Each task executes at the same time the others do and access to shared data needs to be protected.

3.2.3. Fine Threading Approach

Fine grained threading is also known as data parallelism. Data parallelism is applying the same operation to a large group of data that has no dependencies amongst other data in the group.

```
for(int i = 0; i < brains.size(); i++) {
     brains[i].think();
}
```

The above example of code loops through all the brains and calls the think method. If the think method does not depend on any other brain each one can theoretically be run at the same time. This can be done by dividing the loop up amongst a number of threads resulting in a speedup equal to the number of threads used. The diagram below now shows the flow of execution.

3.2.4. Thread pool

The thread pool pattern involves a number of worker threads that exist in a pool waiting to process jobs. This is an important tool that uses the available hardware in an effective time sharing manner. Thread pools are useful for

when more threads than processors exist and all need to receive an equal share of processing time. In multithreaded game engines this is more often than not the case.

The life cycle of a threading jobs and the thread pool is similar to that of the entity manager and entities. Jobs fill up in a ready queue and request the use of a thread from the thread pool. If a thread is available it is given to the task and becomes unavailable until it is returned to the pool. Otherwise the thread waits until a thread becomes available. Once a job has finished its execution the thread is recycled and added back to the pool. Typically the thread pool design is used when more jobs exist than threads. If no jobs exist the threads will sleep until woken up. Thread pools can be very simple in implementation or can be complex threading schedulers that process jobs according to priority or share the available threads amongst the tasks in a popular time sharing scheduler manner. This is necessary when the threads out number the available processors. If a large number of jobs need to have an equal share of the processing power it is best to use the timesharing approach. Therefore thread pools are a necessary part of not just game engines but multithreaded hardware in general.

3.2.5. Task Parallelism of the Physics Engine and the Renderer

The area of the engine that would benefit most from multithreading is placing the physics engine on its own thread and the renderer on another. The reason why is because these two components are the most intensive components of the game engine.

This is task parallelism as the physics engine and renderer are separate tasks that have been dedicated entirely to their own thread. This is the threading technique that was used in the design of the project's game engine. These two components of the engine easily cause the most strain on the system. This threading technique would theoretically allow updates to occur and be displayed at twice the speed when running on multi-core hardware rather than single core hardware. A problem arises though when both the physics engine and the renderer want to access the same piece of data. The renderer

needs access to the physics engine but not vice versa. The renderer only needs access to an entity's position and orientation therefore this is the only shared data that needs protected access.

3.2.6. Data Parallelism of the Physics Engine

Performing data parallelism within the physics engine was not implemented in this project due to the complexities of this task. Furthermore the payout versus the work it requires to put threading into an existing renderer is not worth it. The rest of this section explains how the task of adding data parallelism to the renderer and what the likely affects would be.

The work of solving each physics engine tick could be divided amongst a number of threads. The task is made more complicated because it is possible for some of the threads to depend on others due to the nature of the physics world. This makes the solution not a straight forward data parallelism case but requires more complex solutions. What has to be done to solve this problem is to split up the world into areas that have little or no dependence on surrounding areas. The more challenging part is coming up with the algorithm to perform this. If the algorithm was inefficient and resulted in the divided areas depending heavily on other areas there would be a greater chance of threads waiting on locked mutexes and a greater chance of cache misses occurring. This is a major concern with multithreaded software development as processors benefit immensely from making use of high speed access to data that is in cache. If the data that is needed is not in cache, a cache miss occurs and it is expensive to bring the data in. The chance of a cache miss occurring is high when multiple cores are working on different parts of the same problem because the data might exist in another core. Therefore the need to decrease dependencies amongst the threads is essential.

3.2.7. Data Parallelism of the Renderer

The renderer has great potential to benefit from threading. Performing the movement of scene nodes, the updating of animations and the traversal of the

scene manager are just some jobs that do not have any dependencies. Therefore these areas could benefit from data parallelism. These threading techniques were not applied to the renderer as it was out of the scope of the project. Furthermore data parallelism can be performed at the graphics API level to allow the efficient usage of multithreaded Graphics Processing Units (GPU).

3.2.8. Task Parallelism of Artificial Intelligence

Artificial intelligence is the easiest and most beneficial area to introduce threading into a games engine. This is because artificial intelligence generally performs a far greater percentage of processing compared to the modification of game data. This is especially true for neural networks and genetic programming. Using a thread pool that expands to the amount of cores available is best suited for expanding artificial intelligence onto increasing multi core hardware. In the future when the number of cores becomes larger, individual entities could be assigned to their own thread allowing for more complex and realistic artificial intelligence behaviours to be performed.

3.3. Split the Project

Designing and creating a games engine is a complex task but fortunately there are very obvious divisible components of the engine. The project was split into two halves. The one half contained the physics engine and all related objects, the character and sound. The other half contained the renderer, map, state, state manager, GUI, brain and engine kernel. Joint work was performed on both the entity and entity manager because these objects depend on both the physics engine and the renderer. In this project the implementation of the second group is discussed with the most focus going to the game kernel.

The importance of using version control management software was realised early in the projects development cycle. Not only did version control prove effective for backing up multiple versions of the software it also allowed the software to be developed by multiple developers and was crucial to the

projects success. Subversion was the version control system that was used for this project. Some of the everyday features that were used were the committing of changed files to the repository and updating to the latest version of the project.

3.4. The Use of an Iterative Development Process

The engine was developed using an iterative incremental software development process know as scrum. Weekly meetings took place during the development of the engine to assess the progress of the last iteration. From the very start of the project the engine was developed by adding to the existing version. The overall design was not known from the beginning but developed out of the incremental process and concepts that were tested in the prototype went on to form part of the finished engine. The prototype was very successful at showing that the physics engine could be run on one thread and the renderer on another. The pressure of achieving a speedup when running on multi-core hardware was lessened early on as the multithreaded prototype achieved significant performance increases over its single threaded equivalent.

4. Implementation of the Game Engine

4.1. Use of open source software

The design and implementation of a games engine from scratch is a mammoth task that involves many sub components. For this reason this project aimed to explore the usage of open source components to help decrease development time and to not have to reinvent the wheel. These components would perform the building blocks of the engine. The final assessment of the project would be to show if open source libraries could be integrated to produce a games engine in a short period of time.

4.1.1. Advantages and Disadvantages of Open Source Software

Open source software has a number of advantages and disadvantages. Probably the biggest advantage is all the code is available and can be modified by anyone as long as they obey the license. This gives open source software the advantage of having bugs fixed relatively quickly. Support for the more popular open source projects is generally very good because of the rich communities that have formed. Therefore using a well established open source library is important to make development easier. Open source software has a reputation for having bad documentation, and some projects are plagued by bad designs. This is slowly becoming a thing of the past since most of the large open source projects adhere to strict practices, receive funding from commercial entities and are developed by veterans of the field. This project used numerous open source technologies. The strengths and weaknesses of each piece of software will be explored and alternatives suggested if possible.

4.1.2. Bullet Physics Library

The Bullet physics library is a well established open source physics engine. It has been founded by Erwin Coumans one of the previous developers of the commercially successful Havok physics engine. The library is published under the zLib license which also allows it to be used for commercial projects. Bullet has a large online community with a very active forum. When requiring about threading the bullet physics engine on the forum the lead developer was very quick to offer his help and suggestions [11]. The other members of the forum have also proved to be helpful. The developers release regular updates to the engine and the documentation and tutorials are complete. The API is designed using an object orientated approach and uses call-back functions to do user specific functions. Bullet is used in numerous commercial projects and appears to be the leader in open source real-time physics. The main classes that a developer will rely on when using Bullet will now be explored.

The btRigidBody is the core class to handle game objects. It represents the objects position, orientation and velocity. It also contains the object collision shape that is used to calculate interaction with other btRigidBodies.

The btCollisionWorld is the world in which btRigidBodies live in and collide with other btRigidBodies. Bullet has a number of sub classes to this class. There is a btSimpleDynamicsWorld which merely serves as a unit test. The btDiscreteDynamicsWorld was used in the implementation of the game engine that was produced in this project. It serves most general game purposes. The two sub classes are more complex but are used for specific cases. The btContinuosDynamicsWorld adds optional continuous collision detection for fast moving objects. This collision world is very useful for fast paced first person shooter games with lots of action. The final btSoftRigidDynamicsWorld adds support for soft body objects. This allows objects such as sponges or rubber to be realistically modelled.

4.1.3. Irrlicht

Irrlicht is an open source graphics renderer that has been around for many years. It was originally the dominant open source renderer but now sits in the shadow of Ogre3d [12]. Irrlicht was well suited for this project because of its relatively fast learning curve and easy setup. It is a perfect tool to use for prototypes and projects that have short development time. The available tutorials were sufficient enough to setup simple graphical environments. Irrlicht tries to be more than just a renderer at times. It provides basic file system features such as managing file archives, reading and writing xml files, loading many different image and mesh formats as well as saving image formats. The API has a built in event manager to receive keyboard and mouse events. There is also a skinable Graphical User Environment API for creating interfaces. All these sub components are useful but feel incomplete. The event manager is part of Irrlicht's main run loop but an event manager would be best suited running in its own thread. The event manger's features are quite complete besides the lack of support for joysticks or gamepads. File access should also be run in its own thread rather than in the Irrlicht main loop. The graphical user interface that is included is a nice edition and has basic

elements for building interfaces. The elements are in no way complete and the availability of community built skins are few. There are definitely more complete graphical user interface API's out there. Irrlicht unfortunately had no support for quaternions and this made it hard to interface directly with Bullet which did use quaternions to represent an objects orientation. Despite these problems Irrlicht was still well suited for this project. It was easy to setup a basic scene and use the Irrlicht built in event receiver, file manager and GUI.

4.1.4. No Common Data Type

Both Irrlicht and Bullet have defined basic data types and data structures for their libraries and this poses a problem when using them together. For example when getting a btRigidBody's matrix from its world transform this has to be converted to an Irrlicht matrix to update the entity's scene node. This step is necessary to display the entity in the up to date place as it is represented in the physics engine. The solution was an easy conversion step but the affect is a lot more severe. The affect of the conversion step will be examined later in this paper. This is the conversion step below.

```
/*
 * Updates the Entity's scene node to reflect its btRigiBody transform
 */
void Entity::updateSceneNode() {
    //if the node exists
    if (node) {
        //get the current motionState
        ((MutexMotionState*) getMotionState())-
>getCurrentTransform(transform);
        //get the OpenGL matrix
        //this is done because it is quick access to the rotation
        //irrlicht doesn't use quaternions so bullets roation methods
        //can't be used
        transform.getOpenGLMatrix(matrix);
        //create an irrlicht matrix
        irrlichtMatrix.setM(matrix);
        //set the translation and rotaion
        node->setPosition(irrlichtMatrix.getTranslation());
        node->setRotation(irrlichtMatrix.getRotationDegrees());
    }
}
```

The comments above explain the process clearly. The issue with this method is it has to be called on each entity before it is drawn. This is a relatively expensive operation that can be called thousands of times depending on the

size of the scene. The solution would be that both the physics engine and rendering engine should use the same shared data storage. This will be a common problem when using many different API's to build a game engine and will require some customisation of each component to get them to share common data storage. This is not the case when designing and implementing a complete games engine from scratch or even just making use of a 3rd party physics engine and building the game engine around its data storage.

4.2. Physics Engine

The physics engine implemented is a wrapper for the necessary Bullet objects that are needed to create and manage a discrete dynamics world. Bullet requires a collision configuration object be created and an axis sweep constraint solver and a collision dispatcher. With the exception of the collision dispatcher the default implementations were suited for this design. A custom collision dispatcher had to be designed to allow collision callbacks to be created.

The physics engine is responsible for ticking the world forward. This involves calculating all the resultant forces on rigid bodies, moving the rigid bodies in the world, checking for collisions with static bodies and other rigid bodies, correcting the position, orientation and velocity of the rigid bodies if a collision occurred and finally calling callback functions specified by the user.

One of the most essential callback functions is the one that allows the user to process application specific logic after every physics engine tick. It is crucial that game logic is only performed during this on tick call back function as performing it during a physics engine tick will result in an unstable or broken world. The callback can call many on tick methods in many objects as demonstrated by the diagram. Changing the gravity of the world is a good example of how the physics engine could make use of the tick callback.

The coarse threading of the physics engine and renderer was initially done by placing mutexes in Bullet's API. This protected the access to a rigid body’s motion state. Towards the end of development a custom mutex motion state

was created that did not require Bullets default API to be modified. The physics engines update method that ticked the game forward was placed in its own thread that runs alongside the renderer. The implementation of the threading of the physics engine will be explained in more detail when covering game loops later in this section.

4.3. Renderer

Like the physics engine the renderer implementation is a wrapper for the necessary Irrlicht objects that are needed to create, manage and draw a scene. The renderer creates a video device where a scene can be viewed. Additionally it provides access to a scene manager. This is the heart of the renderer as it performs the culling of non-visible objects, the addition and removal of scene nodes and the rendering of visible scene nodes. Furthermore the renderer provides a means of loading resources such as meshes and textures and a basic file manager for browsing archives of assets. Scene nodes can be created from meshes and different textures and material properties can be assigned to nodes that were constructed using the same mesh data. The renderer also provides a Graphical User Interface framework to create, manage and draw interfaces. The built in event receiver of Irrlicht was used to process input events for the engine and the renderer provides a means of gaining access to this event receiver.

4.4. Entity

The entity object is the core engine object to represent objects in both the physics world and in the renderer. The implementation of the entity object extends the btRigidBody object. This design decision was made rather than encapsulating the btRigidBody in a wrapper class so that when a btRigidBody received collision callback it could be casted to any necessary inherited objects. This was a very important aspect of the engines implementation from the beginning as it is necessary to access an entities data after a collision has occurred. Since the entity extends the Bullet btRigidBody it already has all the necessities to exist in the physics engine. All that has to be done is set the

weight of the btRigidBody and give the rigid body a collision shape. This is the shape that is used for calculating how a rigid body collides with other rigid bodies. The shape and weight also determine the objects inertia and weight distribution which affects the way the body moves in the world. In order for the entity to be visible to the user on the screen it needs to be in the scene manager. Therefore access to a scene node is included in the entity as well as access to the mesh and texture needed to create the scene node. The entity also needs a means of updating the objects position in the renderer if the two engines don't interface directly with each other.

4.5. Character

The character implementation is an extension of the entity. It adds functionality to allow a btRigidBody to behave more like a game character. Conventionally game characters are static objects in the physics world. The downside of this is that the character is not affected by the collisions with dynamic objects. It would be more realistic to have a box hit the character and knock the character back. If the character was a static object this would have to be done manually but if it was a dynamic object it would have this behaviour already. Therefore it was decided to model the character around a dynamic physics object but this comes with its own set of problems. The character can now rotate in all directions. So if a rocket exploded next the character it could send the character spinning, this could be incorporated into the ragdoll nature when the character dies makes general usage troublesome. A simple fix would be to make the character immune to falling over by setting the angular factor of the rigid body.

A very important tool for proper character motion and interaction in the physics world is the use of ray tracing. A simple example of when it is used is for jumping. A ray is cast to a certain point below the character if it intersects with a rigid body the player can jump. A ray cast vehicle has become the standard for creating fully dynamic vehicles in games. This project showed the importance of ray tracing as a tool for adding functionality to a game.

4.6. BSP tree

The BSP tree was made popular by the Quake franchise that used the data structure as a scene manager and for collision detection. A BSP tree is created by recursively subdividing a space into convex sets and ordering the sets as a tree structure. The structure works very well for indoor maps as rooms can easily be defined as convex sets. The data structure was stored along with additional data to make the map format. Unfortunately BSP trees require a compilation step to build them making it difficult to manipulate them in real time. There are many BSP maps already available for free or by using existing quake based games. This has made them popular for quick and dirty development. The downside in constructing new BSP maps requires the use of specialised map tools and the same task cannot be completed in a modelling tool. More popular indoor map formats are portal systems or octrees. Portal systems group items into rooms and use a linking system to move between them.

The implemented map class uses the Quake3 map format to construct btRigidBodies and scene nodes from the BSP tree information contained within the map. The vertices of each plane of the map are used to create the rigid bodies that make up the static map. The Quake3 BSP tree contained in the map file was never used for organising the scene manager so a large portion of the map information was useless. The reason it was not used is because more advanced scene manager algorithms exist that offer improved features as well as performance. The BSP map format is therefore not necessary and a custom map format that purely specifies map data without a structure would be more useful.

4.7. Octree

An octree represents the map by dividing the scene into cubes that enclose geometry. Each internal node of an octree can have up to eight children. And each child can point to another node allowing an octree to provide greater accuracy for areas of a map that require it. An octree can also be seen as a 3

dimensional version of a quad tree which splits the map into squares. An octree can easily be manipulated in real-time. This allows map builders to use conventional modelling tools to create a map and the octree can be automatically calculated from the geometry. Furthermore an octree based scene manager is suited for both indoor and outdoor maps. This makes the octree a more easy to use and general purpose scene manager and is why it was used in the implementation of this project's game engine.

4.8. Game loop

The game loop is the repetitive task of moving the game forward. On a simple scale it involves handling inputs, receiving network packets, updating the game, doing artificial intelligence processing, sending network packets, playing sound effects and drawing the game. To explain the game loop in a simple manor focus will be put on only two tasks, updating the game and drawing the game. A simple single threaded game loop will first be explored and expanded. Once the foundation concepts have been laid down the multithreaded designs will be explored. The simplest game loop implementation is as follows.

4.8.1. Frames Per Second (FPS)

FPS or Frames Per Second is the measurement of the amount of game frames that are displayed per second.

4.8.2. Game Updates

Game updates are the number of times per second that the game state is updated or ticked forward. The physics engine implemented maintained an internal clock for ticking the engine forward.

```
/*
 * Updates the dynamics world
 * Calculates the passed time in microseconds and steps the world forward
 * with this. if the time passed is less than the fixed timestep the
 * dynamics world interpolates the position of all the objects.
 */
```

```
void PhysicsEngine::update() {
    dt = clock.getTimeMicroseconds() / 1000000.f;
    resetClock();
    dynamicsWorld->stepSimulation(dt, 10);
}
```

The stepSimulation(dt, 10) method steps the simulation forward by the passed time. The second parameter of this method is how many steps the physics engine is allowed to catch up on if it is running behind. If steps are running behind it is a good indication that the hardware is not sufficient to run the game in a playable manor.

4.8.3. The time independent game loop

```
bool running = true;

while(running) {
     update();
     draw();
}
```

This loop is very easy to understand, as long as the game is running continue to update and draw it. The most obvious problem is that if the game was running on faster hardware the loop will execute more often. The effect of this can be seen when running old Dos games on modern hardware. The game runs so fast that it is unplayable. The reason this happens is because the loop is time independent. A time dependant game loop updates the game depending on the amount of time that passed since the last update. In the above game loop the game updates are equal to the number frames per second displayed. The game updates are in a sense dependant on fps and fps on the game updates. The next step in the game loop progression would be making it time dependant.

4.8.4. The Time Dependant Game Loop

```
bool running = true;

long lastTime;
long currentTime = getTime();

while(running) {
     lastTime = currentTime;
```

```
        currentTime = getTime();

        update(currentTime - lastTime);
        draw();
}
```

This game loop takes time change since the last update into consideration. Game objects are only updated by the amount of time that has passed since the last update. If the same old school Dos game implemented this game loop it would still run at the same speed on today's fast hardware as it did on older hardware. An important thing to note is that the game updates are still dependant on FPS and vice versa. Another problem with this game loop approach is that it is still prone to problems on both slow and fast hardware.

On slow hardware there is a chance that the game will suffer from lag. This is because the game update step takes longer to complete than on faster hardware. For example a slow machine performs only 10 updates while a fast machine performs 60. Obviously the time dependant game loop already compensates for this to a degree. The problem is that because larger time changes take place in between the game updates the affect of the time change on game updates cannot be guaranteed. This is especially true for physics engines. If a bullet was updated with this small time change it moves a small distance. If the same bullet was updated with this large time change it moves a much larger distance. This large travel distance could result in the bullet passing through other objects. If continuous collision detection was used this wouldn't be a problem, but this method of collision detection is expensive and is not the right answer. The game could just have a strict minimum hardware requirement, this is an option but the game loop is nondeterministic and needs a better solution.

On fast hardware the effect is just as damaging. The problem arises not from the theoretical design of the algorithm but from the numerical inaccuracies of computers. Floating point data types only represent values up to a certain precision. And some values cannot be represented in binary. The most common example of this is 0.1 which cannot be represented with any level of precision. The binary representation has a "1100" sequence continuing infinitely. The rounding errors introduced like the 0.1 example are small but

they accumulate with every game update. The problem gets compounded even further when these values are used to calculate other values that are then reused in the next game update.

Although the implementation of this loop is still simple and it does solve some problems it is not a complete solution and can introduce errors into the game.

4.8.5. Constant Game Updates and Constant FPS Game Loop

The next logical progression in the game loop would just be to allow all the updates to happen at a fixed rate and then to be drawn. The frames per second are still dependant on game updates and vice versa. The design of this game loop is as follows.

```
const int GAME_UPDATES = 40;
const int SKIP = 1000 / GAME_UPDATES;

int sleepTime = 0;
long nextTick = getTime();
bool running = true;

while(running) {
     update();
     draw();

     nextTick += SKIP;
     sleepTime = nextTick - getTime();

     if(sleepTime > 0) {
            sleep(sleepTime);
     }
     else {
            //running behind
     }
}
```

This solution is still easy to understand and simple to implement. The number of game updates to be carried out per second is the first thing that is chosen. If more updates occur per second the game will look smoother but also have higher minimum requirements. Choosing the correct number of updates is also very dependant on the type of game being produced. If it is a fast paced action game it will need more updates than a slower puzzle game. Fast paced games require greater precision and having less game updates makes it easier to notice artefacts such as jittery or blocky motion. This defect occurs because

game objects can move large distances in the game world and the more game updates that are used to approximate the movement the smoother the game will appear.

The beauty of this design from a theoretical perspective is that the game state is deterministic if processing power is not an issue and all the game updates are always performed on time. A deterministic game loop allows features such as game replays to be implemented very easily. The game would simply need to record the input events that took place during each update to have a perfect recreation of the game play. This feature can allow the game play to be rewound, paused, slowed down and opens a door up to use the engine as a basic video editor. The problem is that no one has the perfect hardware and there is no way to guarantee that all the game updates are performed on time. Even if the hardware of the machine way surpasses the games minimum requirements there is no way to guarantee that a large game scene of thousands of items won't take place. This might not be the average amount of items in the scene but merely an outlier of what is possible. The game will therefore react in the same manor it would on slow hardware.

A game that uses this game loop will run in an unpredictable manor on slow hardware that cannot handle the defined game updates per second. Game updates may be skipped or happen out of sync. The most logically way to fix this would be to remember the time needed to catch up by and then perform enough game updates to catch up. This still maintains the deterministic nature of the game loop but the game will be unplayable.

On fast hardware the game will run perfectly but many clock cycles will be wasted when the game loop is put to sleep. The faster the hardware the more clock cycles will be wasted. Another problem is that the frame rate is fixed to the game update rate. On faster hardware the game could draw the scene at rates that far surpass the game update rate. Faster hardware therefore loses out in the visual appeal section.

4.8.6. Constant Game Speed Independent of Variable FPS Multithreaded Game Loop

What the ideal game loop should do is update the game at a constant speed and draw the game as fast as the machine can handle. This game loop works best with a multithreaded design because there are two internal loops to this game loop. The game updates are performed at a constant speed in their own thread. The design of the game update thread is as follows.

```
const int GAME_UPDATES = 40;
const int UPDATE_TIME = 1000/GAME_UPDATES;

long lastTime;
long currentTime = getTime();

bool updatesRunning = true;

while(updatesRunning) {
     lastTime = currentTime;
     currentTime = getTime();

     while(dt > UPDATE_TIME) {
            update();
            dt -= UPDATE_TIME;
     }

     if(dt > 0) {
            interpolate(dt);
     }
}
```

The update loop still updates the game at a constant speed, and catches up any fallen behind updates. This is done in the internal while loop that loops as long as there are UPDATE_TIME sizes of time left over in the time change since the last tick. If smaller amounts of time are left over the updates are simply interpolated. Time changes that are less than the UPDATE_TIME to begin with skip the while loop and go straight to interpolation.

Interpolation is the process of predicting an objects future position using the current position. This might seem remarkably similar to performing an update, but an update is moving from one game step to another. An interpolation is moving in the space between the game steps. Many interpolation steps can be performed in between two game steps. An interpolation does not represent an actual movement of the game object but

rather an estimation of the objects position purely for the purpose of drawing the object in a smoother manner. This allows the drawing process to happen as fast as possible. The drawing thread design is very simple.

```
bool drawRunning = true;

while(drawRunning) {
     draw();
}
```

Using multiple threads requires data access protection. This is to ensure that variables are not being updated by the update game loop and accessed by the draw game loop at the same time. This can be problematic if the update game loop was in the middle of updating the player's position. Imagine this scenario, the x position of the player is updated and its y and z have not. The draw game loop then draws the player at its current position. Only after this draw step has completed does the update game loop change the players y and z values. Even though this is just one frame there is no way to guarantee that it will not happen in the next frame. The visual defect this causes is that the player will seem to jump around between frames. It is also possible that data may become corrupted altogether causing the game to crash. It is therefore important to protect the access of sensitive data. For the game engine that was designed it was necessary to protect an objects world transform object. The pseudo code below creates a mutex for locking an objects transform during updating and getting the transform.

```
mutex accessMutex;

updateGameObjectTransform() {
     lock(accessMutex);
     ...

     //update the game object

     ...
     unlock(accessMutex);
}

getGameObjectTransform(tranform& t) {
     lock(accessMutex);
     ...

     //update the game object
     t = objectTranform;
```

```
    ...
    unlock(accessMutex);
}
```

This method of the game loop theoretically allows the game to achieve double the number of frames per second when ignoring the overhead of thread management. It has also successfully made the game updates not depend on the frame rate and vice versa. This was the first multithreaded game loop that was produced for the project's game engine. This game loop is still not ideal though, because on fast hardware more interpolations are performed and more frames are drawn. It is not necessary to perform more interpolations beyond a certain point and there is no point to draw the game at a faster rate than the display unit can refresh. Therefore some further optimisations can be made.

4.8.7. Multithreaded Game Loop Optimisations

To optimise the update loop upper limits can be set on the number of updates + interpolations that are performed. This was implemented by putting the physics thread to sleep for a period of time.

```
/*
 * Calls the physics engine update method when it is time to do so
otherwise
 * puts the thread to sleep therby saving clock cycles for other tasks.
 */
void* physicsEngineUpdate(void* parameter) {

    time1 = GetTickCount();
    //how long the thread will sleep for before another physics engine
    //update is performed
    sleepTime = 13;

    //the physics engine game loop
    while (engine->isRunning()) {
        //only update the physics engine if it is running
        if (physicsEngine->isRunning()) {

            time2 = GetTickCount() + 1;

            //if the time passed is more than the sleep time perform
            //an physics engine update
            if (time2 >= time1) {
                physicsEngine->update();
                time1 = time2 + sleepTime;
            } else {
                //put the thread to sleep
```

```
                usleep((time1 - time2)*1000);
            }
        }
    }
    return NULL;
}
```

The same optimisation can be achieved for the draw loop by setting an upper limit on the number of frames rendered per second. These optimisations will result in more clock cycles being available to process other game tasks such as artificial intelligence. This game loop is the perfect next step for modern games engines as it can make better use of today's multi-core hardware. With the exception of the frame rate limiter this is the final multithreaded game loop that was designed and implemented in the project's game engine. The performance of various game loop implementations will be explored later in this report.

4.9. Creating a Game using the engine

To create a minimum game using the game engine that was developed requires the developer to extend the state class to add game specific logic. Add some entities, characters, brains, and load a map. A screenshot of the created game can be seen below.

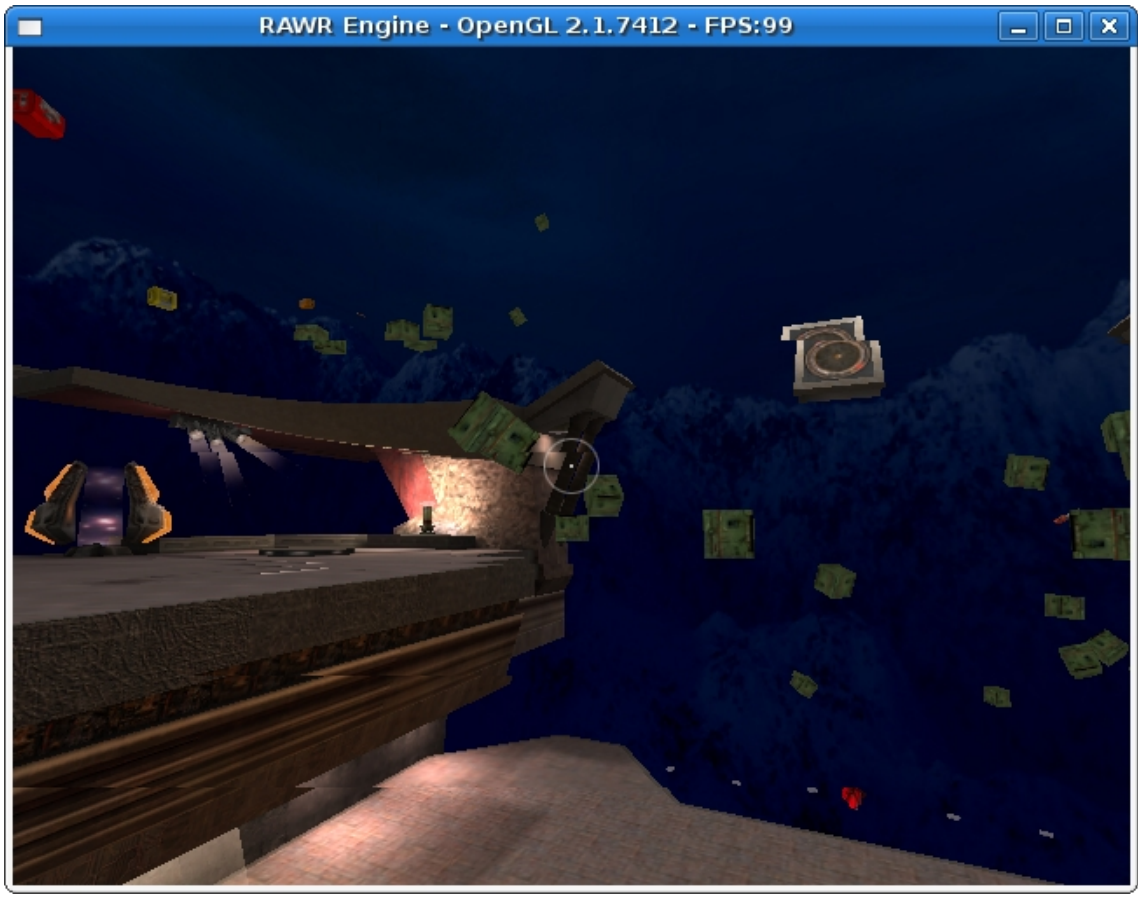

Diagram 3: The test game that was constructed

5. Experimental Design

A number of experiments were conducted on the game engine that was developed to test various design concepts that were implemented. The affect of the various game loop implementations on the engine's performance was assessed. This was done by recording and analysing the FPS of the game engine for each implementation of a game loop. Additionally the processor usage was recorded during the experiment so the strain of the algorithm on the CPU could be analysed. Memory usage would be constant across the various game loops tested as they all used the same memory management techniques. This experiment was important to show the successful usage of multi-core hardware by the game engine.

Experiments were setup by creating a fixed environment that ran for a set time frame. During each run entities were added to put strain on the game engine. Each successive run on a specific game loop implementation added more entities to the game engine in order to increase the strain. The performance of each run was recorded and can be seen Table 1, 2, and 3 of the Appendices.

Various game loop designs were explained in the implementation section. Of those designs three were implemented. All three loops used a constant game speed and maximum FPS that were independent of each other. The first game loop was a single threaded version, the second a multithreaded version and the third a multithreaded version with interpolation limiting optimisations applied.

The experiments were performed on a 2.0ghz dual core Intel CPU, with 1gb of ram and a Nvidia 6600 mobility graphics card.

6. Results/Findings

The results from the game loop experiment were expected and in favour of the multithreaded game loop that had interpolation optimisations applied.

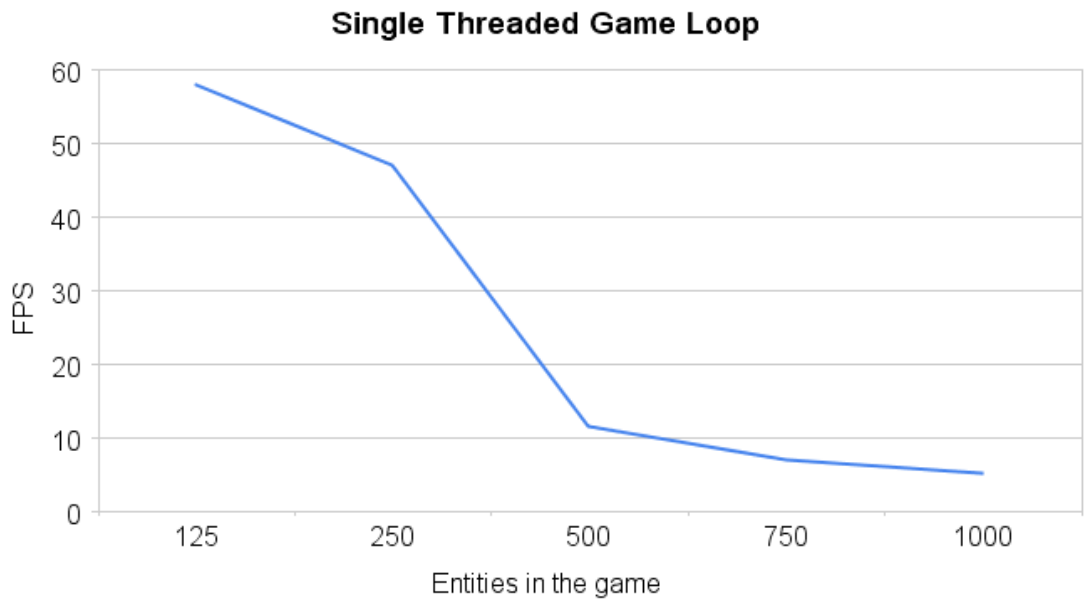

Chart 1: Shows the affect of the number of entities in the game on the FPS for the single threaded game loop

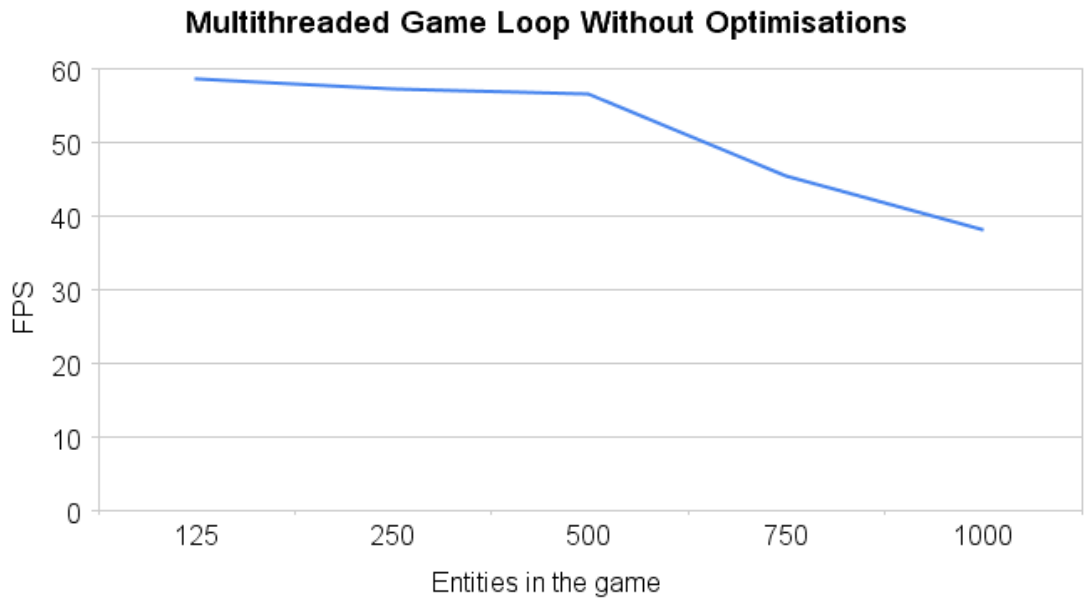

Chart 2: Shows the affect of the number of entities in the game on the FPS for the multithreaded game loop without optimisations

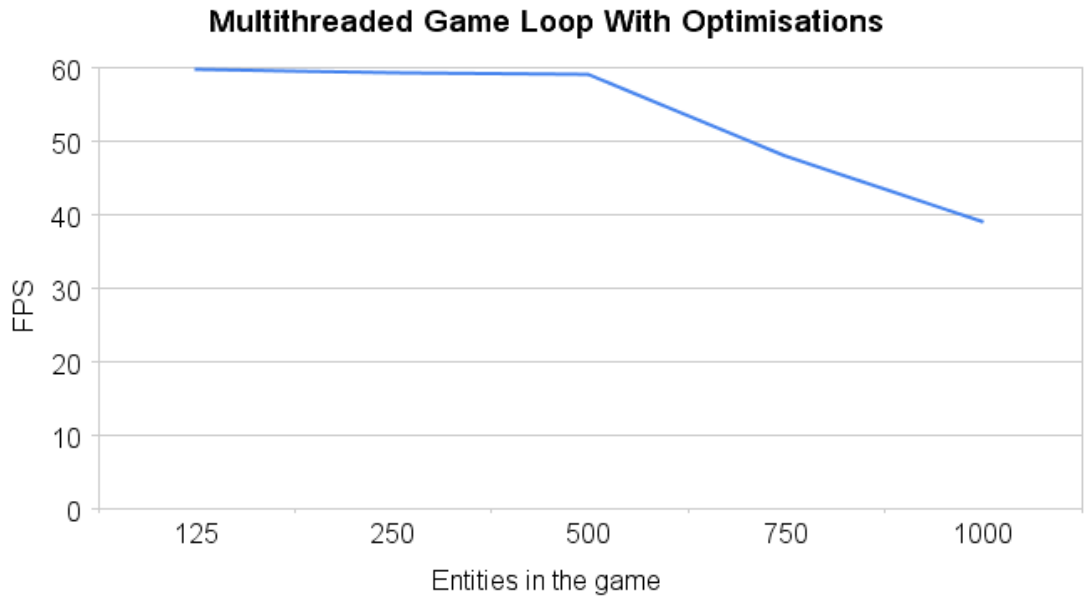

Chart 3: Shows the affect of the number of entities in the game on the FPS for the multithreaded game loop with interpolation optimisations performed

7. Discussion of Results

It is important to note that there was a maximum attainable frame rate of 60 FPS because of the limitations of the laptop screen that the test was performed on. The results are still very conclusive though.

The single threaded game loop is the first to show a decrease in FPS being displayed. It decreases almost immediately and with the addition of more entities the performance of the system degrades rapidly to the point where the game is unplayable. The CPU usage of this game loop is also at 100% of a single core's processing power during all the runs.

The multithreaded game loop without optimisations only begins to show serious signs of performance degradation around the 500 entity mark as shown in Table 2 and Chart 2. Furthermore this implementation does not degrade the game's performance to an unplayable state. Both cores of the CPU are only used at full capacity at the 500 entity mark. This shows a clear cause effect relationship, if more entities are added to the game engine this directly affects the FPS and CPU usage.

The final implementation tested was the multithreaded game loop with interpolation optimisations applied. The game loop degrades at an even smaller rate than the previous two game loops. The overall performance increase is small but with the tweaking of the number of interpolations performed this result could still be improved.

The experiments performed successfully showed the benefits of implementing efficient game loops for extract the maximum performance from the available hardware. The relationship between the number of entities in the system and the resultant FPS and CPU usage was also identified. It is useful to know the point at which performance begins to degrade. This happens around the 500 entity mark for both multithreaded game loops. Knowing this allows game developers to restrict the number of items in the game so that the game can run smoothly on various hardware configurations.

8. Conclusions

The project was successful at designing and implementing a multithreaded game engine. The related works presented in this report were crucial tools used during the design and implementation stages.

The two areas of threading, task parallelism and data parallelism both had their advantages and disadvantages for various areas in a game engine. These areas were explored and the thread pool pattern was identified as a valuable general purpose tool to serve many jobs. The main areas of the game engine to add threading to were the physics engine and the renderer. A task parallelism technique was implemented to place each of these components on their own thread. No data parallelism was performed in these sections because of the complexities of implementation versus the potential performance gain. It was established that task parallelism was the easiest to implement in a game engine. The use of agile iterative development processes and version control software was crucial to the successful management of the project.

In the implementation various game loops were evolved until a custom optimised multithreaded version was settled on. This project would not have been a success if open source libraries weren't used to decrease the risks of the design and implementation. Various implementations of the game loop evolutions were tested to assess the performance of the game engine and the strain on the CPU. The experiments clearly showed that the multithreaded game loops achieved increased performance over the single threaded version. The multithreaded implementations also put less stress on the CPU. The experiments also identified an average amount of entities that could exist in a game before performance would start to degrade.

In conclusion this project successfully showed that a multithreaded modular game engine could be produced using open source software in a limited amount of time. Furthermore the importance of the engine game loop was discussed and the effect it has on the game engine's performance and the strain on the CPU were analysed.

9. Future work

9.1. Data Parallelism of the Entity

It was established in the design of the entity object that it should only include the data needed to represent its state in the game world. Furthermore all the entities are processed in a serial manor within the game engine. Most of the time the updating of entities is a completely independent task and could therefore benefit from data parallelism being incorporated. The entities should make use of the thread pool to perform its tasks. Incorporating this additional area of threading in the game engine should help to improve performance significantly.

9.2. Multithreaded State Management

The engine kernel only processes one state at a time. It would be useful though to be able to process multiple states in the background. An example of where this would be useful would be for loading resources in the background. The state manager should therefore allow multiple states to be processed at the same time. This can be done by the usage of the thread pool or dedicated state manager threads. This new feature will allow the engine to perform background tasks seamlessly and therefore decreasing the time a player has to spend waiting for the game to load.

9.3. Using a More Modern Programming Language

The need for explicit memory management as offered by higher level languages such as c++ is no longer essential to game development. In fact they can often complicate and the development process, increase the time to develop and introduce nasty memory related bugs. The object pool pattern that was used in the entity manager is the ideal way to manage a games memory. It can be created in both memory managed and non memory managed languages. Objects will only be deleted or created in the memory manager during game loading states or in background loading threads. The

reason games have been using memory managed languages for many years is because of the expensive process of garbage collection and the unpredictable nature of when it can be triggered. Modern languages and compilers have highly optimized garbage collection functions that can be finely tuned for specific needs. The D programming language uses garbage collection but allows developers to perform explicit memory management by overloading the new and delete operators or by calling C's malloc and free methods directly. The collector can also be excluded from managing certain memory ranges, can be paused and resumed or forced to perform a full memory collection cycle. These are just some of the options that are available in modern high level languages. It is for this reason that game developers are moving over to C# and making use of Microsoft's XNA framework. C# is a memory managed interpreted higher level language but is proving very successful for game development on both windows pc's and Xbox 360's.

9.4. Replacing Irrlicht with Ogre

Ogre3d has a larger community than Irrlicht and has been included in Google’s summer of code. It has a well designed API that has better support for shaders and basic DirectX 10 support already while Irrlicht has none. The API is also excellently documented, has a better design and offers more advanced tutorials than Irrlicht. Ogre also makes use of quaternions to represent an objects orientation and this makes it easier to interface with Bullet. Ogre has a proven track record when it comes to big software projects where Irrlicht does not. A rough measure of which rendering engine has had greater adoption is by looking at the commercial games that use the engine. This is by no means a perfect measurement as it is not taking into consideration licensing and project lifetime. These factors aside though Ogre has been adopted more heavily by commercial developers. If the game engine is to be expanded and further developed it would be recommended that Ogre be used instead of Irrlicht.

10. References

[1] Scrum. http://www.controlchaos.com/

[2] Bullet physics Library. http://www.bulletphysics.com/

[3] Irrlicht Rendering Engine. http://irrlicht.sourceforge.net/

[4] Tulip, J., Bekkema, J. and Nesbitt, K. Multi-Threaded Game Engine Design. In Anonymous ACM International Conference Proceeding Series; Vol. 207.

[5] Valve Software. http://www.valvesoftware.com/

[6] Jeremy Reimer. *Valve goes multicore.* http://arstechnica.com/articles/paedia/cpu/valve-multicore.ars/

[7] Michael Suess. *Multi-Threaded Challenges in the Game Space – A Conversation with Tom Leonard of Valve Fame.* http://www.thinkingparallel.com/2007/01/06/multi-threaded-challenges-in-the-game-space-a-conversation-with-tom-leonard-of-valve-fame/

[8] Microsoft Xbox. http://www.xbox.com

[9] Sony Playstation. http://www.playstation.com

[10] Microsoft XNA Developer Centre. http://msdn.microsoft.com/en-us/xna/default.aspx

[11] Bullet Physics Library Forum. http://www.bulletphysics.com/Bullet/phpBB3/viewtopic.php?f=9&t=2320&p=9065&hilit=honours+project&sid=627b78a5b6f63aa3fd1df90b94229f7e#p9065

[12] Ogre 3D Open Source Graphics Engine. http://www.ogre3d.org/

11. Appendices

11.1. Tables

11.1.1. Single Threaded Game Loop

Table 1: Single Threaded Game loop		
Entity Count	**FPS**	**Processor Usage**
125	57.87	100.00%
250	47.03	100.00%
500	11.55	100.00%
750	6.98	100.00%
1000	5.17	100.00%

11.1.2. Multithreaded Game Loop without Optimisations

Table 2: Multithreaded Game Loop without Optimisations		
Entity Count	**FPS**	**Processor Usage**
125	58.58	80.00%
250	57.33	90.00%
500	56.63	100.00%
750	45.55	100.00%
1000	38.07	100.00%

11.1.3. Multithreaded Game Loop with Optimisations

Table 3: Multithreaded Game Loop with Optimisations		
Entity Count	**FPS**	**Processor Usage**
125	59.82	50.00%
250	59.28	65.00%
500	59.13	100.00%
750	47.97	100.00%
1000	39.1	100.00%